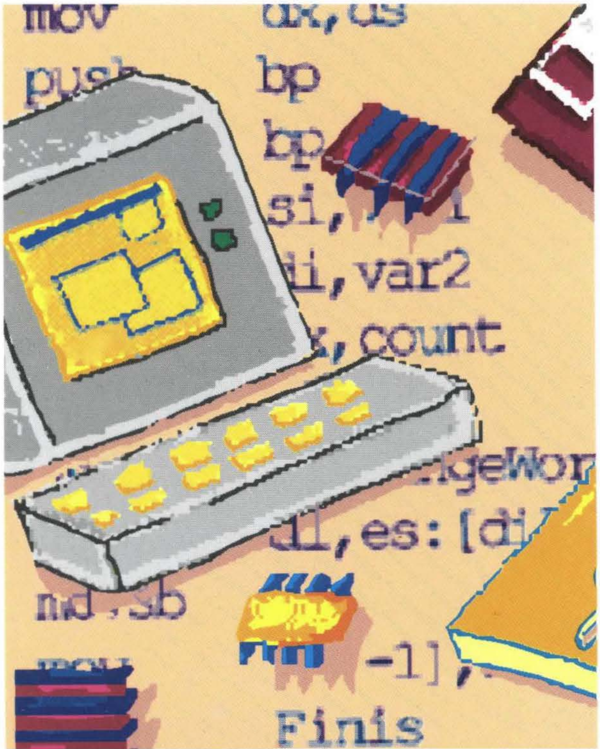
TURBO DEBUGGER®

USER'S GUIDE

BORLAND

USER'S GUIDE

TURBO DEBUGGER®



BORLAND

Turbo Debugger®

User's Guide

Version 1.0

Borland International
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95066-0001

This manual was produced with
Sprint® The Professional Word Processor

Printed in the U.S.A.

10 9 8 7 6 5

R3

Table of Contents

List of Figures

List of Tables

INTRODUCTION

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 and a combination of pull-down and pop-up menus provide a fast, interactive user interface. An online context-sensitive help system provides you with help during all phases of operation.

Here are just some of its features:

- uses expanded memory specification (EMS) to debug large programs
- full C expression evaluation
- full Pascal expression evaluation
- full assembler expression evaluation
- reconfigurable screen layout
- assembler/CPU access when needed
- powerful breakpoint and logging facility
- keystroke recording (macros)
- uses remote system to debug huge programs
- supports 80386 and other vendor's debugging hardware

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 will run on any 80-column monitor, either color or monochrome. We recommend a hard disk, although a two-floppy disk drive will work fine as well.

Turbo Debugger does not require an 8087 math coprocessor chip.

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

Note that when you run Turbo Debugger, you'll need *both* the .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 **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.

A *module* in this manual refers to what is usually called a module in C and in assembler, but also refers to what is called a *unit* in Pascal.

Similarly, a *function* in this manual refers to both a C function and to what is known in Pascal as a subprogram (or routine), which encompasses both *functions* and *procedures.* 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.

Finally, the term *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 user interface, menus, and windows, and shows you how to respond to prompts and error messages.

Chapter 3: Getting Started: 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 Program 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 Source 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.

Chapter 10: Assembler-Level Debugging explains how to view or 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 11: The 8087/80287 Math Coprocessor Chip and Emulator discusses how to examine and modify the contents of the floating-point hardware or emulator.

Chapter 12: Command Reference is a complete listing of all main menu commands and all local menu commands for each window type.

Chapter 13: How to Debug a Program is an introduction to strategies for effective debugging of your programs.

Chapter 14: 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.

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

Appendix B: Turbo Debugger Utilities describes how to use the utilities provided with the debugger. The utility programs include a program allowing CodeView executables to be used with Turbo Debugger, and several others that affect the debugging information appended to .EXE files. There is also a utility called TDUMP that lets you display the component parts of any file.

Appendix C: 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 D: Inline Assembler Keywords lists all instruction mnemonics and other special words used when entering inline 8086/80286 instructions.

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

Appendix F: Hardware Debugger Interface describes how to write device drivers to work with Turbo Debugger.

Appendix H: Prompts and Error Messages lists all the prompts and error messages that can occur, with suggestions on how to respond to them.

Appendix I: Using Turbo Debugger with Different Languages provides several tips when you're debugging programs written in C, assembler, or Pascal.

Appendix J: Glossary is an alphabetical list of commonly used terms in this manual along with short definitions.

Borland's No-Nonsense License Statement

This software is protected by both United States copyright law and international treaty provisions. Therefore, you must treat this software *just like a book* with the following single exception: Borland International authorizes you to make archival copies of Turbo Debugger for the sole purpose of backing up your software and protecting your investment from loss.

By saying, "just like a book," Borland means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another so long as there is **no possibility** of its being used at one location while it's being used at another. Just like a book that can't be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time. (Unless, of course, Borland's copyright has been violated.)

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 "Enter Language Products Forum" 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:

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

You can also telephone our Technical Support department at (408) 438-5300. Please have the following information handy before you call:

- product name and version number
- computer make and model number
- operating system and version number

C H A P T E R

1

Getting Started

The Turbo Debugger Package

Your Turbo Debugger package consists of three distribution disks and the *Turbo Debugger User's Guide* (this manual). The distribution disk contains 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. The Turbo Debugger package also contains documentation on subjects not covered in this manual.

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

Before we get you started using Turbo Debugger, you should make a complete working copy of the distribution disk, then store the original disk in a safe place. Use the original distribution disk as your backup only and run Turbo Debugger off of the copy you've just made—it's your only back-up 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 in the introduction and mail your filled-in product registration card. This enables you to be notified of updates and new products as they become available.

The Distribution Disks

When you install Turbo Debugger on your system, you copy files from the distribution disk to your working floppies or to your hard disk. The distribution disk is not copy protected, and you do not need to run any installation programs. The distribution disks are formatted for double-sided, double-density disk drives and can be read by IBM PCs and close compatibles.

The following files are on the distribution disks:

TD.EXE	Turbo Debugger
TD.OVL	The overlay file containing the menu system
TDINST.EXE	Turbo Debugger installation program
TDHELP.TDH	Turbo Debugger help file
README.COM	Program to read the update file README
README	Last-minute information
TDH386.SYS	The 80386 hardware device driver
TD386.EXE	The program you use for virtual debugging
TDNMI.COM	A utility that enables the handler for Periscope I boards
TDRF.EXE	The Remote File Transfer untility program
TDREMOTE.EXE	The program you use for remote debugging
TDSTRIP	The Symbol Table Stripping utility
TDUMP.EXE	A generic module disassembler utility
TDCONVRT.EXE	A utility to convert CodeView programs to Turbo format
TDMAP	A utility to append .MAP file information onto .EXE files
TDPACK	A utility to reduce the size of the debugging information in .EXE files
TCDEMO.*	The C demo program you use in the tutorial
TCDEMOB.*	The buggy C demo program discussed in Chapter 13
TPDEMO.*	The Pascal demo program you use in the tutorial
TPDEMOB.*	The buggy Pascal demo program discussed in Chapter 13

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 what each one contains.

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 *Up* and *Down arrow* 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 the NMI.

Installing Turbo Debugger

The Installation Disk contains a program called INSTALL.EXE that will assist you with the installation of Turbo Debugger 1.0. There are two options for installation:

Install Turbo Debugger on a Hard Disk System

INSTALL will copy all Turbo Debugger files onto your hard disk and put 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 examples programs into their own subdirectory as well, edit the default example files path *before* selecting `START INSTALLATION`.

Install Turbo Debugger on a Floppy Disk System

This option builds a working set of four Turbo Debugger disks will work on a two drive system. Be sure to have four formatted disks ready before you start. INSTALL builds the following disks for you:

Program Disk	Turbo Debugger main program (TD.EXE), README and README.COM, Turbo Debugger customization program (TDINST.EXE), and the HELPME!.DOC and MANUAL.DOC files.
Work Disk	Turbo Debugger working overlay file (TD.OVL) and help file (TDHELP.TDH).
Utilities Disk	Turbo Debugger utilities (TDSTRIP, TDRF.EXE, TDUMP.EXE, TDCONVRT.EXE, TDMAP, TDPACK).
Examples Disk	Example programs for use with Turbo Debugger.

To start the installation, change your current drive to the one that has the INSTALL program on it and type `INSTALL`. You will be given instructions in a box at the bottom of the screen for each prompt. For example, if you will be installing from drive A, you would enter

```
A:
INSTALL
```

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

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

```
INSTALL -h
```

The TD.OVL File

Turbo Debugger consists of an executable program, TD.EXE, and an overlay file, TD.OVL, which contains the menu system and must be available to TD whenever you use the menus. Both files are required. If you are installing Turbo Debugger on a hard disk system, INSTALL will put them in the same directory.

If you are installing on a two-floppy system, INSTALL will put the overlay file (TD.OVL) and the help file (TDHELP.TDH) on one diskette and TD.EXE on another diskette.

Unarchiving Example Files

The Turbo Debugger UTILITIES/EXAMPLES distribution disk contains several files with an .ARC file extension: TDEXAMPL.ARC, TAEXMPL1.ARC, and TAEXMPL2.ARC. These files contain several other files that have been compressed and placed inside an archive. You can dearchive them yourself by using the UNPACK.COM utility.

For example, entering

```
unpack tdexampl
```

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

INSTALL gives you a choice of copying the .ARC files intact or dearchiving and copying all of the individual files onto your hard disk during the installation process. Note that INSTALL does not unpack the TAEXAMPL1.ARC, TAEXAMPL2.ARC, or CHAPXMPL.ARC files from the UTILITIES/EXAMPLES disk. These files contain example programs for the TURBO ASSEMBLER.

The INSTALL /B Command-Line Option

If you have difficulty reading the text displayed by the INSTALL program, 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 are running on 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.

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 Appendix F for complete information on this device driver interface.

Note: If you have a hardware debugging board (such as Atron, Periscope, 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 Langauge

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 track down! After you have gained a working knowledge of the language, work your way through Chapter 3, "Tutorial," for a quick tour of the major functions of Turbo Debugger. There you'll learn enough about the features you'll need to debug your first program; we'll go into the debugger's more sophisticated capabilities in a later chapter.

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 user interface by reading Chapter 2. 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 12, "Command Reference."

CHAPTER

2

Debugging and Turbo Debugger

There once was a man who believed he never made mistakes. But he was wrong. And that's why we have debuggers.

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

When it comes to programming, stumbling can become a way of life. Very few programmers can ever write an error-free program the first time out the gate. That's nothing to be ashamed of or surprised at. But 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 then correcting errors ("bugs") in your programs. It's not unusual to spend more time on finding and fixing bugs in your program than writing the program in the first place. Debugging is not an exact science; often the best debugging tool you have is your own mind. Nonetheless, there is some advice that can be offered (see Chapter 13), and the 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, realizing you have a problem 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 might 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). You're 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 may need a bit of experimenting to track 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 squash the bug. Now you run the program again, wait for the next error to show up, and start the debugging process again.

Many times this four-step process is accomplished when you are writing the program itself. Many errors of syntax, 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 types of errors and allow you to 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

With the standalone Turbo Debugger, you have access to a much more powerful debugger than exists in your language compiler. (Adding such a feature-full debugger to the program itself would make it too big.)

You can use Turbo Debugger with any program written in C, Pascal, or assembly language using the Borland Turbo products or those from other language manufacturers. (You need to use a conversion utility that we supply before you debug a program written in a Microsoft language, however.) 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 you use the TDMAP utility described in Appendix B.

You can use Turbo Debugger to help with the two hardest parts of the debugging process: finding where the error is, and finding the cause of the error.

Turbo Debugger helps you overcome these debugging hurdles by virture of its extensive abilities to slow down program execution and to examine the state of the program at any given spot. You can even test new values of variables to see how they affect your program. This ability translates specifically into tracing, stepping, viewing, inspecting, changing, and watching.

Tracing You can execute your program one line at a time.

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 a dozen different things: variables, their values, breakpoints, the contents of the stack, a log, a data file, a source file, CPU code, memory, registers, numeric processor info, or program output.

Inspecting You can have Turbo Debugger delve deeper into the workings of your program and come up with the contents of complicated data structures like arrays.

Changing You can replace the current value of 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 progresses.

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 beaver 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 significantly helps you find where it is and what's at fault.

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, Turbo Debugger has at least three things it *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; it would be a waste of memory to include one with Turbo Debugger. 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.
- 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 will not take the place of thinking. When debugging a program, your greatest asset is simple thought. Turbo Debugger is a powerful tool, but if you use it mindlessly, it's unlikely it will 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 while also being easy—dare we say intuitive—to use.

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

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 achieve this goal, Turbo Debugger sports these powerful features:

- Convenient and logical pull-down menus.
- Context-sensitive pop-up menus throughout the product, which practically do away with memorizing and typing commands.
- When you do need to type, Turbo Debugger keeps a list of the text you've typed in similar situations. You can choose from these "history lists," edit the text, or type in new text.
- Full macro control to speed up series of commands and keystrokes.
- Convenient, complete window management.
- Access to several types of online help.

The rest of this chapter discusses these six facets of the Turbo Debugger UI.

Using the Main Menus

As with many Borland products, Turbo Debugger has a convenient system of menus accessible from a menu bar running along the top of the screen. The main menu bar is always available, no matter which window is "active" (that is, which window has a cursor in it). There are pull-down menus available for each item on the menu bar.

There are three ways to go to the menus on the main menu bar:

- Press *F10* and then cursor to the desired menu and press *Enter.*
- Press *F10* and then press the first letter of the menu name (*F, V, R, B, D, W, O).*
- Press *Alt* plus the first letter of any main menu command (*F, V, R, B, D, W, O*) to activate the specified command menu. For example, anywhere in the system, *Alt-F* takes you to the File menu.

You press *Esc* to leave the menu bar without choosing a command.

To move around inside a menu off the main menu bar:

- Press *Esc* to exit a menu. As long as you aren't in a second-level menu, you'll return to the previously active window.
- Press *F10* from within any menu level to return to the previously active window.

- Use the *Right* and *Left arrow* keys to move from one pull-down menu to another.
- Use the *Home* and *End* keys to go to the first and last menu items, respectively.

Some commands in the main menus have shortcut key commands, also known as *hot keys*. Where applicable, the appropriate hot key appears to the right of the menu command.

Figures 12.1, 12.2, and 12.3 in Chapter 12 show the complete pull-down menu tree for Turbo Debugger. Table 12.1 on page 184 lists all the hot keys. For a summary of all the commands available in Turbo Debugger, refer to Chapter 12.

Knowing Where It's At

In addition to the convenient system of Borland pull-down menus, the Turbo Debugger advantage consists of a powerful feature that lessens confusion and reduces the learning curve by actually reducing the number of menus.

To understand this feature, you need to realize that first and foremost, Turbo Debugger is a context-sensitive program. Turbo Debugger keeps tabs on exactly what window you have open, what text is selected, and which part of the window your cursor is in (that is, which "pane"). 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 to your command. Let's take an example to illustrate this fundamental point.

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* (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 plus the index and then press *Ctrl-I*, the debugger would know that you wanted to inspect one member and would show you only the member.

You can "tunnel" down to finer and finer program detail in this way. Pressing *Ctrl-I* while examining 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) 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 sensitivity to context is in its use of pop-up menus specific to the occasion.

Pop-up menus in Turbo Debugger are called "local" to remind you that they are tailored to the particular spot your cursor happens to be. It's important not to confuse pop-up (local) menus with pull-down (global) menus (which were discussed on page 17). Compare the following two lists:

Pull-Down (Global) Menus

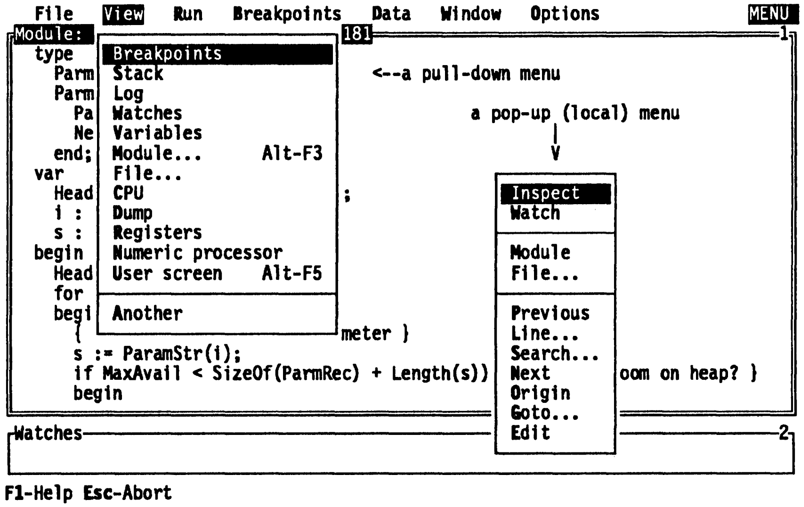
- Pull-down menus are those that you access by pressing *F10* and using the arrow keys or typing the first letter of the menu name.
- The pull-down menus are always available and visible on the menu bar at the top of the screen.
- Their contents never change.
- Some of the menu commands have hot key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

Pop-Up (Local) Menus

- You call up a pop-up menu by pressing *Alt-F10* or *Ctrl-F10.*
- The placement and contents of the menu depends on what text is selected or where your cursor is.
- The contents of pop-up menus can change. (Even so, it's important to realize that 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 the other.) Even the *results* of like-named commands can be different, depending on the context.
- Every command on a pop-up has a hot key shortcut consisting of pressing *Ctrl* plus the first letter of the command.

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

Here is a composite screen shot of both kinds of menus (when actually working in Turbo Debugger, however, you could never have both types of menus showing at the same time):



```
 File  View  Run  Breakpoints  Data  Window  Options                MENU
Module:                       181═══════════════════════════════════1
 type   Breakpoints
  Parm  Stack                   <--a pull-down menu
  Parm  Log
    Pa  Watches                          a pop-up (local) menu
    Ne  Variables                                  |
  end;  Module...      Alt-F3                      V
 var    File...
  Head  CPU                     ;           Inspect
  i :   Dump                                Watch
  s :   Registers
 begin  Numeric processor                   Module
  Head  User screen    Alt-F5               File...
  for
  begi  Another                             Previous
   {                            meter }     Line...
    s := ParamStr(i);                       Search...
    if MaxAvail < SizeOf(ParmRec) + Length(s))  Next     oom on heap? }
    begin                                   Origin
                                            Goto...
Watches                                     Edit                    2

F1-Help Esc-Abort
```

Figure 2.1: Pull-Down vs. Pop-Up Menus

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.

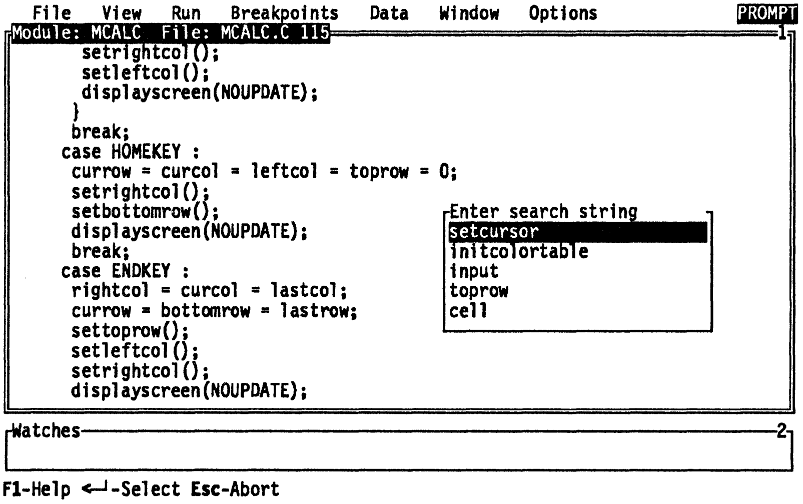
History Lessons

Menus and context-sensitivity comprise just two aspects of the convenient user interface 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 prompt boxes and displays that text whenever you call up the box again.

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

The search prompt box might look like this:


File View Run Breakpoints Data Window Options
PROMPT
Module: MCALC File: MCALC.C 115
1
Enter search string
setcursor
initcolortable
input
toprow
cell
Watches
2
F1-Help ←┘-Select Esc-Abort

```
        setrightcol();
        setleftcol();
        displayscreen(NOUPDATE);
       }
       break;
      case HOMEKEY :
       currow = curcol = leftcol = toprow = 0;
       setrightcol();
       setbottomrow();
       displayscreen(NOUPDATE);
       break;
      case ENDKEY :
       rightcol = curcol = lastcol;
       currow = bottomrow = lastrow;
       settoprow();
       setleftcol();
       setrightcol();
       displayscreen(NOUPDATE);
```

Figure 2.2: A History List in a Prompt Box

You can use this history list as a shortcut to typing by using the arrow keys to select any previous text and then press *Enter* to start the search. If you use an unaltered entry from the history list, the entry moves to the top of the list. You can also edit text (use the arrow keys to insert the cursor into the highlighted text then edit as normal using *Del* or *Backspace*). For example, you can select *myPercentage* and change it to *hisPercentage* rather than typing in the entire text. If you start to type a new item when an entry is highlighted, you will overwrite the highlighted item. The first item in a search list is always the word the cursor is on in the Module window.

The debugger lists the last ten responses unless you tell it otherwise. (You can change its size using the TDINST program.)

Turbo Debugger keeps a separate history list for most prompt boxes. That way, the text you enter for searching for text does not clutter up the box for, say, going to a particular label or line number.

Making Macros

Macros are simply keystroke shortcuts that you define. You can define any series of Turbo Debugger commands and keystrokes to a single key, for "playback" whenever you want.

To create a macro, press *F10* to activate the menu bar, press *O* to select the **O**ptions menu, and choose **M**acros from the **O**ptions menu. At this point, you have a choice of four commands: **C**reate, **R**emove, **D**elete All, and **S**top Recording. Choose **C**reate; Turbo Debugger prompts you for a key to save the upcoming macro to. Press a little-used or easily remembered key (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 recording session, press the newly defined macro key again* (*Shift-F1*, for example), or press *Alt—*.

Whenever you find yourself repeating a series of steps, say to yourself, "Couldn't I be using a macro for this?" For example, do you find yourself resizing and moving windows a lot depending on whether a program has many comments extending to the right margin? If so, create a macro that makes the Module window half-wide (save it as *Shift-F2*, for half) and make another macro that makes it full size (save it as *Shift-F3*, for full). Now you can toggle easily from one to the other as needed.

Window Shopping

Lots of programs do windows nowadays, but Turbo Debugger does them better. Turbo Debugger displays all information and data in menus (local and global), prompt boxes (which you type into), and windows. There are many types of windows depending on what sort of information that's in it. 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 pull-down **V**iew menu (there are 12 types of windows found there). There is another class of window called the Inspect window, which is opened by choosing **D**ata/**I**nspect or by choosing **I**nspect from many of the local pop-up menus.

Windows from the View Menu

Here is a list of the 12 types of windows you can open by choosing commands from the **V**iew menu. You close these windows by pressing *F3* or by choosing **W**indow/**C**lose. If you unintentionally close a window, choose **W**indow/**U**ndo Close to reopen it (with its contents exactly as they were when you closed it). You can recover only the last-closed window in this way.

Module window Displays the program code that you're debugging. You can move around inside the module and examine data and code by "pointing" at program variable names with the cursor and issuing the appropriate local menu command.

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 *Alt-F3* to open the Module window. (Chapter 8 details the Module window and its commands.)

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

Breakpoints window Displays the breakpoints you have set. A breakpoint defines a location in your program where something is meant to happen, such as your program stopping so you can examine the state of the world. (Turbo Debugger's breakpoints encompass all the functionality of what are usually referred to as breakpoints, watchpoints, and tracepoints.)

You can use this window to modify, delete, or add breakpoints. (See Chapter 7 for a complete description of this type of window and how breakpoints work.)

Stack window Displays the current state of the stack. You can get further information on any function or procedure name in the stack by cursoring to it and "inspecting" it by pressing *Ctrl-I.*

By placing the cursor at one of the functions or procedures in the list, you can examine either its local variables or the type of the arguments it was called with. (Chapter 5 provides more information on the Stack window.)

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. The log contains such things as why your program stopped, the results of breakpoints, and the values of structures 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. This can be really handy after a frenzied run of looking here, looking there, until finally you say, "How the dickens did I get here?" (Chapter 7 tells you more about the Log window.)

Variables window Displays all the variables accessible from that spot in your program. The left pane has global variables; the right pane shows local variables, if any.

This window can be helpful when you want to find a function or variable that you know begins with, say, "abc," but when you can't remember its exact name. You can look in the global symbol pane and quickly find what you want. (Chapter 5 describes the Variables window in more detail.)

File window 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 bytes sequences, as well as directly patch any part of the file on disk. Press *F2* to open a File window.

This is handy if you are debugging a program that uses disk files, and you wish 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. (You can learn more about the File window in Chapter 8.)

CPU window Displays the current state of the central processing unit. This window has five panes: one that contains disassembled machine instructions, one that shows hex data bytes, another displays a raw stack of hex words, another 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 some types of subtle bugs. You do not need to use this window to debug the majority of programs.

(Chapter 10 discusses the CPU window and assembler-level debugging.)

Dump window Displays a raw display of an area of memory. You can view the data as characters, hex bytes, words, double words, and all the floating-point formats. (This window is the same as the Data pane of a CPU window.) You can use it when you want to look at some raw data but don't care about 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.

See Chapter 10, which discusses assembler debugging, for more information on how to use this window.

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

Chapter 10, which discusses assembler debugging, has more information on how to use this window.

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. (See Chapter 11 for more information about using the Numeric Processor window.)

User Screen window Shows you your program's 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.

You can use this window to check that your program is at the place in your code you expect it to be, as well as to verify that it is displaying what you want on the screen. *Alt-F5* is the shortcut. After viewing the User Screen window, press any key to go to the debugger screen.

You can also open duplicates of three types of windows—CPU, File, and Module—by choosing **View/Another**. This allows you to keep track of several disparate areas of assembly code, different files the program uses or generates, or several distinct programs 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. (For example, if you choose to view your source file while you're in a Code pane, Turbo Debugger automatically opens a CPU or Module window.)

Inspector Windows

An Inspector window displays the current value of a selected variable. This type of window is never split into panes. Usually, you close this window by pressing *Esc*. 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 inspectors in one swoop by pressing *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.

An Inspector window adapts to the type of data being displayed. It can display simple scalars (**int**, **float**, and so on), as well as 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.

Note that unlike windows from the **View** menu, you can create additional Inspector windows simply by choosing the **Inspect** command again. (You can create additional Module, File, or CPU windows only by choosing **View/Another**.)

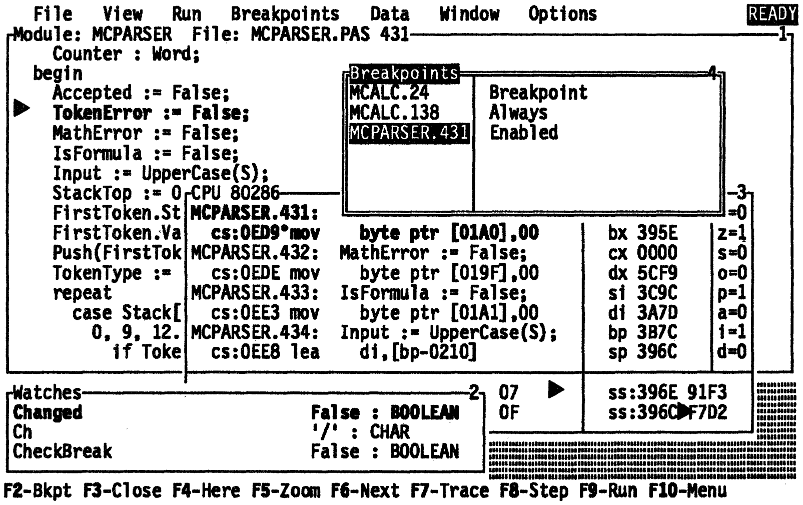
The Active Window

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 line around it, not a single line.
- The active window's title is highlighted.

- The active window is the one with the cursor or highlight bar in it.
- 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 window that might be showing or open.



```
  File   View   Run   Breakpoints   Data   Window   Options                READY
 Module: MCPARSER  File: MCPARSER.PAS 431-------------------------------------1
    Counter : Word;
  begin                              Breakpoints-------------------------4
    Accepted := False;               MCALC.24       Breakpoint
 ►  TokenError := False;             MCALC.138      Always
    MathError := False;              MCPARSER.431   Enabled
    IsFormula := False;
    Input := UpperCase(S);
    StackTop := 0 CPU 80286---------                                      -3
    FirstToken.St MCPARSER.431:                                            =0
    FirstToken.Va   cs:0ED9•mov    byte ptr [01A0],00       bx 395E   z=1
    Push(FirstTok MCPARSER.432:  MathError := False;        cx 0000   s=0
    TokenType :=    cs:0EDE mov    byte ptr [019F],00       dx 5CF9   o=0
    repeat        MCPARSER.433:  IsFormula := False;        si 3C9C   p=1
      case Stack[   cs:0EE3 mov    byte ptr [01A1],00       di 3A7D   a=0
        0, 9, 12. MCPARSER.434:  Input := UpperCase(S);     bp 3B7C   i=1
           if Toke  cs:0EE8 lea    di,[bp-0210]             sp 396C   d=0
 Watches-------------------------------------2 07   ►   ss:396E 91F3
 Changed                       False : BOOLEAN  0F       ss:396C►F7D2
 Ch                            '/' : CHAR
 CheckBreak                    False : BOOLEAN
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 2.3: Can You Spot the Active Window?

Window Hopping

You can make any of the first nine open windows the active one by pressing *Alt* in combination with the number of the window, which appears in the upper right corner of the window. (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.) If you press *Alt-2,* for example, to make the Watches window active, any commands you choose will affect that window and any variables that might be in it.

To see a list of all open windows, press *Alt-0.* Turbo Debugger displays a normal pop-up menu of the open windows for you to select one. Press *Enter* and the selected windows becomes the active one.

You can also press *F6* to cycle through the windows in turn. This is handy if an open window's number is covered so you don't know what shortcut key to press to make it active.

If a window has *panes*—areas of the window designated for distinct types of data—you can move from one pane to another by pressing *Tab* or *Shift-Tab*, or by choosing Window/Next Pane Cycle. 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 sometimes a blinking cursor appears in the pane while other times a highlight bar appears instead. 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 shortcuts for moving around in the pane. Refer to Table 12.1 and Table 12.1 in Chapter 12 for lists of keystroke commands in panes.

If there's a highlight bar in a pane instead of a cursor, you can still use standard keypad 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 searching 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 local command relevant to it. In many lists, you can also just press *Enter* once you have selected an item. This acts as a shortcut 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 187.

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 command in a pane or window—like Goto in a Module window, Search in a File window, or Change in a Registers window.

Resizing and Saving Windows

When Turbo Debugger makes a new window, it appears near the current cursor location and has a default size suitable for the specified window. You can use the Window/Move/Resize command to adjust the size or location of the window. After choosing this command, your active window border changes to a single-line border. You then use the arrow keys to change the placement and size of the window on your screen. Press *Enter* when you're satisfied with its position. Pressing *Scroll Lock* is a shortcut for the **Move/Resize** command.

If you want to quickly enlarge (or then reduce) a window, you can press *F5* to "zoom" or "unzoom" it.

You can also use the **Options/Save Options** command to save a specific window configuration once you have the screen arranged the way you like. The screen will then appear that way each time you start Turbo Debugger from DOS, if the configuration was saved to a file called TDCONFIG.TD. This is the only configuration file that is loaded automatically when Turbo Debugger is loaded. Other configurations (saved in other files) *can* be loaded by using the **Options/Restore Options** command, if this configuration was saved to TDCONFIG.TD.

Getting Help

As you've seen, Turbo Debugger goes out of its way to make debugging easy for you. It requires a minimum of remembering obscure commands, it keeps lists of what you do type in case you want to repeat them, 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:

- A highlighted activity indicator in the upper right corner always displays the current activity. For example, if your cursor is in a window, the activity indicator read READY; if there's a menu visible, it reads MENU; when at a prompt box, it reads PROMPT. If you ever get confused about what's happening in Turbo Debugger, look at the activity indicator for help. (Other possibilities for the activity indicator are MOVE/RESIZE, MOVE, and ERROR.)
- Remember that the active window is always topmost and has a double line around it.
- You can access an extensive context-sensitive help system by pressing *F1*.
- The bottom line of the screen always offers a quick reference summary of keystroke commands. The line changes as the context changes and as you press *Shift*, *Alt*, or *Ctrl*.

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

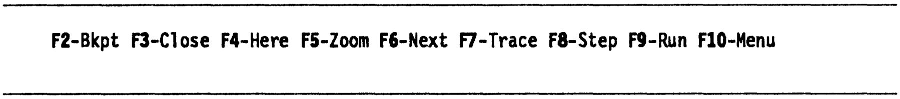
Press *F1* to bring up a Help window showing information pertinent to the current context. Some help screens contain highlighted keywords that allow you to get additional help on that highlighted item. Use the arrow keys to move to any keyword and then press *Enter* to get to that item's screen. You can use the *Home* and *End* keys to go to the first and last keywords on the screen, respectively.

If you want to return to a previous Help screen, press *Alt-F1*. 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 context-related screens. To access the Help Index, Press *F1* from within the Help system. To exit from Help, press *Esc*.

The Bottom Line

Wherever you're in Turbo Debugger, a quick reference help line appears at the bottom of the screen. This line provides at-a-glance keystroke command help for your current context.

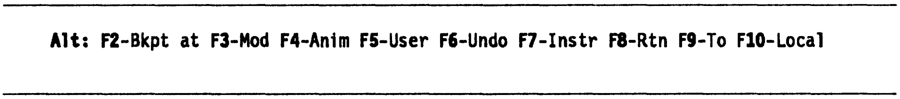
The normal bottom (reference) line shows the commands performed by the function keys, and looks like this:



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

Figure 2.4: The Normal Reference Line

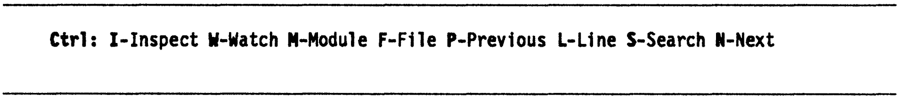
If you hold down the *Alt* key for a second or two, the commands performed by the *Alt* function keys are displayed.



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

Figure 2.5: The Reference Line with *Alt* Pressed

If you hold down the *Ctrl* key for a second or two, the commands performed by the *Ctrl* letter keys are displayed. This help 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 bottom 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.



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

Figure 2.6: Typical Reference Line with *Ctrl* Pressed

CHAPTER

3

Getting Started: 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 user interface and context-sensitive help system allow you to learn as you go along.

This chapter leads you through all of Turbo Debugger's basic features. After describing the sample programs—one in C and another in Pascal—provided on the distribution disk, it shows you how to

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

The Sample Programs

The sample programs (TCDEMO.C for C, 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. (For example, each shows you how to access command-line parameters, even though they're not really used for anything.)

Each demo program lets you type in some lines of text, then counts the number of words and letters that you entered. At the end of the program, 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 the tutorial. For the C example, you'll need TCDEMO.C and TCDEMO.EXE; for the Pascal example, you need TPDEMO.PAS and TPDEMO.EXE.

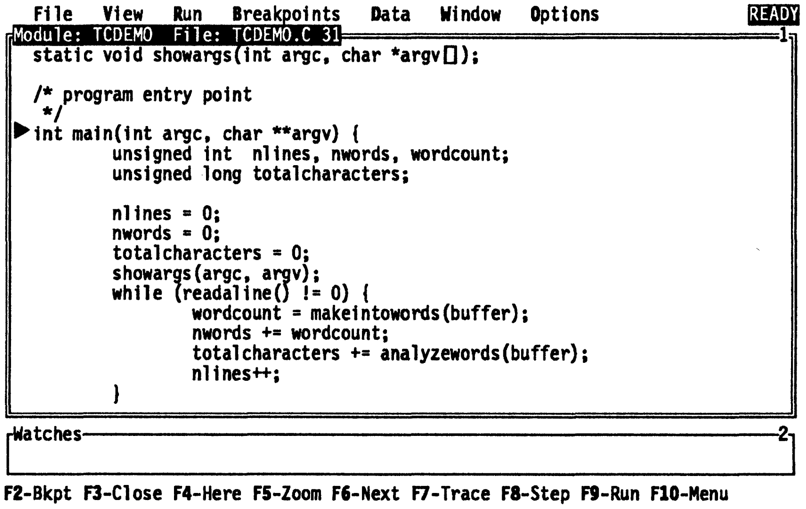
To start the C program, enter

```
TD TCDEMO
```

To start the Pascal program, enter

```
TD TPDEMO
```

Turbo Debugger loads the demo program and positions the cursor at the start of the program.



```
 File   View   Run   Breakpoints   Data   Window   Options                 READY
┌Module: TCDEMO  File: TCDEMO.C 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);
                  nwords += wordcount;
                  totalcharacters += analyzewords(buffer);
                  nlines++;
          }
┌Watches─────────────────────────────────────────────────────────────────────2┐

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

Figure 3.1: The Startup Screen Showing TCDEMO

This screen consists of the main menu line, the Module and Watches windows, and the help line.

To exit from the tutorial at any time and return to DOS, press *Alt-X*. If you get hopelessly lost while 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 L* to do that.

Press *F1* whenever you need help about the current window, command, prompt, 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 top line of the screen shows the main menu bar. To pull down a menu off of it, press either the *F10* key or the *Alt* key in combination with the first letter of one of the menu names.

Press *F10* now. Notice that the cursor disappears from the Module window, and the File command on the main menu becomes highlighted. The bottom line of the screen also changes to show you which keys you can use in the File menu; in this case, only the *F1* (Help) and *Esc* (Abort) keys appear.

Use the arrow keys to move around the menu system. Press *Down arrow* to pull down the menu for the highlighted item on the main menu.

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

The Help Line

The bottom line of the screen shows relevant function keys and what they do. This line further changes depending on what you are entering (menu commands, a response to a prompt, and so on). Hold *Alt* down for a second or two, for example. Notice that the bottom line changes to show you the function keys you can use with *Alt*.

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

The Windows

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

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. Press *F6* to switch to another window. Do that now. The Watches window becomes active, with a double-line border and a highlighted title.

Use the **View** menu command from the main menu bar to create new windows. Press *Alt-V* to pull down the **View** menu, then press *S* to open a Stack window. The View menu disappears, and the Stack window pops up on top of the Module window.

To remove the active window, press *F3.* Do that now. 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, press *Alt-W* to open the **Window** menu. Press *U* to choose the **Undo Close** command. The Stack window reappears. You can also press *Alt-F6* to recover the last-closed window.

The **Window** menu contains the commands that let you adjust the appearance of the windows you already have on the screen. You can both move the window around the screen and change its size. (You can use *Scroll Lock* to do this too.)

Press *Alt-W M* (to display the **Window** menu and then choose the **Move/ Resize** command) and use the arrow keys to reposition the active window (the Stack window) on the screen. 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.

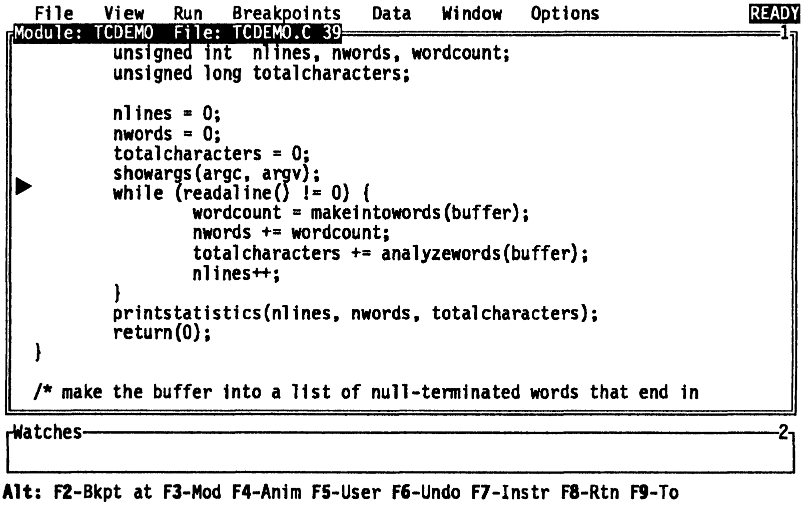
Now, to prepare for the next section, remove the Stack window by pressing *F3.* Depending on whether you've loaded the C or Pascal demo program, you should continue with the next section (for the C sample) or move to the Pascal section starting on page 43.

Using the C 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.

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 38. First, position the cursor on line 38, then press *F4*. This will run the program up to (but not including) line 38. Now press *F7*, which executes one line of source code at a time; in this case, it executes line 38, a call to the function **showargs**. The cursor will immediately jump to line 150, where the definition of **showargs** is found. Continuing to press *F7* will step you through the function **showargs** and then return you to the line following the call—line 39. If you had pressed *F8* instead of *F7* on line 38, the cursor would have gone directly to line 39 instead of to the function. *F8* is similar to *F7* in that it executes functions, but it doesn't enter them.


File View Run Breakpoints Data Window Options
READY
Module: TCDEMO File: TCDEMO.C 39
1
Watches
2
Alt: F2-Bkpt at F3-Mod F4-Anim F5-User F6-Undo F7-Instr F8-Rtn F9-To

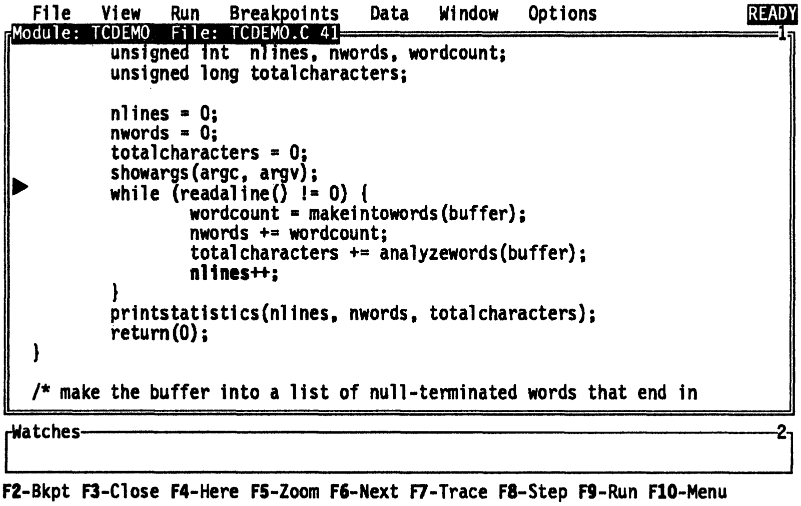
```
        unsigned int nlines, nwords, wordcount;
        unsigned long totalcharacters;

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

/* make the buffer into a list of null-terminated words that end in
```

Figure 3.2: The Program Stops after Returning from Function showargs

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 prompt box appears. Type `readaline` and press *Enter*. The program runs, then stops at the beginning of function **readaline** (line 141).

Setting Breakpoints in the C Demo Program

Another way to control where your program stops running is with breakpoints. The simplest way to set a breakpoint is with the *F2* key. Move the cursor to line 43 and press *F2*. Turbo Debugger highlights the line, indicating there is a breakpoint set on it.


File View Run Breakpoints Data Window Options
READY
Module: TCDEMO File: TCDEMO.C 41
1
Watches
2
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
        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++;
        }
        printstatistics(nlines, nwords, totalcharacters);
        return(0);
}

/* make the buffer into a list of null-terminated words that end in
```

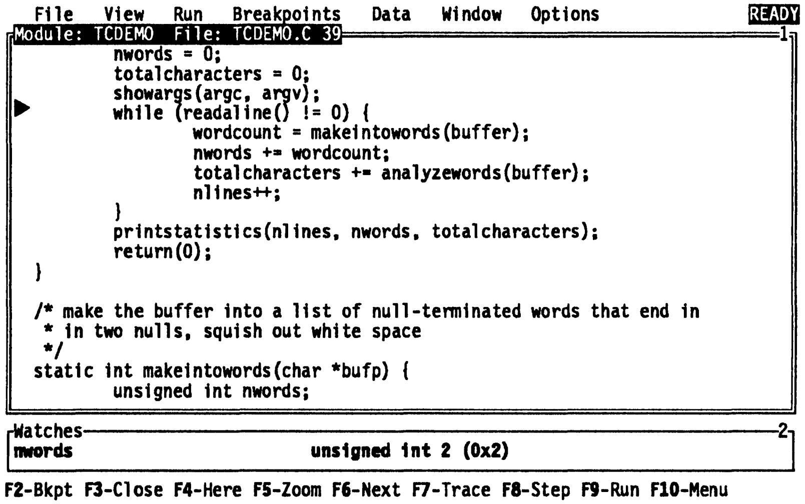
Figure 3.3: A Breakpoint at Line 43

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 43, where you set a breakpoint that has stopped the program.

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

Using Watches

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 41 and press *Ctrl-W*. This is the shortcut for the local menu command *Alt-F10 W*.


File View Run Breakpoints Data Window Options
READY
Module: TCDEMO File: TCDEMO.C 39
1
Watches
2
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
        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;
```

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

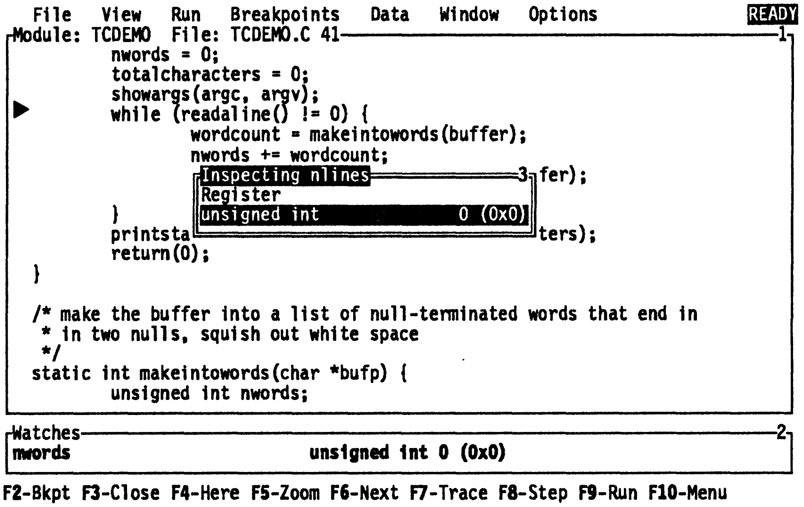
Figure 3.4: A Variable in the Watches Window

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 writing 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 press *Ctrl-I*. An Inspector window pops up.



```
  File   View   Run   Breakpoints   Data   Window   Options                    READY
┌Module: TCDEMO  File: TCDEMO.C 41─────────────────────────────────────────────1┐
│        nwords = 0;
│        totalcharacters = 0;
│        showargs(argc, argv);
│►       while (readaline() != 0) {
│                wordcount = makeintowords(buffer);
│                nwords += wordcount;
│               ╔Inspecting nlines═══════════3╗fer);
│               ║Register                     ║
│        }      ║unsigned int          0 (0x0)║
│        printsta╚════════════════════════════╝ters);
│        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 0 (0x0)
└──────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 3.5: An Inspector Window

The first line tells you that the compiler has optimized this variable by placing it in a CPU register. You need not worry about where the variable is actually stored—Turbo Debugger takes care of everything, letting you refer to all your data exactly as shown in your program. The second 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.

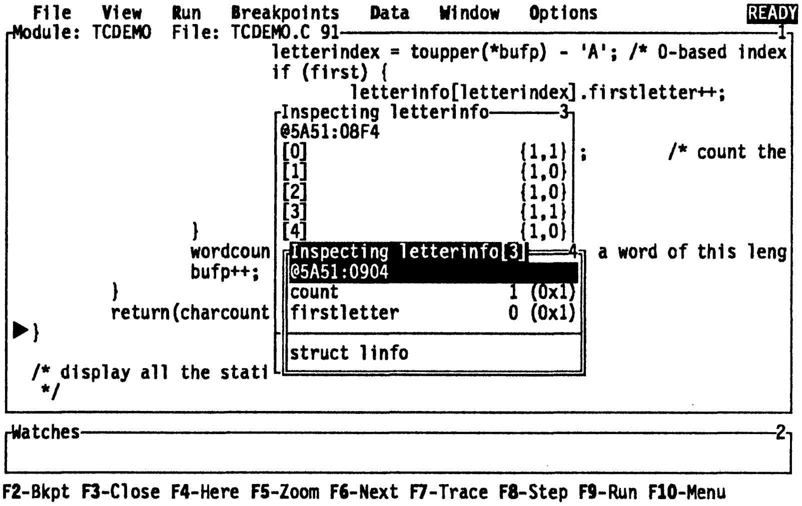
Now, having examined the variable, press *Esc* to close the Inspector window. You can also use *F3* to remove the inspector, just like any other window.

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, press *Alt-D I.* A prompt box appears, asking you to enter the variable to inspect. Type `letterinfo` and press *Enter.* An inspector appears, showing the values of the `letterinfo` array elements. The title of the inspector 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 the Inspect command. A new inspector appears, showing the contents of that element in the array. This inspector shows the contents of a structure of type `linfo`.



```
  File   View   Run   Breakpoints   Data   Window   Options                READY
┌Module: TCDEMO  File: TCDEMO.C 91──────────────────────────────────────────────1┐
│                          letterindex = toupper(*bufp) - 'A'; /* 0-based index│
│                          if (first) {                                        │
│                                  letterinfo[letterindex].firstletter++;      │
│                          ┌Inspecting letterinfo──────3┐                      │
│                          │@5A51:08F4                  │                      │
│                          │[0]                    {1,1}│;        /* count the│
│                          │[1]                    {1,0}│                      │
│                          │[2]                    {1,0}│                      │
│                          │[3]                    {1,1}│                      │
│                 }        │[4]                    {1,0}│                      │
│                 wordcoun │╔Inspecting letterinfo[3]══4╗ a word of this leng│
│                 bufp++;  │║@5A51:0904                 ║                     │
│         }                │║count              1 (0x1) ║                     │
│         return(charcount │║firstletter        0 (0x1) ║                     │
│▶}                        ├║───────────────────────────║                     │
│                          │║struct linfo               ║                     │
│  /* display all the stati└╚═══════════════════════════╝                     │
│   */                                                                        │
└───────────────────────────────────────────────────────────────────────────────┘
┌Watches────────────────────────────────────────────────────────────────────────2┐
│                                                                               │
└───────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 3.6: Inspecting a Structure

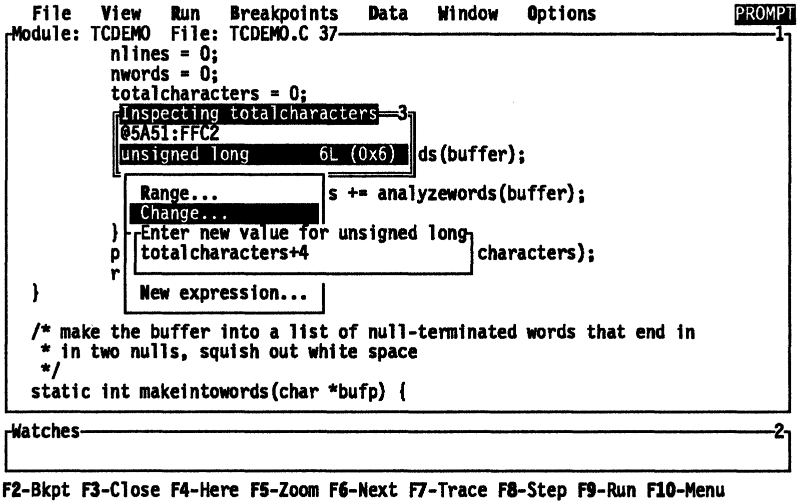
When you place the cursor over one of the member names, the data type of that member appears in the bottom pane of the inspector. If one of these members were in turn a compound data object, you could issue an inspect command and dig down further into the data structure.

Press *F3* to remove both Inspector windows and return to the Module window. (*F3* is a convenient way of removing several inspectors at once. If you had pressed *Esc*, only the latest inspector 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 37 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. Press *C* to choose the Change option. (You could also have done this directly by pressing *Ctrl-C*.) A prompt appears, asking for the new value.



```
  File   View   Run   Breakpoints   Data   Window   Options                PROMPT
┌Module: TCDEMO  File: TCDEMO.C 37──────────────────────────────────────────1┐
│          nlines = 0;
│          nwords = 0;
│          totalcharacters = 0;
│         ┌Inspecting totalcharacters═3┐
│         │@5A51:FFC2                  │
│         │unsigned long     6L (0x6)  │ds(buffer);
│         └─┌──────────────────┐───────┘
│           │ Range...         │s += analyzewords(buffer);
│           │ Change...        │
│         }┌Enter new value for unsigned long┐
│         p│totalcharacters+4                │characters);
│         r└─────────────────────────────────┘
│  }        │ New expression...│
│           └──────────────────┘
│  /* 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) {
└────────────────────────────────────────────────────────────────────────────┘
┌Watches────────────────────────────────────────────────────────────────────2┐
│
└────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 3.1: The Change Command Prompt

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 now shows the new value, 10L (0xA).

To change a data item that isn't displayed in the Module window, press *Alt-D E*. A prompt box appears, in which you enter the name of the variable to change. Type `argc` and press *Enter*, then press *Tab* twice to move to the line labeled New Value. Type `123` and press *Enter*. The integer result (second line) will change to `int 123 (0x7B)`.

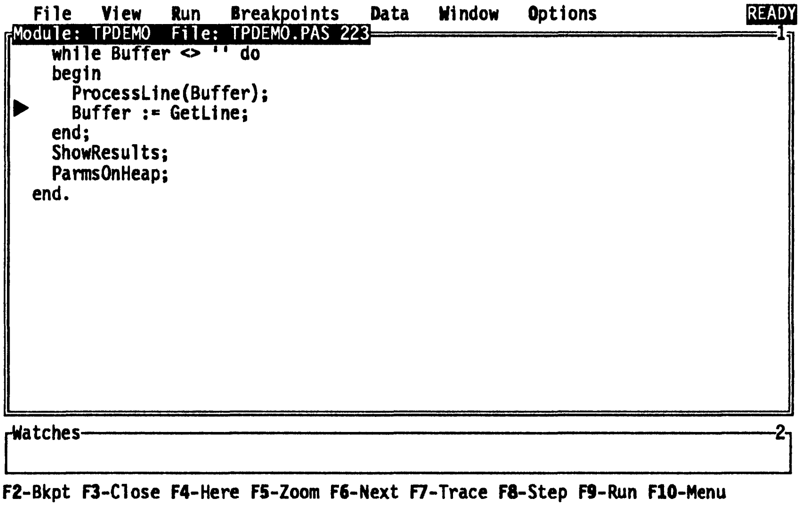
That's a quick introduction to using the Turbo Debugger with a Turbo C program. For a more extensive walk-through, take a look at Chapter 13's sample debugging session—it uses a "buggy" version of this program.

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 line.

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

To make the program execute until it reaches line 222, move the cursor to that line and then press *F4*. TPDEMO will prompt you to enter a string. Type a few keystrokes, then press *Enter*. Now, with the cursor still on line 222, press *F7* to execute another single line of source code. Since the line you executed is a call to a different procedure, the arrow now appears on the first line of the function *ProcessLine*. Press *Alt-F8* to make the program stop when *ProcessLine* returns. This command is very useful when you want to jump past the end of a function or procedure.


File View Run Breakpoints Data Window Options
READY
Module: TPDEMO File: TPDEMO.PAS 223
1
Watches
2
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

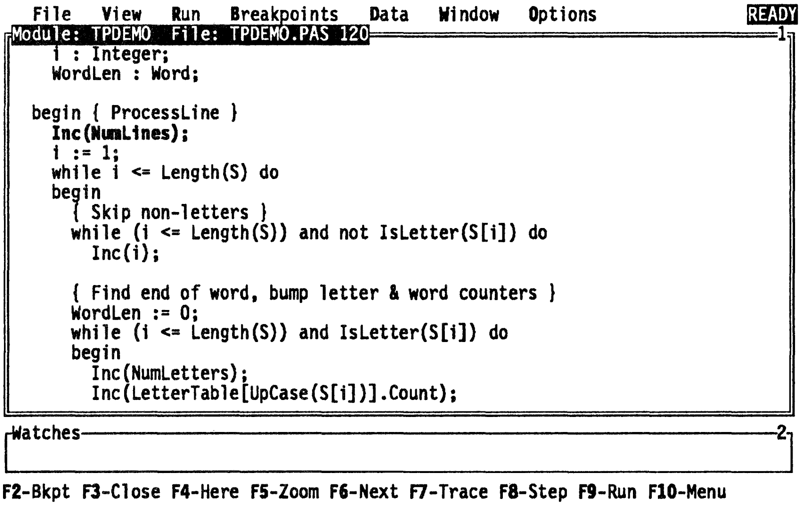
```
   while Buffer <> '' do
   begin
     ProcessLine(Buffer);
►    Buffer := GetLine;
   end;
   ShowResults;
   ParmsOnHeap;
 end.
```

Figure 3.8: The Program Stops after Returning from a Procedure

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 prompt box appears. Type `GetLine` and press *Enter.* The program runs, then stops at the beginning of function *GetLine.*

Setting Breakpoints in the Sample Pascal Program

Another way to control where your program stops running is with breakpoints. The simplest way to set a breakpoint is with the *F2* key. Move the cursor to line 120 and press *F2.* Turbo Debugger highlights the line, indicating there is a breakpoint set on it.



```
 File   View   Run   Breakpoints   Data   Window   Options                 READY
Module: TPDEMO  File: TPDEMO.PAS 120                                          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

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

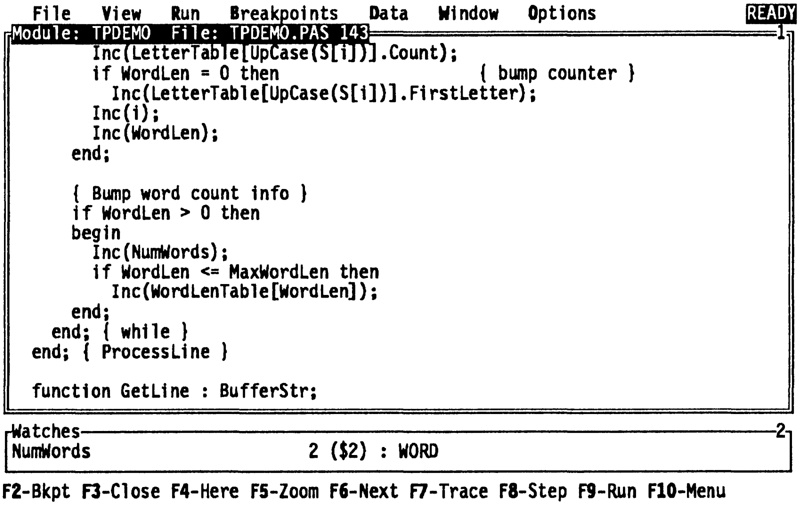
Figure 3.9: A Breakpoint at Line 120

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 120, where you set a breakpoint that has stopped the program.

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

Using Watches

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 143 and press *Ctrl-W.* This is the shortcut for the local menu command *Alt-F10 W.*


File View Run Breakpoints Data Window Options
READY
Module: TPDEMO File: TPDEMO.PAS 143
1
Watches
2
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

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

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

function GetLine : BufferStr;
```

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

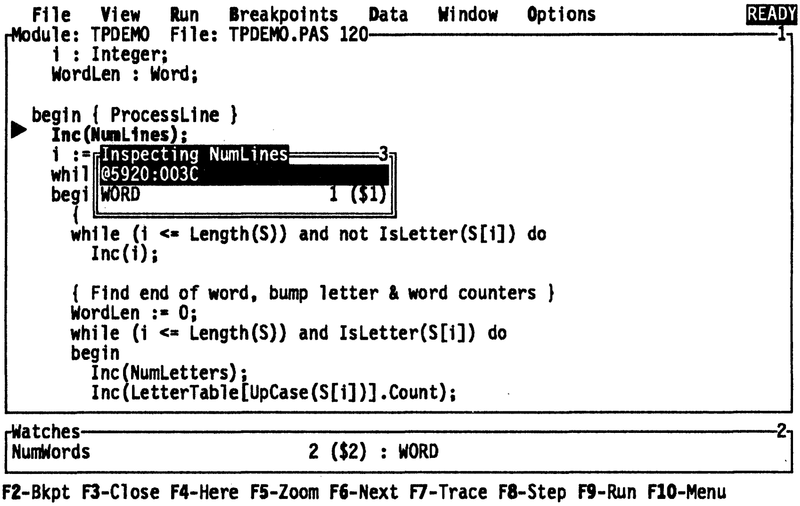
Figure 3.10: A Pascal Variable in the Watches Window

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 Inspect command. This very powerful facility lets you examine data structures in the same way that you visualize them when writing a program.

The Inspect commands (in various local menus and in the Data menu) let you examine any variable you specify. Suppose you want to look at the value of the variable *NumLines.* Move the cursor back to line 120 so it's under one of the letters in *NumLines* and press *Ctrl-I.* An Inspector window pops up.



```
  File   View   Run   Breakpoints   Data   Window   Options                READY
┌Module: TPDEMO  File: TPDEMO.PAS 120──────────────────────────────────────────1┐
│   i : Integer;
│   WordLen : Word;
│
│ begin { ProcessLine }
▶   Inc(NumLines);
│   i :=┌Inspecting NumLines══════3┐
│   whil│@5920:003C                │
│   begi│WORD               1 ($1) │
│     { └──────────────────────────┘
│     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                                     │
└───────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 3.11: An Inspector Window

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.

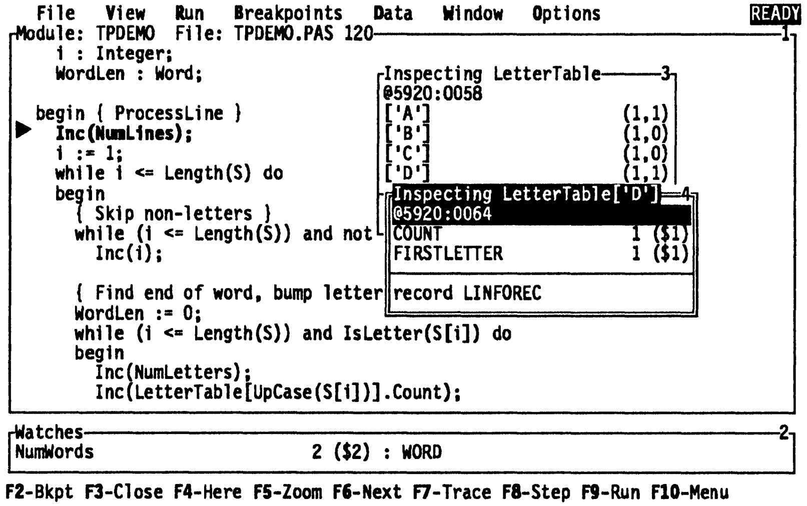
Now, having examined the variable, press *Esc* to close the Inspector window. You can also use *F3* to remove the inspector, just like any other window.

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, press *Alt-D I.* A prompt box appears, asking you to enter the variable to inspect. Type `LetterTable` and press *Enter.* An inspector 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 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 Data Objects in Pascal

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



```
  File   View   Run   Breakpoints   Data   Window   Options                READY
┌Module: TPDEMO  File: TPDEMO.PAS 120──────────────────────────────────────────1┐
│   i : Integer;                                                                │
│   WordLen : Word;                      ┌Inspecting LetterTable─────3┐         │
│                                        │@5920:0058                  │         │
│  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┐        │
│     { Skip non-letters }               │@5920:0064                   │        │
│     while (i <= Length(S)) and not    └│COUNT                 1 ($1) │        │
│       Inc(i);                          │FIRSTLETTER           1 ($1) │        │
│                                        ├─────────────────────────────┤        │
│     { Find end of word, bump letter    │record 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                                  │
└────────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 3.12: Inspecting a Structure

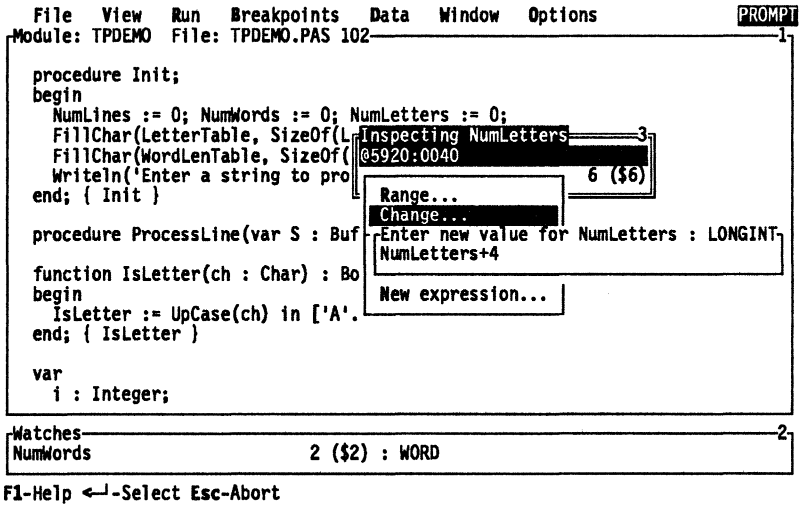
When you place the cursor over one of the member names, the data type of that member appears in the bottom pane of the inspector. If one of these members were in turn a compound data object, you could issue an inspect command and dig down further into the data structure.

Press *F3* to remove both Inspector windows and return to the Module window. (*F3* is a convenient way of removing several inspectors at once. If you had pressed *Esc*, only the topmost inspector 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 102 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's local menu. Press *C* to choose the Change option. (You could also have done this directly by pressing *Ctrl-C*.) A prompt appears, asking for the new value.



```
  File   View   Run   Breakpoints   Data   Window   Options                PROMPT
Module: TPDEMO  File: TPDEMO.PAS 102                                            1

  procedure Init;
  begin
    NumLines := 0; NumWords := 0; NumLetters := 0;
    FillChar(LetterTable, SizeOf(L Inspecting NumLetters       3
    FillChar(WordLenTable, SizeOf( @5920:0040
    Writeln('Enter a string to pro                          6 ($6)
  end; { Init }                    Range...
                                   Change...
  procedure ProcessLine(var S : Buf Enter new value for NumLetters : LONGINT
                                    NumLetters+4
  function IsLetter(ch : Char) : Bo
  begin                            New expression...
    IsLetter := UpCase(ch) in ['A'.
  end; { IsLetter }

  var
    i : Integer;

Watches                                                                         2
NumWords                        2 ($2) : WORD

F1-Help <┘-Select Esc-Abort
```

Figure 3.13: The Change Command Prompt

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, press *Alt-D E*. A prompt box appears, in which you 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 will set the variable *NumLines* equal to *123*.

Well, that wraps up our quick intro to using Turbo Debugger with a Turbo Pascal program. Chapter 13 offers a more extensive debugging sample.

CHAPTER

4

Starting Turbo Debugger

To debug a program with Turbo Debugger, you simply type `TD` and the name of the program, and press *Enter.* Turbo Debugger then loads and runs your program, displaying its source code so you can step through your program statement by statement.

Note: The overlay file, TD.OVL, which contains the menu system, must be available when you load TD. If it is not, an error message will appear, warning you that it cannot be loaded.

Note: If you are running on a two-floppy system, INSTALL has put the overlay file (TD.OVL) and the help file (TDHELP.TDH) on one diskette and TD.EXE on another diskette. To start Turbo Debugger, you insert the disk containing TD.EXE and type `TD` and your program name on the command line. Turbo Debugger will prompt you to insert the diskette containing the overlay file. Once you have inserted the overlay diskette, don't remove it for the remainder of your debugging session.

If you've run your program from the DOS prompt and noticed a bug while using it, you have to exit from your program and load it under the debugger first.

This chapter tells you how to prepare programs for debugging. We show you how to start Turbo Debugger from the DOS command line and 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 should 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. You can generate debug information only for specific modules, but it can be annoying later to enter a module that doesn't have any debug information available. We suggest recompiling all modules—unless you're not using EMS, need memory space, and are sure the code in certain modules works.

Preparing Turbo C Programs

If you're using the Turbo C integrated environment (TC), specify Standalone in the **Debug/Source Debugging** option before you compile your source modules.

If you're using the standalone 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 may also want to 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, be aware that you need 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), you must go to the **Debug** menu and change the Standalone Debugging setting to `On`. Turn **Options/Compiler/Debug** Information `On`. If you want to be able to reference local symbols (any declared within procedures and functions), you must either set the **Options/Compiler/Local** Symbols selection to `On`, or put the {$*L*+} directive at the start of your program. 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

When using Turbo Assembler, specify the **-zi** command-line option to get full debugging information.

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

Preparing Microsoft Programs

See Appendix B of this book for information about how to use the utility program TDCONVRT.EXE, which converts CodeView executable programs to Turbo Debugger format.

Running Turbo Debugger

To run Turbo Debugger, type TD at the DOS prompt, followed by an optional set of command-line arguments, and press *Enter.* Command-line arguments can include the name of the program to debug and debugger options.

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

Note: The overlay file TD.OVL must be available for TD to call upon for its menus and help windows.

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*.

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

Note that when you run Turbo Debugger, you'll need *both* the .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 Options/Path for Source command, third in the current directory, and fourth in the directory the .EXE file is in.

Command-Line Options

All 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 E.

The following describes all available command-line options; Appendix A has an easy-to-use list of these command-line options.

The -c Option

This option loads the specified configuration file. A space cannot exist 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.

The –d Options

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

–do

Runs the 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 only use this option on a color display, since only these displays have multiple display pages. You can't use this option if the program you are debugging uses multiple display pages itself.

–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 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.

The –h and –? Options

Display a screenful of help that describes Turbo Debugger's command-line syntax and options.

The –i Option

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

The –l Option

Forces startup in assembler mode, showing CPU viewer. Doesn't execute compiler startup code.

The –m Option

Sets the working heap to *n* Kbytes, where the syntax is

```
-mn
```

A space cannot exist between the **–m** option and the size of the heap. Here is an example:

```
TD -m64 TCDEMO.EXE
```

The default size is 40K; the high boundary is 64K. Specifying a value larger than 64K may cause unexpected results. If you need memory, use this option to reduce the amount of heap Turbo Debugger uses. Use this option also to increase the amount of heap when you debug small programs. This option lets Turbo Debugger store transient information, such as command history lists.

Note: If you specify a heap size of 0 with the `-m` command-line option (`-m0`), Turbo Debugger will use the maximum that it's able to use, 64K.

The –c Option

The syntax is

```
-cfilename
```

Tells Turbo Debugger to use *filename* as the configuration file.

The –r Options

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.

The –s Options

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

–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 will ignore case only if you've linked your program with the "case ignore" option enabled.

Note: This option has no effect if you're debugging a Turbo Pascal program (because Turbo Pascal is always case-insensitive).

–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.

The –v Options

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 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.

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 default options that will apply when there is no configuration file and to build the configuration file.

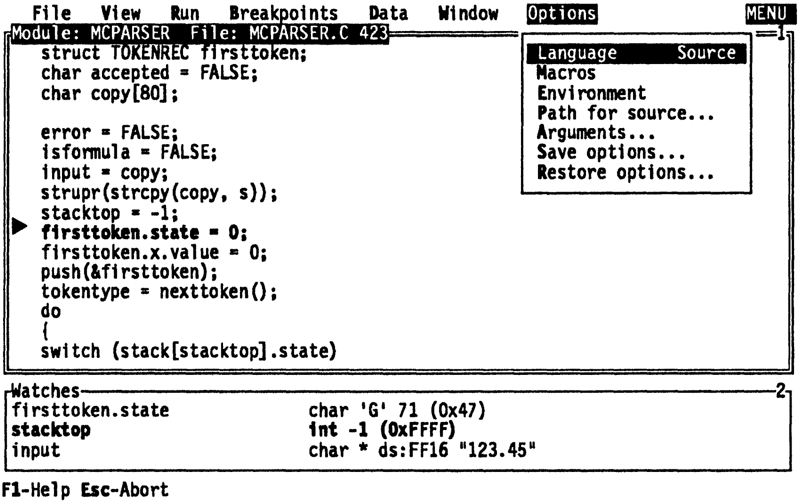
Turbo Debugger looks for the configuration file TDCONFIG.TD first in the current directory, next in the TURBO directory set up with the TDINST installation program, and then in the directory that contains TD.EXE. If you are running on DOS version 2, the debugger won't look for TDCONFIG.TD in the TD.EXE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

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

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

The Options Menu

This 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 where appropriate refer you to other sections of the manual where you can find more details.



```
   struct TOKENREC firsttoken;
   char accepted = FALSE;
   char copy[80];

   error = FALSE;
   isformula = FALSE;
   input = copy;
   strupr(strcpy(copy, s));
   stacktop = -1;
▶  firsttoken.state = 0;
   firsttoken.x.value = 0;
   push(&firsttoken);
   tokentype = nexttoken();
   do
   {
   switch (stack[stacktop].state)
```

```
firsttoken.state              char 'G' 71 (0x47)
stacktop                      int -1 (0xFFFF)
input                         char * ds:FF16 "123.45"
```

Figure 4.1: The Options Menu

Language Command

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

Macros Command

This command displays another menu that lets you define new keystroke macros or delete ones that you have already assigned to a key. It has the following commands:

Create

Starts recording keystrokes that will be assigned to a key (for example, *Alt-M*). You are first prompted for the key to assign the keystrokes to. You then type the keystrokes that you want to record. These keystrokes will be acted upon by Turbo Debugger exactly as if you are not recording a macro. To begin a recording session, select Option/Macro/Create. You will be 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.

Once you have finished recording keystrokes, you must issue the *F10*/ Options/Macros/Stop Recording command or its shortcut, *Alt-hyphen*. You may also press the key you assigned the macro to (*Alt-M*) once more.

While the macro is recording, the mesage `Recording` is displayed in the upper right-hand corner of the screen.

Alt-= is a shortcut for starting to record a macro.

Stop Recording

Stops recording keystrokes that will be assigned to a key. Use this command after issuing the *F10*/Options/Macros/Create command to assign keystrokes to a key.

Alt-hyphen is a shortcut for ending a macro.

Remove

Removes a macro assigned to a single key. You will be prompted to press the key whose macro you want to delete.

Delete All

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

Environment Command

This command displays a menu that lets you set several options to control the apperance of the Turbo Debugger display. It has the following options:

Integer Format

This option lets you cycle among 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.

Display Swapping

The Display Swapping option lets you cycle among three ways of controlling how your program's and Turbo Debugger's screens get swapped back and forth. The three setting are

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

Smart Only swap to the user screen when display output may occur. Turbo Debugger will swap the screens any time that you step over a routine, or if you execute and 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 the screen. If you select this option, the screen will flicker every time you step your porgram, since Turbo Debugger's screen will be replaced for a short time with your program's screen.

Screen Size

Use this option to determine whether Turbo Debugger's screen will use the normal 25-line display or the 43 or 50-line display available on EGA and VGA display adapters.

Tab Size

This option lets you set how many columns each tab stop will occupy. 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 will search for your source files. See Chapter 8 where the Module window is discussed for more information on this option.

Arguments Command

Lets you set new command-line arguments for your program. This is discussed more in Chapter 5.

Save Options Command

Saves your current options to a configuration file on disk. This saves:

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

Turbo debugger allows you to save your options in three ways:

All Saves all settings made in the Options menu, including windows and macros

Layout Saves only the windowing layout

Macros Saves only the currently defined macros

Restore Options Command

Restores your options from a disk file. You can have multiple configuration files, containing different macros, window layouts, etc. You must specify an option file that was created by the Save options command.

Running DOS While in Turbo Debugger

When debugging a program, you sometimes need to use another program or utility. Do this via File/OS Shell.

When you start the DOS command processor, the program you are debugging is swapped to disk if necessary. This allows you to perform DOS commands even while debugging a program that takes all of 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.

When you have finished issuing commands to DOS, type `EXIT` 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*. 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.

CHAPTER

5

Controlling Program Execution

When you debug a program, you usually execute portions of your program and check that it behaves correctly at a stopping point. 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

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, 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 the debugger is in control. We teach you various ways to execute portions of your program and also show you how to interrupt your program while it's running. Finally, we list the ways you can restart a debugging session, both with the same program and with a different program.

Examining the Current Program State

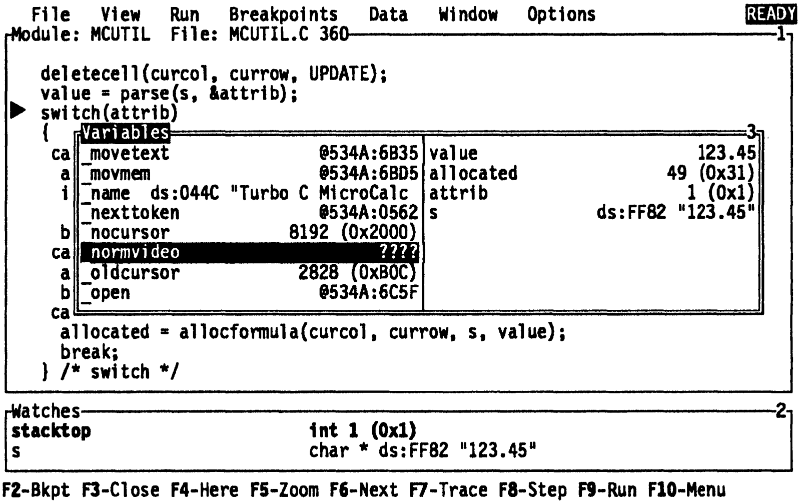
The "state" of your program consists of the following elements:

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

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

The Variables Window

This window shows you all the variables (names and values) that are accessible from the current location in your program. Use this to find variables whose names you can't remember how to spell. 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.


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

```
Module: MCUTIL  File: MCUTIL.C 360                                        1

   deletecell(curcol, currow, UPDATE);
   value = parse(s, &attrib);
 ▶ switch(attrib)
   {   Variables                                                        3
    ca _movetext                  @534A:6B35 value                  123.45
     a _movmem                    @534A:6BD5 allocated            49 (0x31)
     i _name  ds:044C "Turbo C MicroCalc     attrib                1 (0x1)
       _nexttoken                 @534A:0562 s               ds:FF82 "123.45"
     b _nocursor              8192 (0x2000)
    ca _normvideo                      ????
     a _oldcursor              2828 (0xB0C)
     b _open                      @534A:6C5F
    ca
     allocated = allocformula(curcol, currow, s, value);
     break;
   } /* switch */

Watches                                                                   2
stacktop                   int 1 (0x1)
s                          char * ds:FF82 "123.45"
```

Figure 5.1: The Variables Window

Note: When debugging a Turbo Pascal program, the right pane will display the symbols of the current code segment and IP registers (*CS:IP*), which always point to the current location in the program. Also, the variables won't be arranged alphabetically.

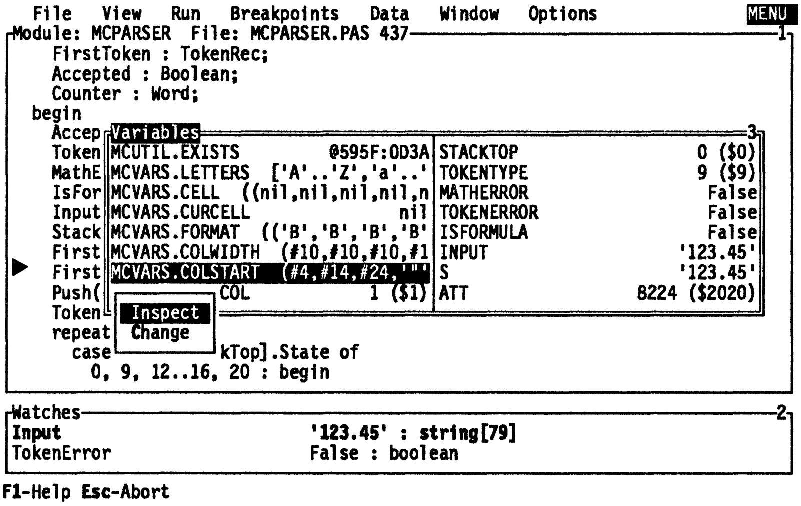
You open a Variables window by choosing **View/Variables** at the main menu; otherwise, from anywhere else you can either press *F10* to get to the main menu or *Alt-V* to get to **View**, then choose **Variables** from the **View** menu. Variables windows have two panes. The *Global* pane (on the left) shows all the global symbols in your program. The *Static* pane (on the right) shows all the static symbols in the current module, which is 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 (*????*) as shown in Figure 5.1.

As with all local menus, press *Alt-F10* to pop up the Global pane's local menu. If Control key shortcuts are enabled, you can press *Ctrl* with the first letter of the desired command to access the command.

If your program contains functions that perform recursive calls, or you wish 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.



```
  File   View   Run   Breakpoints   Data   Window   Options              MENU
 Module: MCPARSER  File: MCPARSER.PAS 437                                   1
    FirstToken : TokenRec;
    Accepted : Boolean;
    Counter : Word;
  begin
    Accep Variables                                                       3
    Token MCUTIL.EXISTS          @595F:0D3A STACKTOP                  0 ($0)
    MathE MCVARS.LETTERS  ['A'..'Z','a'..' TOKENTYPE                 9 ($9)
    IsFor MCVARS.CELL  ((nil,nil,nil,nil,n MATHERROR                   False
    Input MCVARS.CURCELL                nil TOKENERROR                  False
    Stack MCVARS.FORMAT  (('B','B','B','B' ISFORMULA                   False
    First MCVARS.COLWIDTH  (#10,#10,#10,#1 INPUT                    '123.45'
 ▶  First MCVARS.COLSTART  (#4,#14,#24,'' S                        '123.45'
    Push(              COL            1 ($1) ATT                 8224 ($2020)
    Token  Inspect
    repeat Change
      case          kTop].State of
        0, 9, 12..16, 20 : begin

 Watches                                                                    2
 Input                            '123.45' : string[79]
 TokenError                       False : boolean

 F1-Help Esc-Abort
```

Figure 5.2: The Global Pane Local Menu

Inspect

Opens an Inspector window that shows you the contents of the currently highlighted global symbol. See Chapter 6 for more information on how inspector windows behave.

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

If the variable you inspect has a name that is superseded by a local variable with the same name, you will see the actual value of the global variable, not the local one. This behavior 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. The different behavior gives you a convenient way of looking at the value of global variables whose names are also used as local variables.

Change

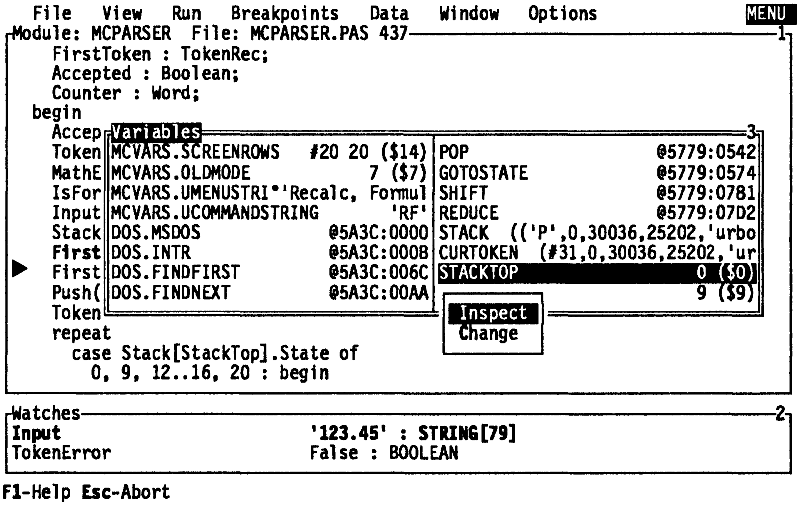
Changes the value of the currently selected (highlighted) global symbol to the value you enter at the prompt. Turbo Debugger performs any necessary data type conversion exactly as if the assignment operator for your current language had been used to change the variable. See Chapter 9 for more information on assignment and data type conversion.

You can also change the value of the currently highlighted symbol by simply starting to type a new value. When you do this, the same prompt 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 the command.

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



```
Module: MCPARSER  File: MCPARSER.PAS 437                                  1
    FirstToken : TokenRec;
    Accepted : Boolean;
    Counter : Word;
  begin
    Accep Variables                                                        3
    Token MCVARS.SCREENROWS   #20 20 ($14) POP              @5779:0542
    MathE MCVARS.OLDMODE             7 ($7) GOTOSTATE        @5779:0574
    IsFor MCVARS.UMENUSTRI•'Recalc, Formul SHIFT            @5779:0781
    Input MCVARS.UCOMMANDSTRING         'RF' REDUCE           @5779:07D2
    Stack DOS.MSDOS             @5A3C:0000 STACK  (('P',0,30036,25202,'urbo
    First DOS.INTR              @5A3C:000B CURTOKEN  (#31,0,30036,25202,'ur
 ▶  First DOS.FINDFIRST         @5A3C:006C STACKTOP                   0 ($0)
    Push( DOS.FINDNEXT          @5A3C:00AA                            9 ($9)
    Token
    repeat
      case Stack[StackTop].State of
        0, 9, 12..16, 20 : begin

Watches                                                                    2
Input                         '123.45' : STRING[79]
TokenError                    False : BOOLEAN
```

Figure 5.3: The Static Pane Local Menu

Inspect

Opens an Inspector window that displays the contents of the currently highlighted module's local symbol. See Chapter 6 for more information on how inspectors behave.

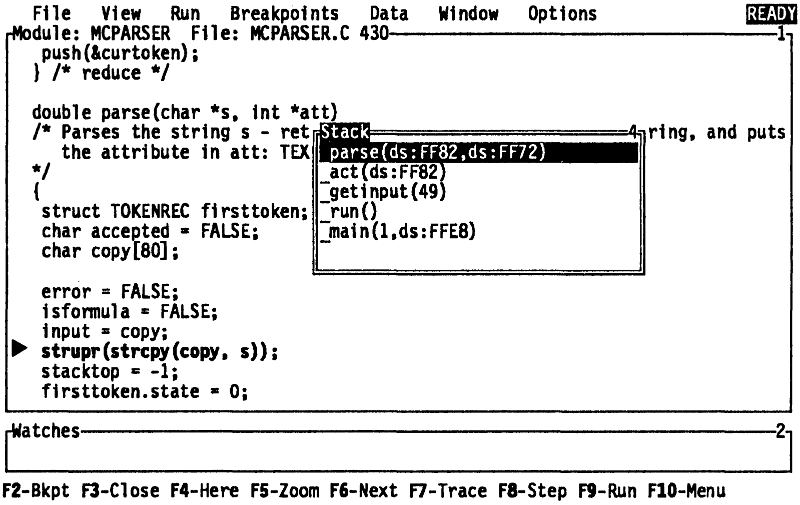
Change

Changes the value of the currently selected (highlighted) local symbol to the value you enter at the prompt. 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. See Chapter 9 for more information on assignment and data type conversion.

You can also change the value of the currently highlighted symbol by simply starting to type a new value. When you do this, the same prompt box appears as if you had first specified the Change command.

The Stack Window

You create a Stack window with *F10*/View/Stack. The Stack window lists all the 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.



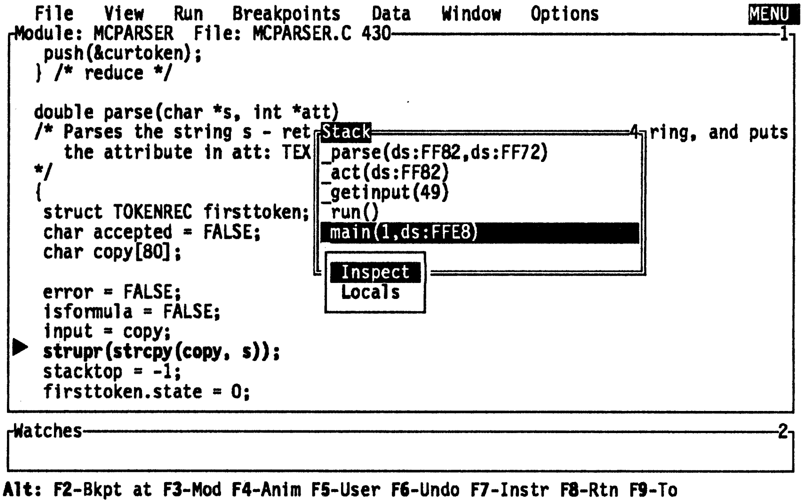
```
  File   View   Run   Breakpoints   Data   Window   Options              READY
┌Module: MCPARSER  File: MCPARSER.C 430──────────────────────────────────────1┐
│  push(&curtoken);                                                           │
│ } /* reduce */                                                              │
│                                                                             │
│ double parse(char *s, int *att)                                             │
│ /* Parses the string s - ret┌Stack═════════════════════════════4┐ring, and puts
│    the attribute in att: TEX│ parse(ds:FF82,ds:FF72)             │          │
│ */                          │_act(ds:FF82)                       │          │
│ {                           │_getinput(49)                       │          │
│  struct TOKENREC firsttoken;│_run()                              │          │
│  char accepted = FALSE;     │_main(1,ds:FFE8)                    │          │
│  char copy[80];             │                                    │          │
│                             └════════════════════════════════════┘          │
│  error = FALSE;                                                             │
│  isformula = FALSE;                                                         │
│  input = copy;                                                              │
│▶ strupr(strcpy(copy, s));                                                   │
│  stacktop = -1;                                                             │
│  firsttoken.state = 0;                                                      │
└─────────────────────────────────────────────────────────────────────────────┘
┌Watches─────────────────────────────────────────────────────────────────────2┐
│                                                                             │
└─────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 5.4: The Stack Window

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

The Stack Window Local Menu

This local menu has two commands: Inspect and Locals.



```
  File   View   Run   Breakpoints   Data   Window   Options                    MENU
┌Module: MCPARSER  File: MCPARSER.C 430──────────────────────────────────────────1┐
   push(&curtoken);
  } /* reduce */

  double parse(char *s, int *att)
  /* Parses the string s - ret┌Stack═══════════════════════════4┐ring, and puts
     the attribute in att: TEX│_parse(ds:FF82,ds:FF72)          │
  */                          │_act(ds:FF82)                    │
  {                           │_getinput(49)                    │
   struct TOKENREC firsttoken;│_run()                           │
   char accepted = FALSE;     │ main(1,ds:FFE8)                 │
   char copy[80];             │┌────────┐                       │
                              └│Inspect │═══════════════════════┘
   error = FALSE;              │Locals  │
   isformula = FALSE;          └────────┘
   input = copy;
▶  strupr(strcpy(copy, s));
   stacktop = -1;
   firsttoken.state = 0;
└─────────────────────────────────────────────────────────────────────────────────┘
┌Watches─────────────────────────────────────────────────────────────────────────2┐
└─────────────────────────────────────────────────────────────────────────────────┘
Alt: F2-Bkpt at F3-Mod F4-Anim F5-User F6-Undo F7-Instr F8-Rtn F9-To
```

Figure 5.5: The Stack Window Local Menu

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 window is positioned on the line in the function that will be executed after the called function returns.

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

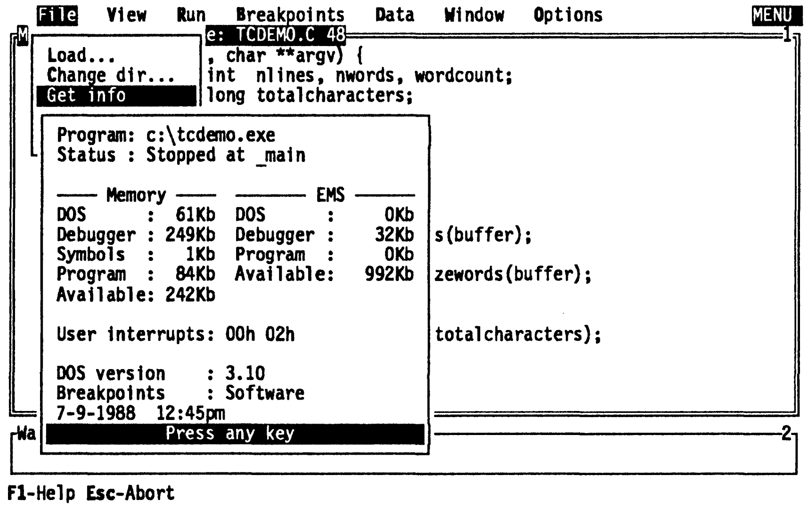
Locals

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 function, this command lets you look at the local variables in that instance of the function.

The Origin Local Menu Command

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

Viewing Execution Status: The Get Info Command



```
  File  View  Run  Breakpoints  Data  Window  Options                        MENU
 M                 e: TCDEMO.C 48                                              1
   Load...        , char **argv) {
   Change dir...  int  nlines, nwords, wordcount;
   Get info       long totalcharacters;

    Program: c:\tcdemo.exe
    Status : Stopped at _main

    ─── Memory ───   ──── EMS ────
    DOS      :  61Kb DOS       :    0Kb
    Debugger : 249Kb Debugger  :   32Kb  s(buffer);
    Symbols  :   1Kb Program   :    0Kb
    Program  :  84Kb Available:   992Kb  zewords(buffer);
    Available: 242Kb

    User interrupts: 00h 02h             totalcharacters);

    DOS version    : 3.10
    Breakpoints    : Software
    7-9-1988  12:45pm
 Wa             Press any key                                                  2

F1-Help Esc-Abort
```

Figure 5.6: The File Get Info Command

You can choose File/Get Info to look at memory usage and to determine why the debugger gained control. This and other information appears in a box 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 usage appears to the right of main memory usage

- A list of interrupts intercepted by the program you are debugging
- The DOS version you're running
- Whether breakpoints are handled entirely in software or if they have hardware assistance

Here are the messages you will see on the second line, describing why your program stopped:

Stopped at __
Your program stopped as the result of a **Run/Execute To**, **Run/Go to Cursor**, or **Run/Until Return** command completing. This status line also displays 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 any 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* after you reconfigured the *Break* key to something else.

Trace
You executed a single source line or machine instruction with *F7* or *F10*/**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 numerical 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 *F10*/**Run/Program** Reset.

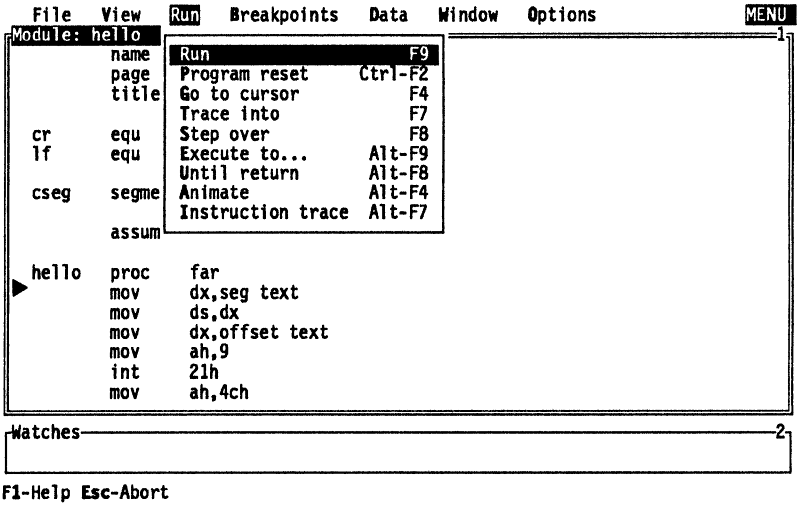
Loaded
You loaded Turbo Debugger and specified a program *and* the option that prevents the compiler startup code from executing. No instructions have been executed at this point, including those that set up your stack and segment registers. This means that if you try to examine certain data in your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

Step
You executed a single source line or machine instruction, skipping function calls, with *F8* or *F10*/Run/Step.

Interrupt
You pressed the interrupt key (usually *Ctrl-Break*) to regain control. Your program is immediately interrupted and the debugger restarted.

The Run Menu

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



```
  File   View   Run   Breakpoints   Data   Window   Options                 MENU
 Module: hello                                                                1
         name    Run                 F9
         page    Program reset  Ctrl-F2
         title   Go to cursor        F4
                 Trace into          F7
  cr     equ     Step over           F8
  lf     equ     Execute to...   Alt-F9
                 Until return    Alt-F8
  cseg   segme   Animate         Alt-F4
                 Instruction trace Alt-F7
         assum

  hello  proc    far
 ▶       mov     dx,seg text
         mov     ds,dx
         mov     dx,offset text
         mov     ah,9
         int     21h
         mov     ah,4ch

 Watches                                                                      2

 F1-Help Esc-Abort
```

Figure 5.7: The Run Menu

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*

Program Reset [*Ctrl-F2*]

Reloads the program you're debugging from disk. Use this when you've executed "too far," that is, passed the place where a bug occurred.

If you're in a Module or CPU window, the display won't return to the start of the program. Instead, you'll stay exactly where you were when you chose the **Program Reset** command. If you chose the **Program Reset** command because you just executed one source statement more than you intended, you can position the cursor up a few lines in you source file and press *F4* to run to that location.

Go to Cursor [*F4*]

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

Trace Into [*F7*]

Executes a single source line or machine instruction. Usually, a single source line is executed. If the current line contains any procedure or function calls, Turbo Debugger traces into that routine. However, if the current window is a CPU window, only a single machine instruction is executed.

Step Over [*F8*]

Executes a single source line or machine instruction, skipping over any procedure or function call(s). This 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 same line, so you don't end up at the start of one of those 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 step over a single machine instruction, Turbo Debugger treats certain instructions as a single instruction, even when they cause multiple 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, CMPSB, CMPSW, LODSB, LODSW, MOVS, MOVSB, MOVSW, SCAS, SCASB, SCASW, STOS, STOSB, STOSW.**

Execute To [*Alt-F9*]

Executes your program until the address you specify at the prompt is reached. The address you specify may never be reached if a breakpoint action is encountered first or 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 *F10*/Run/Trace instead of *F10*/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 [*Alt-F4*]

Performs a continuous series of Trace commands, updating the screen after each one. This lets you watch the current location in your source code and see the values of variables changing. You can interrupt this command by pressing any key.

After activating *Alt-F4*, you will be prompted for a time delay between successive traces. The time delay is measured in tenths of a second; the default is three.

Instruction Trace [*Alt-F7*]

Executes a single 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.

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 action, but the code is never reached.

Ctrl-Break

This key combination will almost always interrupt your program and return control to the debugger. This key combination takes effect as soon as the key is pressed, so you can sometimes appear to be in an unexpected piece of code. This code could be 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 to change this key. You can set the interrupt key to be any normal key pressed in combination with *Ctrl*.

Terminating Your Program

When your program terminates and exits back to DOS, the 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 Debug Session

Turbo Debugger has several features that make restarting a debug session as painless as possible. When debugging a program, it's easy to go just a little too far, overshooting the real cause of the problem. What you want to do then is restart debugging, but suspend execution before the last few commands that caused you to miss the problem that you wanted to observe.

Most debuggers force you to manually type in 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, press *F10*/**Run/Program** Reset. Turbo Debugger reloads the program from disk, with any data you may 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 watches intact.**

Keystroke Recording and Playback

You can use the keystroke macro facility to record keystroke sequences that you use frequently. When 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 the debugger up 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 technique to record keystrokes that must be typed to your program. You can only record Turbo Debugger command keystrokes.

To record your entire session, the first thing you must do after starting Turbo Debugger from DOS is to define a keystroke macro. Choose Options/Macros/Create to do this. You'll be 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 Macros/Stop Recording. Now save the macro to disk by choosing the Options/Save Options command. 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. For programs that do not require you to enter anything, this keystroke-recording mechanism can completely automate the restarting procedure, saving many keystrokes.

Note: When a macro is saved to a configuration file, the configuration of the total environment is saved, including opened view windows and zoomed windows. Thus if you record a macro that opens a view window and don't close the window before saving it, the next time you restore that configuration file, the window will be automatically opened without executing the macro.

Loading a New Program to Debug

You load a new program to debug with *F10*/File/Load. You can use DOS-style wildcards to get a list of file choices or type a specific file name to load.

If you press *Enter* after the prompt appears, a list of all the .EXE files in the current directory will be displayed. 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 will move 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, namely, *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 *F10*/Options/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 usually answer Yes, since for most programs, the new arguments will only take effect when you first load the program.

CHAPTER

6

Examining and Modifying Data

Turbo Debugger provides a unique and intuitive way to peruse your program's data. Inspectors let you examine 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.

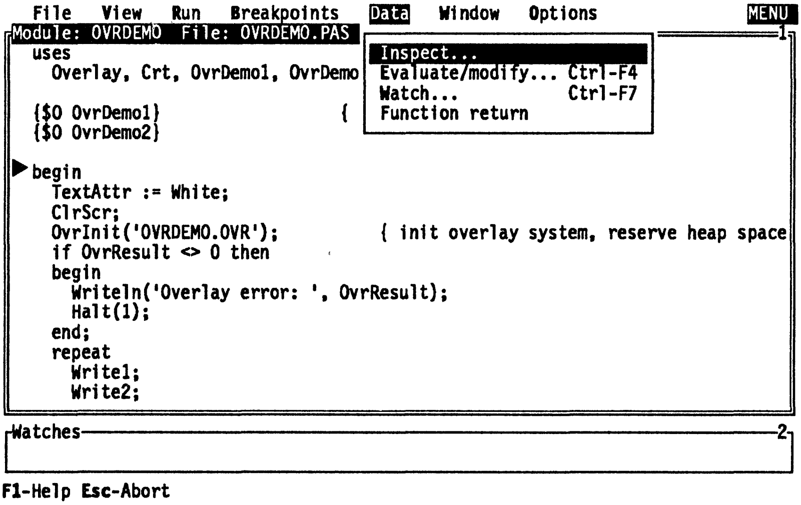
This chapter presumes you understand the various data types that can be used in the language you're using (C, Pascal, or assembler). If you are fairly new to a language and have not yet explored all its data types, this chapter can still give you valuable information about the basic data types (char, int, integer, boolean, real, and so on). When you have delved into the more involved data types (pointers, records, structs, unions, and so on), return to this chapter to learn more about looking at them with Turbo Debugger.

This chapter shows you how to examine and modify variables in your program. First, we explain the **D**ata command and its options. We then discuss how you can modify program data by evaluating expressions that have side effects. Next, we show you how to point directly at data items in your source modules. We introduce the Watches window and, finally, describe the way that the basic data types of each language appear in inspectors.

If you want to examine or modify arbitrary blocks of memory as hex data bytes, refer to Chapter 10, which covers assembler-level debugging.

The Data Menu

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



```
  File   View   Run   Breakpoints   Data   Window   Options                MENU
 Module: OVRDEMO  File: OVRDEMO.PAS  ┌─────────────────────────────┐ ══════════1
  uses                               │ Inspect...                  │
    Overlay, Crt, OvrDemo1, OvrDemo  │ Evaluate/modify...  Ctrl-F4 │
                                     │ Watch...            Ctrl-F7 │
  {$O OvrDemo1}                    { │ Function return             │
  {$O OvrDemo2}                      └─────────────────────────────┘

 ►begin
    TextAttr := White;
    ClrScr;
    OvrInit('OVRDEMO.OVR');            { init overlay system, reserve heap space
    if OvrResult <> 0 then
    begin
      Writeln('Overlay error: ', OvrResult);
      Halt(1);
    end;
    repeat
      Write1;
      Write2;

 Watches────────────────────────────────────────────────────────────────────2


 F1-Help Esc-Abort
```

Figure 6.1: The Data Menu

Inspect

Prompts you for the variable that references the data you wish 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, as long as that expression references a memory location and doesn't just evaluate to a constant.

If the cursor is in a Text pane when you issue this command, the prompt automatically contains the variable at the cursor, if any. If you select an expression (using *Ins*), the prompt 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, you can "walk" through your program's data structures as easily as you scroll through your source code in a Module window.

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

Evaluate/Modify

Prompts you for the expression to evaluate, then evaluates it, exactly as the compiler would. See Chapter 9 for a complete discussion of expressions.

If the cursor is in a Text pane when you issue this command, the prompt automatically contains the variable at the cursor, if any. If you mark an expression using *Ins*, the prompt is initialized to the marked expression.

Remember that you can add a format control string after the expression that you want to watch. See Chapter 9 for a discussion of format control. This is useful when you want to watch something but have it displayed in a format other than Turbo Debugger's default display mode for the data type.

The prompt box has three panes. You type the expression you want to evaluate in the top pane. The pane has a history list, just like other input prompts. The middle pane displays the result of evaluating your expression. The bottom pane is an input area where you can enter a new value for the expression. If the expression can't be modified, this pane reads "Cannot be changed" and you can't move your cursor to that pane.

You move between the panes using the *Tab* and *Shift-Tab* keys, just as in other windows that have panes. Your entry in the Evaluate or New entry pane takes effect when you press *Enter*. Pressing *Esc* while inside any pane removes the prompt box.

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. Chapter 9 describes the format control string in more detail.

When you type the name of just one of your program variables, Turbo Debugger displays its value. If you want a quick look at the value, this is more convenient than opening an inspector, looking at the value, and then deleting the inspector. You can also use this command as a simple calculator by using numbers as operands instead of program variables.

Note for 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++**.

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.

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

Function Return

Shows you the value the current function is about to return. You can only use this command 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.

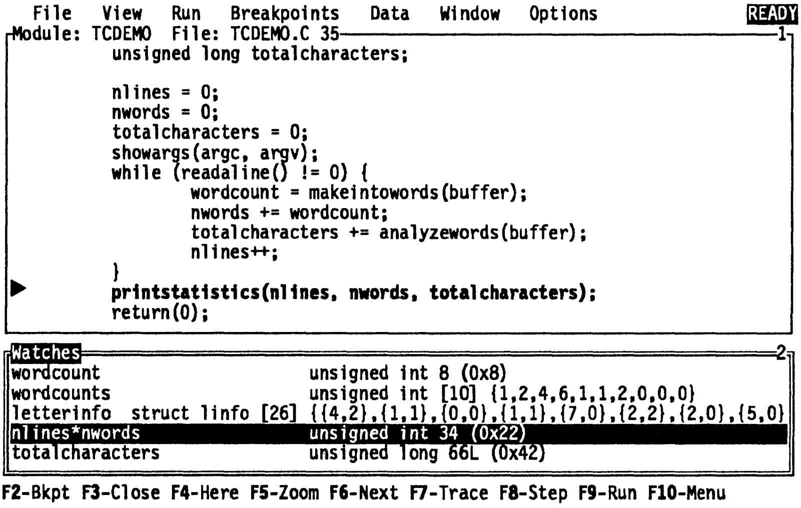
This command prevents you from having to switch to a CPU window to examine the return value that is placed in the CPU registers.

Pointing at Data Items in Source Files

Turbo Debugger has a powerful mechanism to relieve you from always typing in the names of program variables that you wish 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 to inspect by pressing *Ins* and using the cursor keys to highlight it before choosing the Inspect command. See Chapter 8 for a full discussion of using Module windows.

The Watches Window

The Watches window lets you list variables and expressions in your program whose values you wish to track. You can watch the value of simple variables, such as integers, and also watch the contents of complex data items, 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, $x * y + 4$.



```
  File   View   Run   Breakpoints   Data   Window   Options                   READY
Module: TCDEMO  File: TCDEMO.C 35                                                1
            unsigned long totalcharacters;

            nlines = 0;
            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);

Watches                                                                          2
wordcount                     unsigned int 8 (0x8)
wordcounts                    unsigned int [10] {1,2,4,6,1,1,2,0,0,0}
letterinfo  struct linfo [26] {{4,2},{1,1},{0,0},{1,1},{7,0},{2,2},{2,0},{5,0}
nlines*nwords                 unsigned int 34 (0x22)
totalcharacters               unsigned long 66L (0x42)

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

Figure 6.2: The Watches Window

Choose View/Watches to create a Watches window. It shows a list of variables or expressions whose values you want to watch. For each item, the variable name or expression appears to the left and its data type and value to 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 and a bullet (•) indicates the omission.

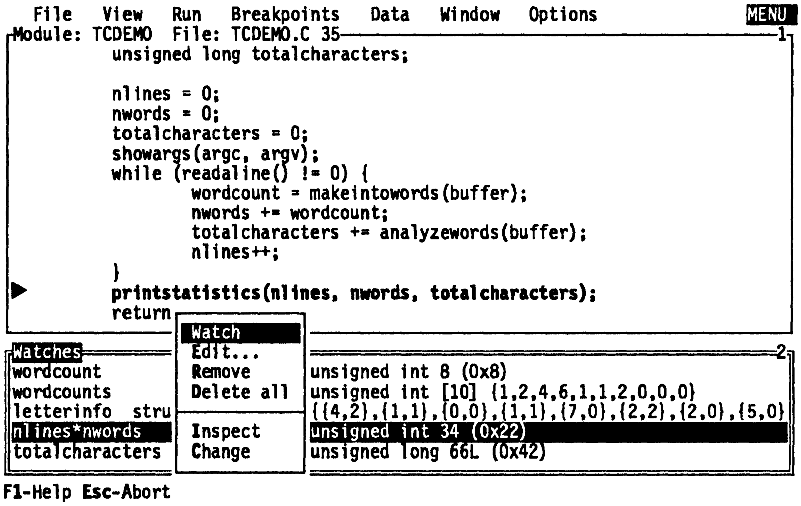
When you enter an expression to be watched, feel free to use variable names that are not yet valid because they are in a function that has not yet been called. This lets you set up a watch expression before its scope becomes active. (See Chapter 9 for a complete discussion of scopes and when a variable or parameter is valid.) This is the only situation in Turbo Debugger where you can enter an expression that cannot be immediately evaluated.

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 will become 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, expressions in the Watches window have the same value as if they appear in your program at the place where it 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 (*????*).

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 the command.



```
Module: TCDEMO  File: TCDEMO.C 35
        unsigned long totalcharacters;

        nlines = 0;
        nwords = 0;
        totalcharacters = 0;
        showargs(argc, argv);
        while (readaline() != 0) {
                wordcount = makeintowords(buffer);
                nwords += wordcount;
                totalcharacters += analyzewords(buffer);
                nlines++;
        }
▶       printstatistics(nlines, nwords, totalcharacters);
        return
```

```
Watches
wordcount          unsigned int 8 (0x8)
wordcounts         unsigned int [10] {1,2,4,6,1,1,2,0,0,0}
letterinfo  stru   {{4,2},{1,1},{0,0},{1,1},{7,0},{2,2},{2,0},{5,0}
nlines*nwords      unsigned int 34 (0x22)
totalcharacters    unsigned long 66L (0x42)
```

Figure 6.3: The Watches Window Local Menu

Watch

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

Edit

Lets you edit the expression in the Watches window. You can change the watch 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 you want to change.

Remove

Removes the currently selected item from the Watches window.

Delete All

Removes all the items from the Watches window. Use the Watch command to view more variables. 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.

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 allows you to view all its elements, not just the ones that fit in the Watches window. (The section "Inspectors" on page 88 explains all about Inspector windows.)

Change

Changes the value of the currently highlighted item in the Watches window to the value you enter at the prompt. 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. See Chapter 9 for more information on the assignment operator and type conversion (casting).

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 perused. 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.

To examine the contents of an Inspector window as raw data bytes, select the View/CPU command while you're in the Inspector window. The CPU window will come up with the data pane 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 (or *F3*).

The following section describes the different Inspector windows that can appear for each of the languages supported by Turbo Debugger: C, Pascal, and assembler. The language being used dictates the format of the information displayed in Inspector windows. Data items and their values always appear in a format similar to the way they were declared 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

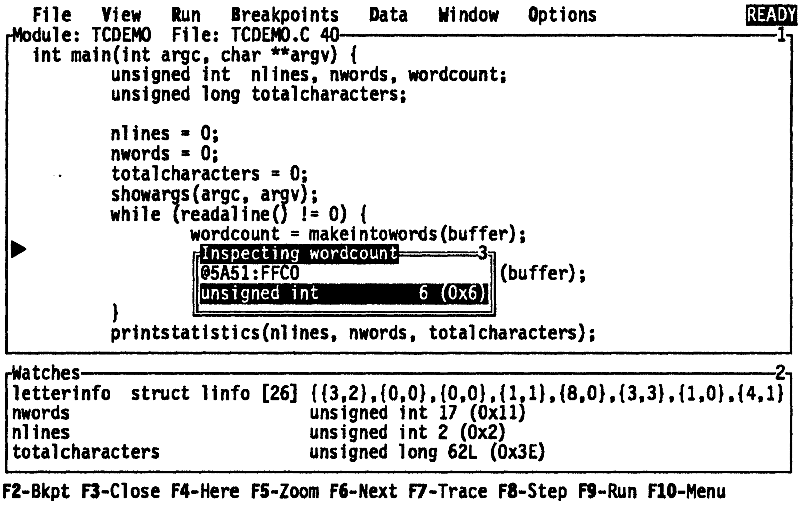
Scalars

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

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

These Inspector windows only have a single line of information following the top line that describes the address of the variable. To the left 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 character equivalent 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.



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

```
Module: TCDEMO  File: TCDEMO.C 40                                             1
 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);
►               Inspecting wordcount          3
                @5A51:FFC0                      (buffer);
                unsigned int         6 (0x6)
        }
        printstatistics(nlines, nwords, totalcharacters);

Watches                                                                       2
letterinfo  struct linfo [26] {{3,2},{0,0},{0,0},{1,1},{8,0},{3,3},{1,0},{4,1}
nwords                        unsigned int 17 (0x11)
nlines                        unsigned int 2 (0x2)
totalcharacters               unsigned long 62L (0x3E)
```

Figure 6.4: A Scalar Inspector Window

Pointers

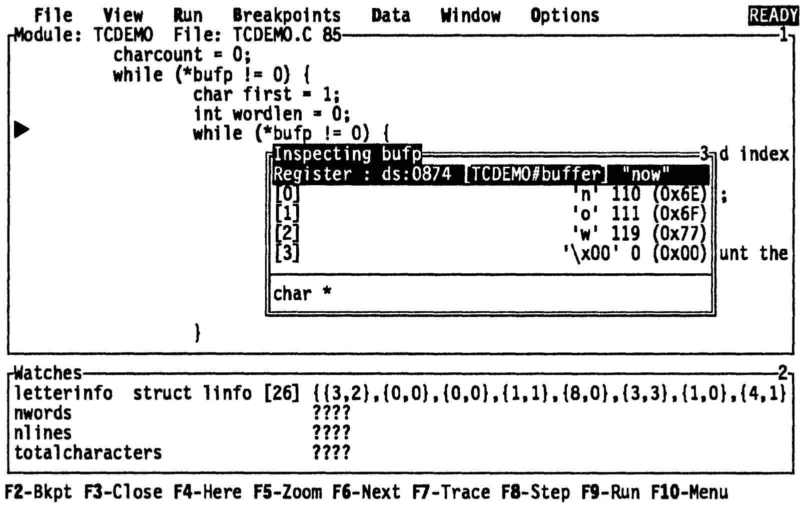
Pointer Inspector windows show you the value of data items that point to other data items, such as

```
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 .

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, as much of it as possible will be 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.



```
  File   View   Run   Breakpoints   Data   Window   Options                   READY
┌Module: TCDEMO  File: TCDEMO.C 85──────────────────────────────────────────────1┐
│         charcount = 0;
│         while (*bufp != 0) {
│                 char first = 1;
│                 int wordlen = 0;
▶                 while (*bufp != 0) {
│                         ┌Inspecting bufp═══════════════════════════3┐d index
│                         │Register : ds:0874 [TCDEMO#buffer] "now"   │
│                         │[0]                       'n' 110 (0x6E)   │;
│                         │[1]                       'o' 111 (0x6F)   │
│                         │[2]                       'w' 119 (0x77)   │
│                         │[3]                    '\x00' 0 (0x00)     │unt the
│                         ├───────────────────────────────────────────┤
│                         │char *                                     │
│                         └───────────────────────────────────────────┘
│                 }
└────────────────────────────────────────────────────────────────────────────────┘
┌Watches──────────────────────────────────────────────────────────────────────2┐
│letterinfo  struct linfo [26] {{3,2},{0,0},{0,0},{1,1},{8,0},{3,3},{1,0},{4,1}│
│nwords                        ????
│nlines                        ????
│totalcharacters               ????
└────────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

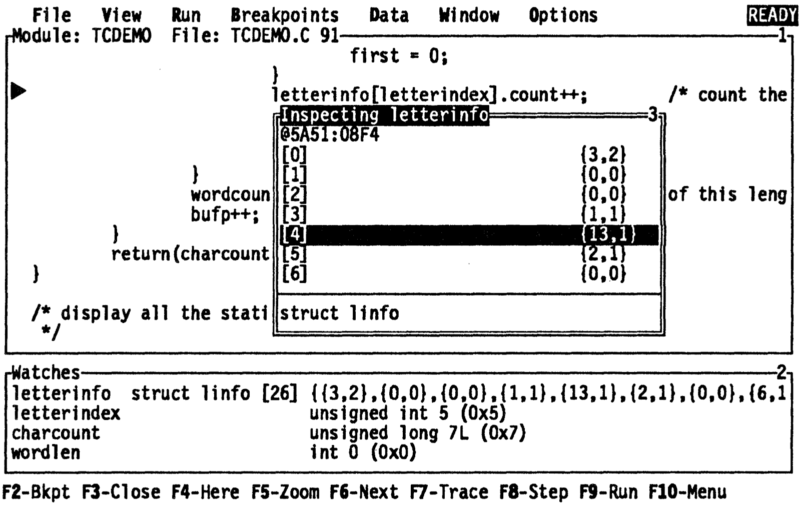
Figure 6.5: A C Pointer Inspector Window

Arrays

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

```
long threed[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 being pointed to. If the value is a complex data item such as a structure or array, as much of it as possible will be displayed, with the values enclosed in braces ({ and }).



```
  File   View   Run   Breakpoints   Data   Window   Options                    READY
┌Module: TCDEMO  File: TCDEMO.C 91────────────────────────────────────────────────1┐
│                                    first = 0;                                    │
│                               }                                                  │
│▶                              letterinfo[letterindex].count++;        /* count the│
│                              ┌Inspecting letterinfo═══════════════3┐             │
│                              │@5A51:08F4                           │             │
│                              │[0]                       {3,2}      │             │
│                 }            │[1]                       {0,0}      │             │
│                 wordcoun     │[2]                       {0,0}      │of this leng │
│                 bufp++;      │[3]                       {1,1}      │             │
│             }                │[4]                       {13,1}     │             │
│             return(charcount │[5]                       {2,1}      │             │
│   }                          │[6]                       {0,0}      │             │
│                              ├─────────────────────────────────────┤             │
│   /* display all the stati   │struct linfo                         │             │
│    */                        └─────────────────────────────────────┘             │
└──────────────────────────────────────────────────────────────────────────────────┘
┌Watches──────────────────────────────────────────────────────────────────────────2┐
│letterinfo  struct linfo [26] {{3,2},{0,0},{0,0},{1,1},{13,1},{2,1},{0,0},{6,1   │
│letterindex                   unsigned int 5 (0x5)                                │
│charcount                     unsigned long 7L (0x7)                              │
│wordlen                       int 0 (0x0)                                         │
└──────────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

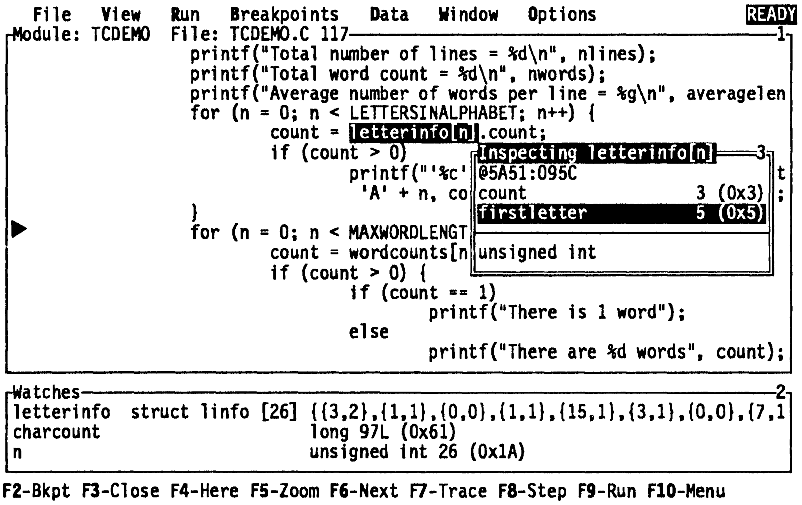
Figure 6.6: A C Array Inspector Window

Structure and Union

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.



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

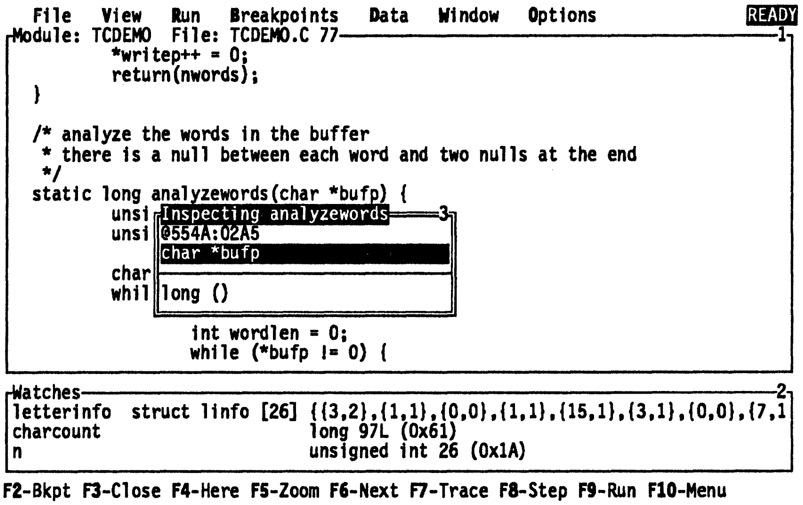
```
Module: TCDEMO  File: TCDEMO.C 117                                        1
            printf("Total number of lines = %d\n", nlines);
            printf("Total word count = %d\n", nwords);
            printf("Average number of words per line = %g\n", averagelen
            for (n = 0; n < LETTERSINALPHABET; n++) {
                    count = letterinfo[n].count;
                    if (count > 0)          Inspecting letterinfo[n]      3
                            printf("'%c' @5A51:095C                        t
                             'A' + n, co count                   3 (0x3) ;
            }                           firstletter             5 (0x5)
 ▶          for (n = 0; n < MAXWORDLENGT
                    count = wordcounts[n unsigned int
                    if (count > 0) {
                            if (count == 1)
                                    printf("There is 1 word");
                            else
                                    printf("There are %d words", count);

Watches                                                                   2
letterinfo  struct linfo [26] {{3,2},{1,1},{0,0},{1,1},{15,1},{3,1},{0,0},{7,1
charcount                     long 97L (0x61)
n                             unsigned int 26 (0x1A)
```

Figure 6.7: A C Structure or Union Inspector Window

Structures and unions appear the same in Inspector windows. The title 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.

Function

Function Inspector windows show the return type of the function as part of the title. Each parameter a function is called with appears after the memory address at the top of the list.



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

```
Module: TCDEMO  File: TCDEMO.C 77                                        1
          *writep++ = 0;
          return(nwords);
  }

  /* analyze the words in the buffer
   * there is a null between each word and two nulls at the end
   */
  static long analyzewords(char *bufp) {
          unsi Inspecting analyzewords   3
          unsi @554A:02A5
               char *bufp

          char
          whil long ()

                  int wordlen = 0;
                  while (*bufp != 0) {
Watches                                                                   2
letterinfo  struct linfo [26] {{3,2},{1,1},{0,0},{1,1},{15,1},{3,1},{0,0},{7,1
charcount                     long 97L (0x61)
n                             unsigned int 26 (0x1A)
```

Figure 6.8: A C Function Inspector Window

Function Inspector windows give you information about the calling parameters, return data type, and calling conventions for a function, for example,

```
static int near pascal readit(char *buf, int count) {
}

char *nexterror(int errnum) {
}
```

Pascal Data Inspector Windows

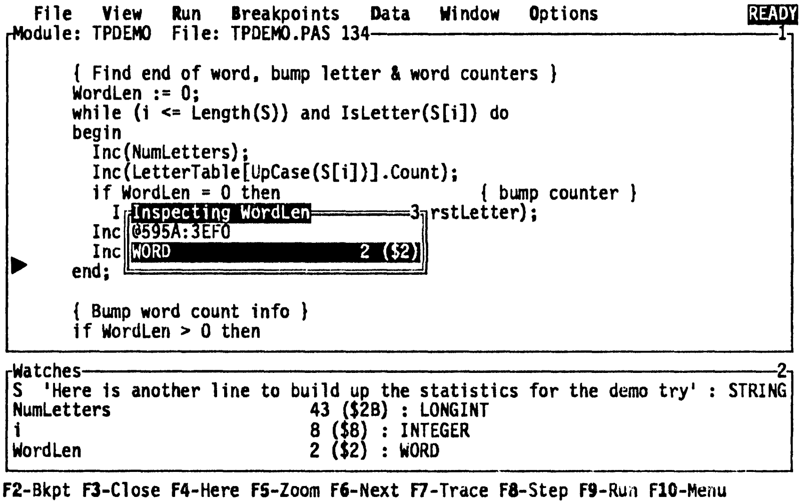
Scalars

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

```
var
  X : integer;
  Y : longint;
```

These Inspector windows only have a single line of information following the top line that describes 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 of $). 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 # followed by a number to display the character value. This character value appears before the decimal or hex values.


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

```
Module: TPDEMO  File: TPDEMO.PAS 134                                    1

      { 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);
        if WordLen = 0 then                         { bump counter }
          I Inspecting WordLen          3 rstLetter);
        Inc @595A:3EF0
        Inc WORD                  2 ($2)
▶     end;

      { Bump word count info }
      if WordLen > 0 then

Watches                                                                  2
S  'Here is another line to build up the statistics for the demo try' : STRING
NumLetters                       43 ($2B) : LONGINT
i                                8 ($8) : INTEGER
WordLen                          2 ($2) : WORD
```

Figure 6.9: A Pascal Scalar Inspector Window

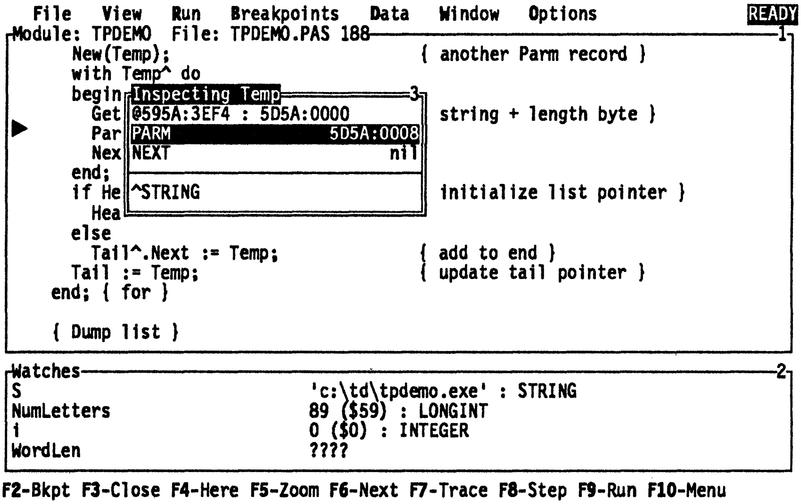
Pointers

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 only have a single line of information following the top line that describes 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, as much of it as possible will be displayed, with the values enclosed in parentheses.

You will also get multiple lines if you opened the Inspector window and issued the Range local command and specified a count greater than 1.


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

```
Module: TPDEMO  File: TPDEMO.PAS 188                                        1
      New(Temp);                           { another Parm record }
      with Temp^ do
      begin Inspecting Temp              3
        Get  @595A:3EF4 : 5D5A:0000         string + length byte }
 ▶      Par  PARM                5D5A:0008
        Nex  NEXT                      nil
      end;
      if He  ^STRING                        initialize list pointer }
        Hea
      else
        Tail^.Next := Temp;                { add to end }
      Tail := Temp;                        { update tail pointer }
    end; { for }

    { Dump list }
```

```
Watches                                                                     2
S                            'c:\td\tpdemo.exe' : STRING
NumLetters                   89 ($59) : LONGINT
i                            0 ($0) : INTEGER
WordLen                      ????
```

Figure 6.10: A Pascal Pointer Inspector Window

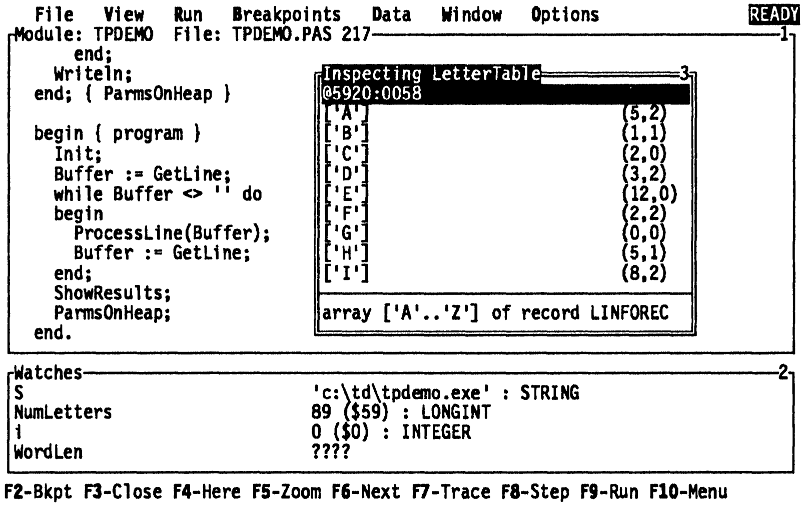
Arrays

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 will be displayed, with the values enclosed in parentheses.

You can use the Range 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.



```
 File  View  Run  Breakpoints  Data  Window  Options                     READY
┌Module: TPDEMO  File: TPDEMO.PAS 217─────────────────────────────────────────1┐
│   end;                                                                       │
│  Writeln;                          ┌Inspecting LetterTable══════════════3┐   │
│ end; { ParmsOnHeap }               │@5920:0058                           │   │
│                                    │['A']                          (5,2) │   │
│ begin { program }                  │['B']                          (1,1) │   │
│   Init;                            │['C']                          (2,0) │   │
│   Buffer := GetLine;               │['D']                          (3,2) │   │
│   while Buffer <> '' do            │['E']                         (12,0) │   │
│   begin                            │['F']                          (2,2) │   │
│     ProcessLine(Buffer);           │['G']                          (0,0) │   │
│     Buffer := GetLine;             │['H']                          (5,1) │   │
│   end;                             │['I']                          (8,2) │   │
│   ShowResults;                     ├─────────────────────────────────────┤   │
│   ParmsOnHeap;                     │array ['A'..'Z'] of record LINFOREC  │   │
│ end.                               └═════════════════════════════════════┘   │
└──────────────────────────────────────────────────────────────────────────────┘
┌Watches─────────────────────────────────────────────────────────────────────2┐
│S                             'c:\td\tpdemo.exe' : STRING                    │
│NumLetters                    89 ($59) : LONGINT                             │
│i                             0 ($0) : INTEGER                               │
│WordLen                       ????                                           │
└─────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

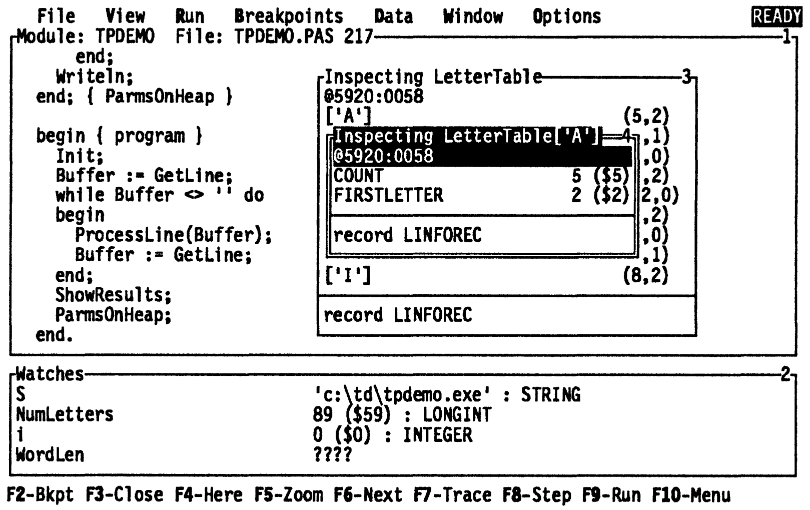
Figure 6.11: A Pascal Array Inspector Window

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.



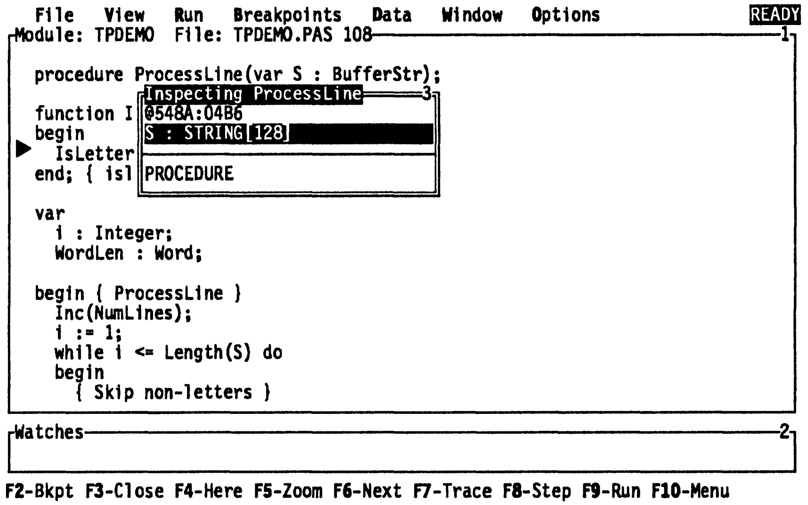
```
  File   View   Run   Breakpoints   Data   Window   Options                  READY
┌Module: TPDEMO  File: TPDEMO.PAS 217──────────────────────────────────────────1┐
│     end;                                                                      │
│   Writeln;                      ┌Inspecting LetterTable──────────3┐           │
│ end; { ParmsOnHeap }            │@5920:0058                        │           │
│                                 │['A']                        (5,2)│           │
│ begin { program }               │╔Inspecting LetterTable['A']═4╗,1)│           │
│   Init;                         │║@5920:0058                   ║,0)│           │
│   Buffer := GetLine;            │║COUNT               5 ($5)   ║,2)│           │
│   while Buffer <> '' do         │║FIRSTLETTER         2 ($2)   ║2,0)           │
│   begin                         │║─────────────────────────────║,2)│           │
│     ProcessLine(Buffer);        │║record LINFOREC              ║,0)│           │
│     Buffer := GetLine;          │╚═════════════════════════════╝,1)│           │
│   end;                          │['I']                        (8,2)│           │
│   ShowResults;                  ├──────────────────────────────────┤           │
│   ParmsOnHeap;                  │record LINFOREC                   │           │
│ end.                            └──────────────────────────────────┘           │
└───────────────────────────────────────────────────────────────────────────────┘
┌Watches───────────────────────────────────────────────────────────────────────2┐
│S                               'c:\td\tpdemo.exe' : STRING                    │
│NumLetters                      89 ($59) : LONGINT                             │
│i                               0 ($0) : INTEGER                               │
│WordLen                         ????                                           │
└───────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 6.12: A Pascal Record Inspector Window

Procedures and Functions

Procedure and Function Inspector windows in Pascal programs give you information about the calling parameters and return data type for a procedure or function, for example,

```
function times2plus(a: integer, b: longint): longint
begin
  times2plus := a * 2 + b;
end;
procedure
swap(var a,b : integer);
var
  temp := a;
  a := b;
  b := a;
end;
```



```
File  View  Run  Breakpoints  Data  Window  Options                     READY
Module: TPDEMO  File: TPDEMO.PAS 108─────────────────────────────────────────1
  procedure ProcessLine(var S : BufferStr);
                     Inspecting ProcessLine═════3
  function I║@548A:04B6
  begin     ║S : STRING[128]
▶  IsLetter ║
  end; { isl║PROCEDURE

  var
    i : Integer;
    WordLen : Word;

  begin { ProcessLine }
    Inc(NumLines);
    i := 1;
    while i <= Length(S) do
    begin
      { Skip non-letters }
Watches──────────────────────────────────────────────────────────────────────2
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 6.13: A Pascal Procedure Inspector Window

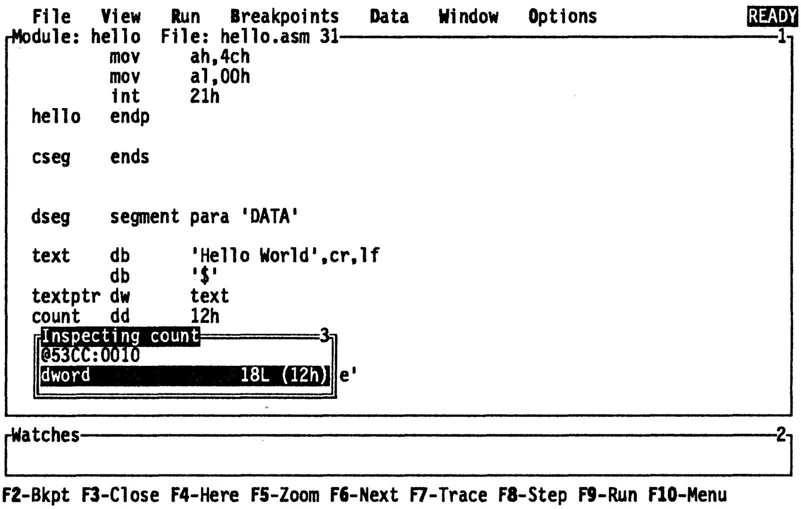
Assembler Data Inspector Windows

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 only have a single line of information following the top line that describes 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 of h). You can use TDINST to change how the value is displayed.


File View Run Breakpoints Data Window Options
READY
1
2
3
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
Module: hello  File: hello.asm 31
        mov     ah,4ch
        mov     al,00h
        int     21h
hello   endp

cseg    ends


dseg    segment para 'DATA'

text    db      'Hello World',cr,lf
        db      '$'
textptr dw      text
count   dd      12h
Inspecting count
@53CC:0010
dword                   18L (12h)  e'

Watches
```

Figure 6.14: An Assembler Scalar Inspector Window

Pointers

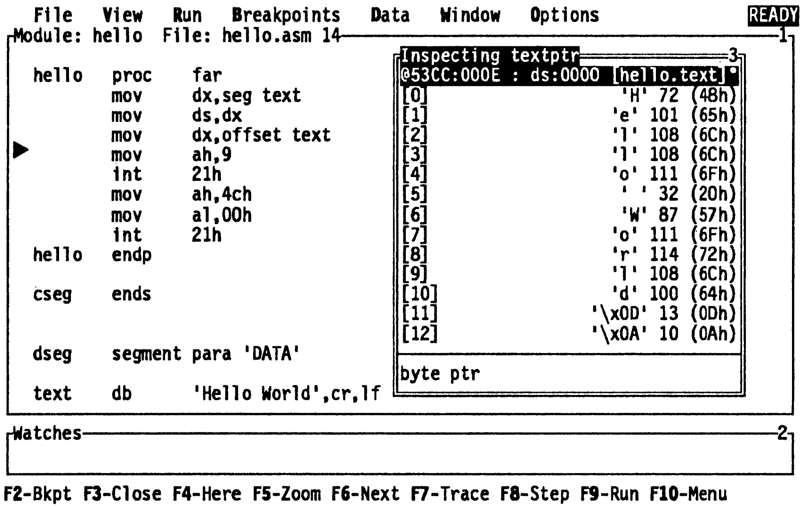
Pointer Inspector windows in assembler programs show you the value of data items that point to other data items, such as

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

Pointer Inspector windows usually only have a single line of information following the top line that describes 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, as much of it as possible will be 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 will also get multiple lines if you opened the Inspector window with a Range local command and specified a count greater than 1.


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

```
Module: hello  File: hello.asm 14                                          1
                                      Inspecting textptr                 3
hello   proc     far                  @53CC:000E : ds:0000 [hello.text]
        mov      dx,seg text          [0]                   'H' 72 (48h)
        mov      ds,dx                [1]                   'e' 101 (65h)
        mov      dx,offset text       [2]                   'l' 108 (6Ch)
▶       mov      ah,9                 [3]                   'l' 108 (6Ch)
        int      21h                  [4]                   'o' 111 (6Fh)
        mov      ah,4ch               [5]                   ' ' 32 (20h)
        mov      al,00h               [6]                   'W' 87 (57h)
        int      21h                  [7]                   'o' 111 (6Fh)
hello   endp                          [8]                   'r' 114 (72h)
                                      [9]                   'l' 108 (6Ch)
cseg    ends                          [10]                  'd' 100 (64h)
                                      [11]               '\x0D' 13 (0Dh)
                                      [12]               '\x0A' 10 (0Ah)
dseg    segment para 'DATA'
                                      byte ptr
text    db       'Hello World',cr,lf
Watches                                                                    2
```

Figure 6.15: An Assembler Pointer Inspector Window

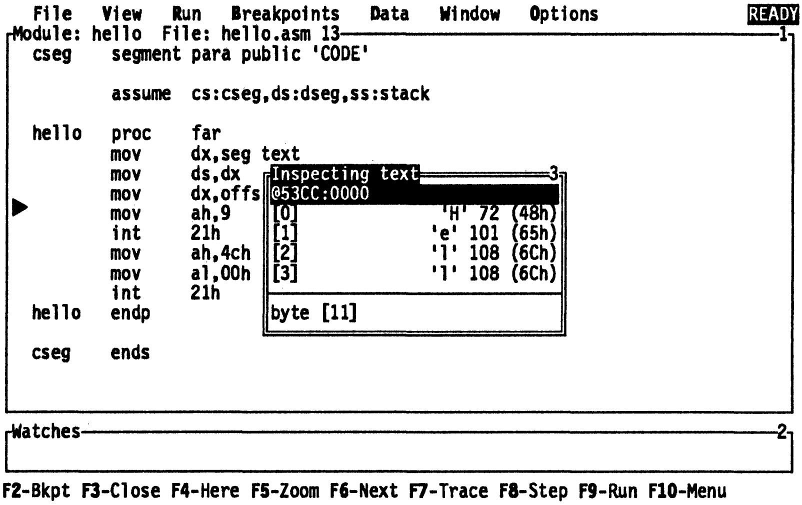
Arrays

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, as much of it as possible will be displayed, with the values enclosed in braces.

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


File View Run Breakpoints Data Window Options
READY
Module: hello File: hello.asm 13
1
Watches
2
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
cseg    segment para public 'CODE'

        assume  cs:cseg,ds:dseg,ss:stack

hello   proc    far
        mov     dx,seg text
        mov     ds,dx
        mov     dx,offs
▶       mov     ah,9
        int     21h
        mov     ah,4ch
        mov     al,00h
        int     21h
hello   endp

cseg    ends
```

```
Inspecting text                3
@53CC:0000
[0]                  'H' 72 (48h)
[1]                  'e' 101 (65h)
[2]                  'l' 108 (6Ch)
[3]                  'l' 108 (6Ch)

byte [11]
```

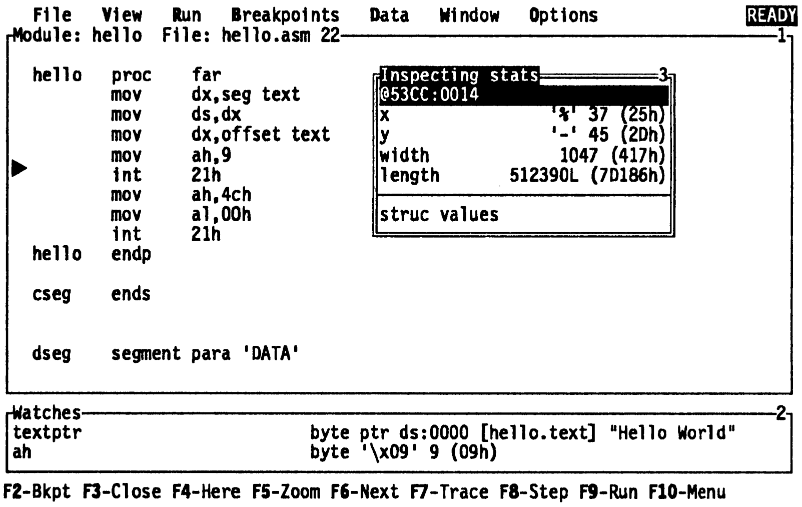
Figure 6.16: An Assembler Array Inspector Window

Structure and Union

Structure Inspector windows in assembler programs show you the value of the fields in your strucs and unions, 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.



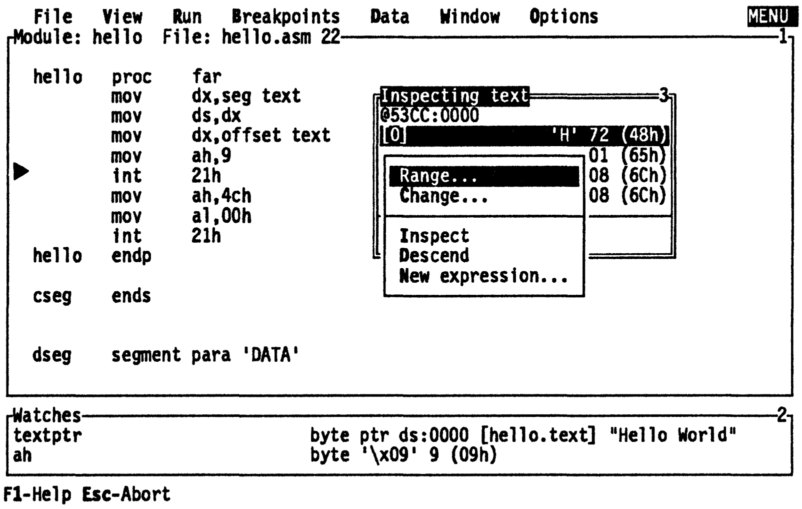
```
  File  View  Run  Breakpoints  Data  Window  Options                READDY
Module: hello  File: hello.asm 22                                          1
  hello   proc    far                   Inspecting stats             3
          mov     dx,seg text           @53CC:0014
          mov     ds,dx                 x                   '%' 37 (25h)
          mov     dx,offset text        y                   '-' 45 (2Dh)
          mov     ah,9                  width                1047 (417h)
▶         int     21h                   length       512390L (7D186h)
          mov     ah,4ch
          mov     al,00h                struc values
          int     21h
  hello   endp

  cseg    ends


  dseg    segment para 'DATA'

Watches                                                                    2
textptr                          byte ptr ds:0000 [hello.text] "Hello World"
ah                               byte '\x09' 9 (09h)
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 6.17: An Assembler Record Inspector Window

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 structures. 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 the command.



```
   File  View  Run  Breakpoints  Data  Window  Options                    MENU
┌Module: hello  File: hello.asm 22──────────────────────────────────────────1┐
│                                                                             │
│  hello  proc    far                                                         │
│         mov     dx,seg text         ┌Inspecting text════════════3┐          │
│         mov     ds,dx               │@53CC:0000                  │          │
│         mov     dx,offset text      │[0]              'H' 72 (48h)│          │
│         mov     ah,9                │┌────────────────┐01 (65h)│          │
│ ►       int     21h                 ││Range...        │08 (6Ch)│          │
│         mov     ah,4ch              ││Change...       │08 (6Ch)│          │
│         mov     al,00h              │├────────────────┤────────│          │
│         int     21h                 ││Inspect         │        │          │
│  hello  endp                        └│Descend         │════════┘          │
│                                      │New expression...│                  │
│  cseg   ends                         └────────────────┘                   │
│                                                                             │
│  dseg   segment para 'DATA'                                                 │
│                                                                             │
└─────────────────────────────────────────────────────────────────────────────┘
┌Watches────────────────────────────────────────────────────────────────────2┐
│textptr                     byte ptr ds:0000 [hello.text] "Hello World"     │
│ah                          byte '\x09' 9 (09h)                             │
└─────────────────────────────────────────────────────────────────────────────┘
F1-Help Esc-Abort
```

Figure 6.18: The Inspector Window Local Menu

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 meant was that p pointed to an array of characters, not just a single character.

Change

Changes the value of the currently highlighted item to the value you enter at the prompt. 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 wish to see each of those items.

If the current Inspector window is inspecting a function, issuing the Inspect command will show you the source code for that function.

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

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

Descend

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

Note: 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 on the screen.

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.

CHAPTER

7

Breakpoints

Turbo Debugger uses the single concept of "breakpoint" to describe the debugger functions usually referred to as breakpoints, watchpoints, and tracepoints.

Traditionally, breakpoints, watchpoints, and tracepoints are defined like this: A *breakpoint* is a place in your program where you wish 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 either a single location in your program or it can be global, where 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:

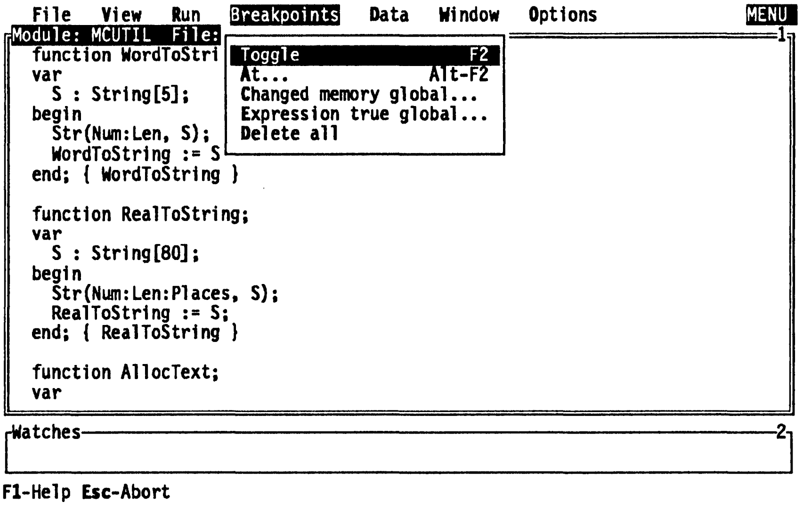
- stop program execution (a breakpoint)
- log the value of an expression
- execute an expression (splice code)

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 window and the Log window, about how to set simple breakpoints, conditional breakpoints, and breakpoints that log the value of your program variables, and finally, 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 the *F2* key. 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. There is no limit to the number of breakpoints you can set.

The Breakpoints Menu

You can access the global **B**reakpoints menu at any time by pressing the *Alt-B* hot key.



```
function WordToStri
var
  S : String[5];
begin
  Str(Num:Len, S);
  WordToString := S
end; { WordToString }

function RealToString;
var
  S : String[80];
begin
  Str(Num:Len:Places, S);
  RealToString := S;
end; { RealToString }

function AllocText;
var
```

Figure 7.1: The Breakpoints Menu

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...

Sets a breakpoint at a specific location in your program. You will be prompted for the address at which to set the breakpoint. *Alt-F2* is the shortcut.

Changed Memory Global...

Sets a breakpoint that's triggered when an area of memory changes value. You will be prompted for the area of memory to watch. For more information, see the Changed Memory command in "The Breakpoint 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 will be prompted for the expression. For more information, see the Condition Expression True command in "The Breakpoint Window Local Menu" section later in this chapter.

Delete All

Removes all the breakpoints you have set.

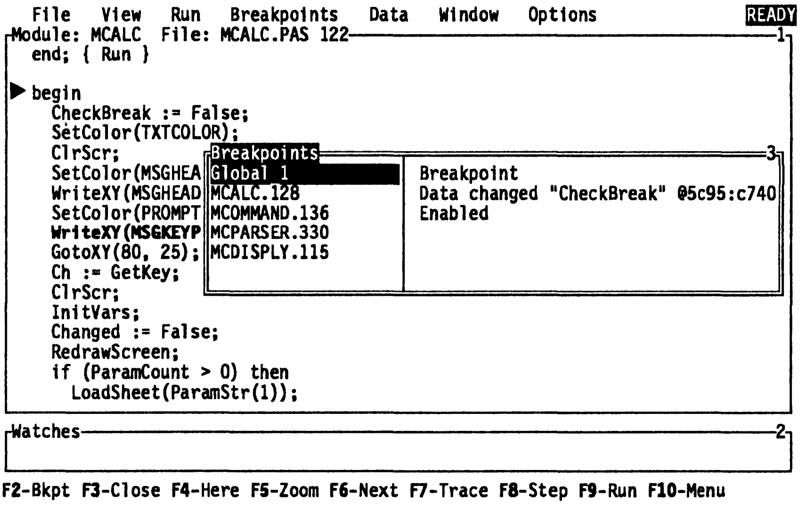
Scope of Breakpoint Expressions

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 at the scope of the current location the program is stopped at. This means that your breakpoint expression can only use 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 create a Breakpoints window by choosing the **View/Breakpoints** main menu command. What this does is give 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.



```
  File   View   Run   Breakpoints   Data   Window   Options                READY
Module: MCALC  File: MCALC.PAS 122                                              1
  end; { Run }

▶ begin
    CheckBreak := False;
    SetColor(TXTCOLOR);
    ClrScr;           Breakpoints                                               3
    SetColor(MSGHEA  Global 1          Breakpoint
    WriteXY(MSGHEAD  MCALC.128         Data changed "CheckBreak" @5c95:c740
    SetColor(PROMPT  MCOMMAND.136      Enabled
    WriteXY(MSGKEYP  MCPARSER.330
    GotoXY(80, 25);  MCDISPLY.115
    Ch := GetKey;
    ClrScr;
    InitVars;
    Changed := False;
    RedrawScreen;
    if (ParamCount > 0) then
      LoadSheet(ParamStr(1));

Watches                                                                         2

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

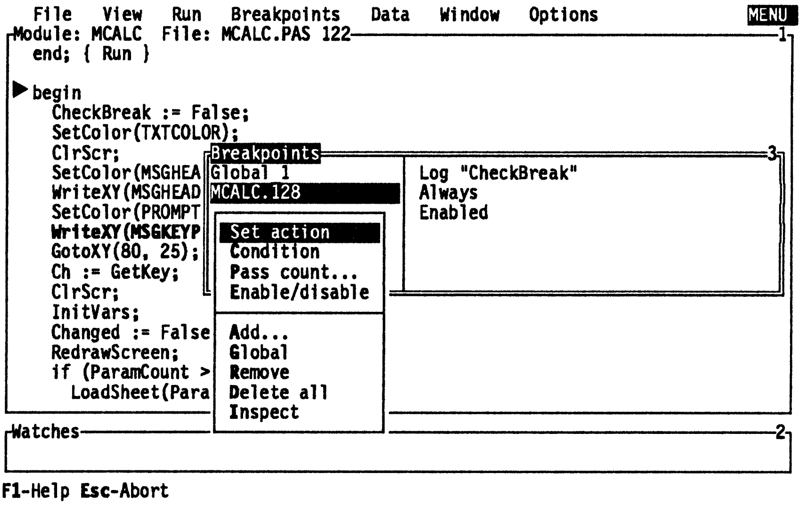
Figure 7.2: The Breakpoints Window

Breakpoint 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 can get to by pressing *Alt-F10.*

The Breakpoints Window Local Menu

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

Alt-F10 pops up the Breakpoint 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.

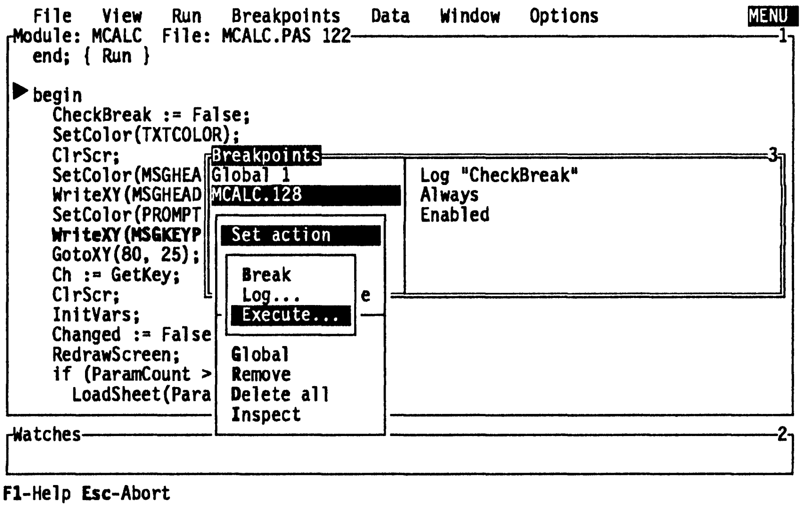


```
 File   View   Run   Breakpoints   Data   Window   Options              MENU
Module: MCALC  File: MCALC.PAS 122─────────────────────────────────────────1
 end; { Run }

▶begin
   CheckBreak := False;
   SetColor(TXTCOLOR);
   ClrScr;         Breakpoints═════════════════════════════════════════════3
   SetColor(MSGHEA Global 1                  Log "CheckBreak"
   WriteXY(MSGHEAD MCALC.128                 Always
   SetColor(PROMPT                           Enabled
   WriteXY(MSGKEYP   Set action
   GotoXY(80, 25);   Condition
   Ch := GetKey;     Pass count...
   ClrScr;           Enable/disable
   InitVars;
   Changed := False  Add...
   RedrawScreen;     Global
   if (ParamCount >  Remove
     LoadSheet(Para  Delete all
                     Inspect
Watches────────────────────────────────────────────────────────────────────2

F1-Help Esc-Abort
```

Figure 7.3: The Breakpoint Window Local Menu

Set Action

Allows you to define what happens when the breakpoint is triggered. This command pops up the menu shown in Figure 7.4.



```
  File  View  Run  Breakpoints  Data  Window  Options                    MENU
Module: MCALC  File: MCALC.PAS 122                                           1
  end; { Run }

► begin
    CheckBreak := False;
    SetColor(TXTCOLOR);
    ClrScr;            Breakpoints                                           3
    SetColor(MSGHEA  Global 1              Log "CheckBreak"
    WriteXY(MSGHEAD  MCALC.128             Always
    SetColor(PROMPT                        Enabled
    WriteXY(MSGKEYP   Set action
    GotoXY(80, 25);
    Ch := GetKey;       Break
    ClrScr;             Log...       e
    InitVars;           Execute...
    Changed := False
    RedrawScreen;      Global
    if (ParamCount >   Remove
      LoadSheet(Para   Delete all
                       Inspect
Watches                                                                      2

F1-Help Esc-Abort
```

Figure 7.4: The Set Action Menu

Break

Causes your program to stop when the breakpoint is triggered. The Turbo Debugger screen will be redisplayed and you can enter commands to look around at your program's data structures.

Log

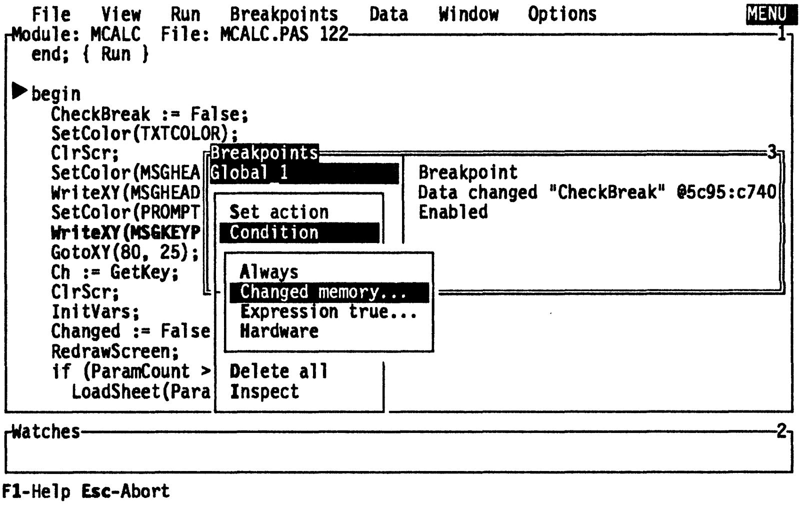
Causes the value of an expression to be recorded in the Log window. You are prompted for the expression whose value you wish to log. Be careful that the expression doesn't have any unexpected side effects. See Chapter 9 for a description of expressions and side effects.

Execute

Causes an expression to be executed. You are prompted for the expression. 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.

Condition

Allows you to control the conditions under which the breakpoint is triggered. This command pops up the menu shown in Figure 7.5.



```
  File   View   Run   Breakpoints   Data   Window   Options             MENU
┌Module: MCALC  File: MCALC.PAS 122──────────────────────────────────────1┐
│ end; { Run }
│
│▶begin
│   CheckBreak := False;
│   SetColor(TXTCOLOR);
│   ClrScr;          ┌Breakpoints════════════════════════════════════3┐
│   SetColor(MSGHEA ║Global 1          ║ Breakpoint
│   WriteXY(MSGHEAD ║┌──────────────┐  ║ Data changed "CheckBreak" @5c95:c740
│   SetColor(PROMPT ║│ Set action    │  ║ Enabled
│   WriteXY(MSGKEYP ║│ Condition     │  ║
│   GotoXY(80, 25); ║│┌─────────────────┐
│   Ch := GetKey;   ║││ Always           │
│   ClrScr;         ╚╡│ Changed memory...│═══════════════════════════
│   InitVars;        ││ Expression true...│
│   Changed := False ││ Hardware          │
│   RedrawScreen;    │└─────────────────┘
│   if (ParamCount > │ Delete all     │
│     LoadSheet(Para │ Inspect        │
└────────────────────┴────────────────┴──────────────────────────────────┘
┌Watches─────────────────────────────────────────────────────────────────2┐
└─────────────────────────────────────────────────────────────────────────┘
F1-Help Esc-Abort
```

Figure 7.5: The Condition Menu

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. You are prompted for an expression referencing the object you wish to watch, and 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 will execute much more slowly, since 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 (*) will appear 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). You are prompted for the expression to evaluate each time the action is encountered.

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.

Appendix F discusses the hardware debugger interface and the options available under this menu.

Pass Count...

Sets the number of times the action must be encountered before it is triggered. The **Pass Count** command 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.

Enable/Disable

Enables or disables the currently highlighted breakpoint. This command acts as a toggle, switching between enabled and disabled each time you use it. A disabled breakpoint is "invisible" until you enable it again; it behaves as if it was deleted.

This command 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. This saves you from having to delete the breakpoint, and then reenter it along with its conditions and action.

Add...

Adds a breakpoint to the list of breakpoints. You are prompted for the address in your program where the breakpoint will occur. If you wish to set a global action that occurs at every program line, use the Global command from this menu.

You can also add a breakpoint by simply starting to type the address at which you want to set it. A prompt box will appear 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.

Global

Adds a global breakpoint to the list of breakpoints. 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 slow down the execution of your program by a large amount. However, they can be very convenient for finding where your program is "bashing" some data.

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

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 when encountering a breakpoint.

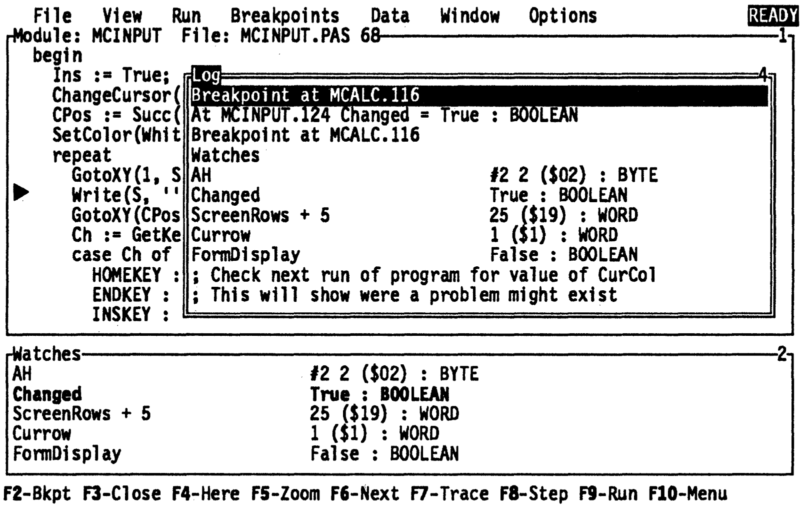
Inspect

Shows you the source code line or assembler instruction corresponding 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 will be opened and set to that line. Otherwise, a CPU window will be 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.



```
  File   View   Run   Breakpoints   Data   Window   Options                READY
Module: MCINPUT  File: MCINPUT.PAS 68-------------------------------------------1
  begin
    Ins := True;  Log==================================================4
    ChangeCursor( Breakpoint at MCALC.116
    CPos := Succ( At MCINPUT.124 Changed = True : BOOLEAN
    SetColor(Whit Breakpoint at MCALC.116
    repeat        Watches
      GotoXY(1, S AH                                  #2 2 ($02) : BYTE
 ▶    Write(S, '' Changed                             True : BOOLEAN
      GotoXY(CPos ScreenRows + 5                      25 ($19) : WORD
      Ch := GetKe Currow                              1 ($1) : WORD
      case Ch of  FormDisplay                         False : BOOLEAN
        HOMEKEY : ; Check next run of program for value of CurCol
        ENDKEY :  ; This will show were a problem might exist
        INSKEY :
Watches------------------------------------------------------------------------2
AH                           #2 2 ($02) : BYTE
Changed                      True : BOOLEAN
ScreenRows + 5               25 ($19) : WORD
Currow                       1 ($1) : WORD
FormDisplay                  False : BOOLEAN
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 7.6: The Log Window

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 installation program (TDINST). You can preserve the entire log, continuously writing it to a disk file, by using the Open Log File local menu command.

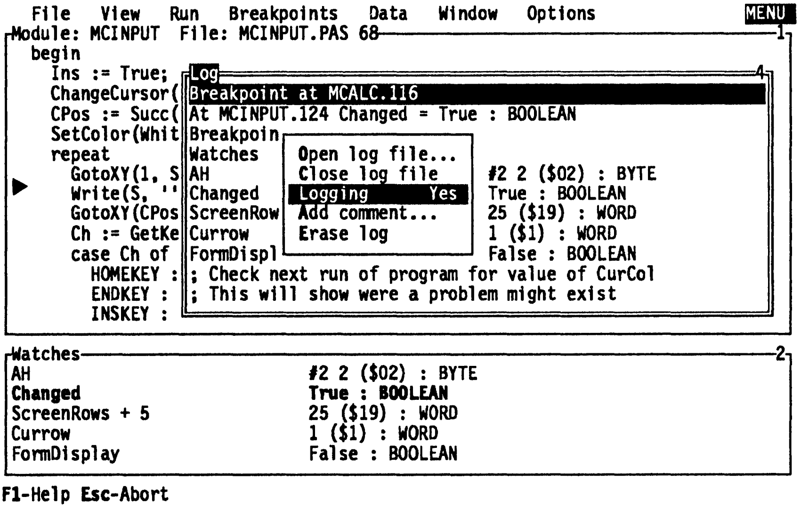
Here's a list of what can cause lines to be written to the log:

- Your program stops at a location you specified. The location it stops at is recorded in the log.
- You issue the Add Comment local menu command. You are prompted for a comment to write to the log.
- A breakpoint is triggered that logs the value of an expression; this value is put in the log.
- You use the Window/Dump Pane To Log command (from the main menu bar) to record the current contents of a pane in a window.

The Log Window Local Menu

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.



```
  File  View  Run  Breakpoints  Data  Window  Options                 MENU
┌Module: MCINPUT  File: MCINPUT.PAS 68──────────────────────────────────1┐
│ begin                                                                   │
│   Ins := True;  ╔Log══════════════════════════════════════════════4╗   │
│   ChangeCursor( ║Breakpoint at MCALC.116                            ║   │
│   CPos := Succ( ║At MCINPUT.124 Changed = True : BOOLEAN            ║   │
│   SetColor(Whit ║Breakpoin┌──────────────────┐                      ║   │
│   repeat        ║Watches  │ Open log file... │                      ║   │
│     GotoXY(1, S ║AH       │ Close log file   │ #2 2 ($02) : BYTE    ║   │
│ ▶   Write(S, '' ║Changed  │ Logging      Yes │ True : BOOLEAN       ║   │
│     GotoXY(CPos ║ScreenRow│ Add comment...   │ 25 ($19) : WORD      ║   │
│     Ch := GetKe ║Currow   │ Erase log        │ 1 ($1) : WORD        ║   │
│     case Ch of  ║FormDispl└──────────────────┘ False : BOOLEAN      ║   │
│       HOMEKEY : ║; Check next run of program for value of CurCol    ║   │
│       ENDKEY :  ║; This will show were a problem might exist        ║   │
│       INSKEY :  ╚═══════════════════════════════════════════════════╝   │
└─────────────────────────────────────────────────────────────────────────┘
┌Watches──────────────────────────────────────────────────────────────────2┐
│AH                           #2 2 ($02) : BYTE                            │
│Changed                      True : BOOLEAN                               │
│ScreenRows + 5               25 ($19) : WORD                              │
│Currow                       1 ($1) : WORD                                │
│FormDisplay                  False : BOOLEAN                              │
└──────────────────────────────────────────────────────────────────────────┘
F1-Help Esc-Abort
```

Figure 7.7: The Log Window Local Menu

Open Log File...

Causes all lines written to the log to also be written to a disk file. You are prompted for the name of the file to write the log to.

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 the Erase Log File command before choosing the Open Log File command.

Close Log File

Stops writing log lines to the file specified in the **Open** log file local menu command, and the file is closed.

Logging

Enables or disables the log, controlling whether anything actually gets written to the Log window.

Add Comment

Allows you to insert a comment into the log. You are prompted for a line of text that may 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 when debugging programs is to 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 action. 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*). Issuing this command 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*). Issuing this command 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.

- Issue the Add local menu command from the breakpoint list pane of the Breakpoint 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 wish 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 local menu command in the Breakpoint window to set the number of times you want to skip the breakpoint before it is actually triggered.

If you wish to stop your program at a specific location but only when a certain condition is true, you can specify an expression using the Condition/Expression True local menu command. 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 Condition/Changed Memory local menu command to specify a breakpoint that only occurs 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 wish to have a breakpoint occur every time a source line or instruction is encountered, you use global breakpoints. There are a number of ways to create a global breakpoint, with each method best-suited for a particular situation:

- Choose the Global local menu command from the action list pane of the Breakpoint window. Use this method when you want to set a qualifying condition and/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 from the main menu bar. Use this to stop when an area of memory changes.
- Choose the **B**reakpoints/**E**xpression True Global command from the main menu bar. Use this command to stop execution when an expression becomes true.

When you set a global breakpoint, you usually use the local menu in the Breakpoint 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.

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 when you restart your program with one of the **R**un commands from the mainmenu bar (or their 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.

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 you can use the **C**ondition/**C**hanged Memory local menu command in the detail pane of the Action window. Enter an expression that refers to the memory area you wish 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 *F10*/**V**iew/**B**reakpoints command. Any breakpoint that is hardware assisted will have an asterisk (*) beside it. These breakpoints will be much faster than other global breakpoints that are not hardware assisted.

Logging Variable Values

Sometimes, you may find it useful to log the value of certain variables each time you reach a certain place in your program. (**Note:** You can only set one breakpoint per address.) 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 for each time the function is called, you can determine when it was called with erroneous parameters.

Issue the Set Action/Log command from the Breakpoints window local menu. You are prompted for the expression whose value is to be logged each time the breakpoint is triggered. If you wish 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.

CHAPTER

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 disk files. First, we show you how to examine and edit program source files, and then we show you how to examine and modify other disk files.

Examining Program Source Files

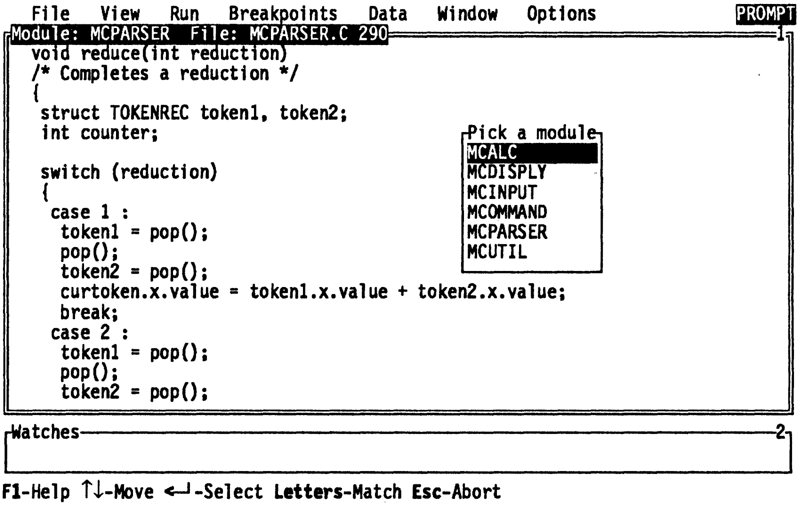
Program source files are your source files that are compiled and that generate an object module (an .EXE file). You usually examine them when you wish to look at the behavior or design of a portion of your code. When debugging, you often need to look at the source code for a function to verify either 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 included in a source file by a compiler directive that generate line #'s (like #include in C and **INCLUDE** in assembler) are also considered to be program source files. 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 set breakpoints or examine program variables simply by moving to the appropriate place in the file. These techniques and others are described in the following section.

The Module Window

You create a Module window by choosing the **View/Module** command from the main menu bar (or press the shortcut *Alt-F3*).


File View Run Breakpoints Data Window Options
PROMPT
Module: MCPARSER File: MCPARSER.C 290
1
Pick a module
MCALC
MCDISPLY
MCINPUT
MCOMMAND
MCPARSER
MCUTIL
Watches
2
F1-Help ↑↓-Move ←┘-Select Letters-Match Esc-Abort

```
  void reduce(int reduction)
  /* Completes a reduction */
  {
   struct TOKENREC token1, token2;
   int counter;

   switch (reduction)
   {
    case 1 :
     token1 = pop();
     pop();
     token2 = pop();
     curtoken.x.value = token1.x.value + token2.x.value;
     break;
    case 2 :
     token1 = pop();
     pop();
     token2 = pop();
```

Figure 8.1: The Module Window

A list of modules will be displayed from which you can pick the module you wish 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 *F10*/**Options/Path for Source** command or the **-sd** command-line option
3. in the current directory
4. in the directory that contains the program you're debugging

Module windows show the contents of the source file for the module you've selected. The title of the Module window shows the name of the module you're viewing, along with the source file name and the line number the cursor is on. An arrow (➤) in the first column of the window shows the current program location.

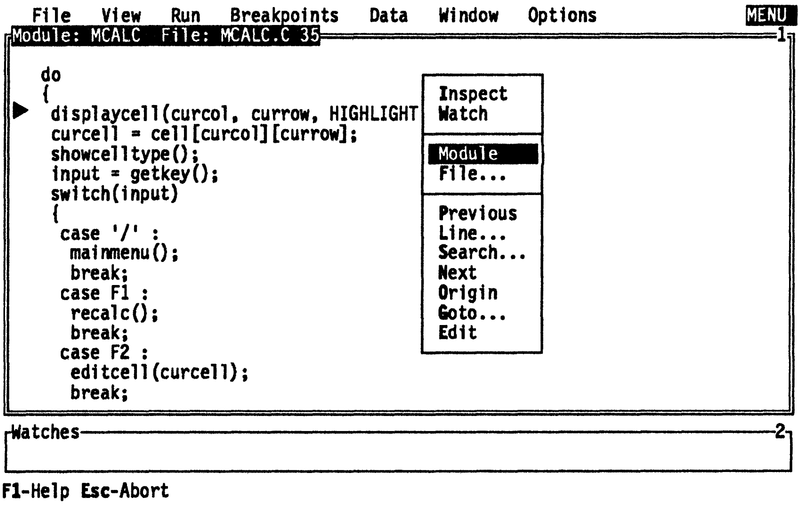
Note that when you run Turbo Debugger, you'll need *both* the .EXE file and the original source file available. Turbo Debugger searches for source files first in the directory the compiler found them in when it compiled, second in the directory specified in the **O**ptions/**P**ath for Source command,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

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 to be displayed on the wrong line.

The Module Window Local Menu

The Module window local menu provides a number of commands that let you move around in the displayed module, point at data items and examine them, and set the window to display a new file or module. You will probably use this menu more than any other menu in the 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.



```
do
{
▶ displaycell(curcol, currow, HIGHLIGHT
 curcell = cell[curcol][currow];
 showcelltype();
 input = getkey();
 switch(input)
 {
  case '/' :
   mainmenu();
   break;
  case F1 :
   recalc();
   break;
  case F2 :
   editcell(curcell);
   break;
```

Figure 8.2: The Module Window Local Menu

Inspect

Opens an inspector to show you the contents of the program variable at the current cursor position. Before issuing this command, you must place the cursor at one of your program variables in the source file.

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 this command 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 continuously monitor the value of a variable as your program executes. Before issuing this command, you must place the cursor at one of your program variables in the source file.

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

Allows you to 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 on the screen.

If you wish to view more than one module simultaneously, use the **View/Another/Module** command from the main menu bar to create another Module window.

File

Allows you to switch to view one of the other source files that makes up the module you are viewing. Pick the file that you wish 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.

If you wish to view more than one file simultaneously, either use the **View/Another/Module** main menu command to create another Module window, or use the **View/File** command to create a File window.

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 prompt 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 prompt. 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 use simple wildcards, with ? indicating a match on any single character, and * matching 0 or more characters. The search does not wrap around from the end of the file to the beginning. To search the entire file, go to the first line by pressing *Ctrl-PgUp*.

Next

Searches for the next instance of the character string you specified with the Search command; you can only use this after issuing a Search command.

Sometimes, a search command matches an unexpected string before reaching the one you really wanted to find. Next lets you repeat the search without having to reenter what you want to search for.

Origin

Positions you at the module and line number that is the current program location (CS:IP). If the module you are currently viewing is not the module that contains the current program location, the Module window will be switched to show that module. This command is useful after you have looked around in your code and want to return to where your program is currently stopped.

Goto

Positions you at any location within your program. Enter the address you wish 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 prompt box exactly as if you had specified the Label command. This is a handy shortcut for this frequently used command.

Edit

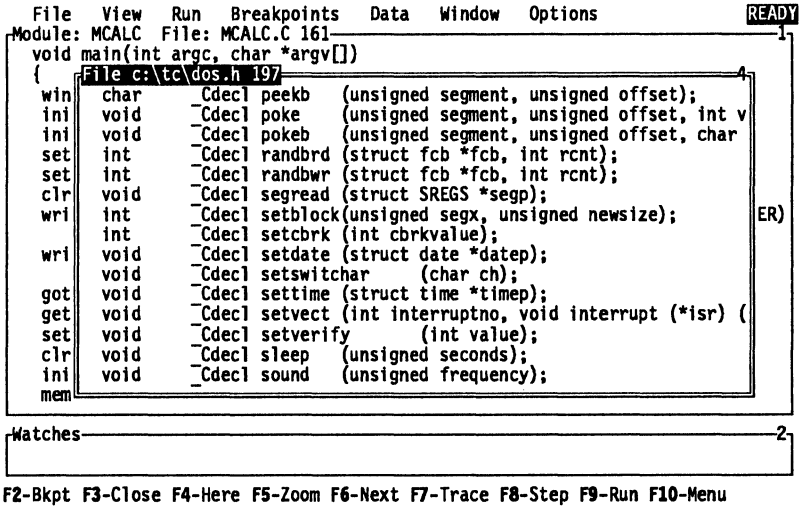
Starts up your choice of an editor so that you can make changes to the source file for the module you are viewing. You can specify the command that starts your editor from the installation program TDINST.

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 Ascii and Hex commands described in later sections of this chapter.

The File Window

You create a File window by choosing the View/File command from the main 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.

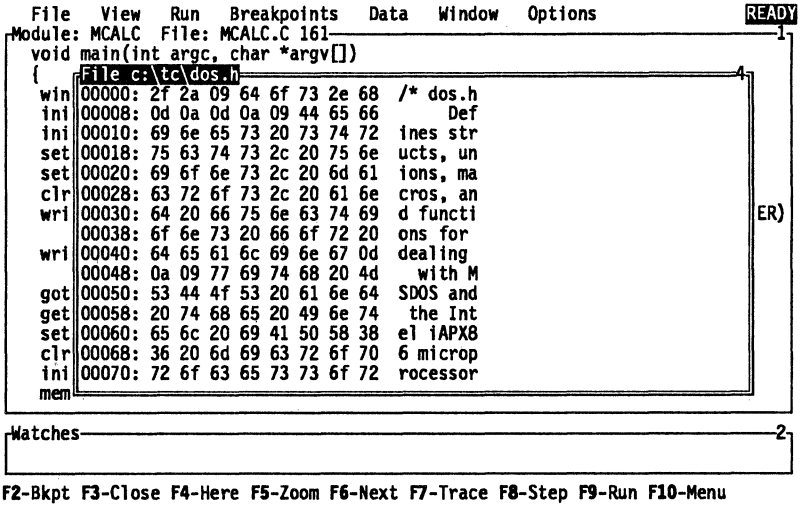


```
  File   View   Run   Breakpoints   Data   Window   Options              READY
 Module: MCALC  File: MCALC.C 161                                           1
  void main(int argc, char *argv[])
  {     File c:\tc\dos.h 197                                            4
   win    char      _Cdecl peekb   (unsigned segment, unsigned offset);
   ini    void      _Cdecl poke    (unsigned segment, unsigned offset, int v
   ini    void      _Cdecl pokeb   (unsigned segment, unsigned offset, char
   set    int       _Cdecl randbrd (struct fcb *fcb, int rcnt);
   set    int       _Cdecl randbwr (struct fcb *fcb, int rcnt);
   clr    void      _Cdecl segread (struct SREGS *segp);
   wri    int       _Cdecl setblock(unsigned segx, unsigned newsize);          ER)
          int       _Cdecl setcbrk (int cbrkvalue);
   wri    void      _Cdecl setdate (struct date *datep);
          void      _Cdecl setswitchar     (char ch);
   got    void      _Cdecl settime (struct time *timep);
   get    void      _Cdecl setvect (int interruptno, void interrupt (*isr) (
   set    void      _Cdecl setverify       (int value);
   clr    void      _Cdecl sleep   (unsigned seconds);
   ini    void      _Cdecl sound   (unsigned frequency);
   mem

 Watches                                                                    2

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

Figure 8.3: 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 winaow, the file will appear either as ASCII text or as hex 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 Display As command described later.



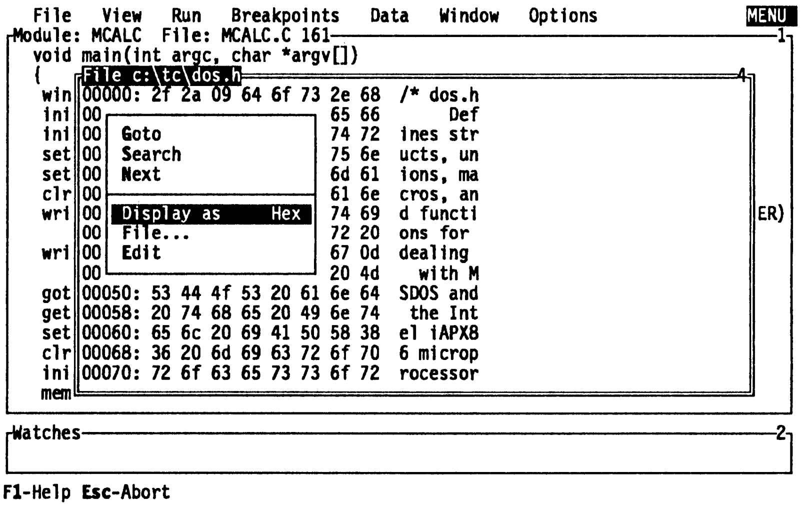
```
  File   View   Run   Breakpoints   Data   Window   Options              READY
┌Module: MCALC  File: MCALC.C 161──────────────────────────────────────────1┐
  void main(int argc, char *argv[])
  {   ┌File c:\tc\dos.h══════════════════════════════════════════════════4┐
   win║00000: 2f 2a 09 64 6f 73 2e 68  /* dos.h                          ║
   ini║00008: 0d 0a 0d 0a 09 44 65 66       Def                          ║
   ini║00010: 69 6e 65 73 20 73 74 72  ines str                          ║
   set║00018: 75 63 74 73 2c 20 75 6e  ucts, un                          ║
   set║00020: 69 6f 6e 73 2c 20 6d 61  ions, ma                          ║
   clr║00028: 63 72 6f 73 2c 20 61 6e  cros, an                          ║
   wri║00030: 64 20 66 75 6e 63 74 69  d functi                          ║ER)
      ║00038: 6f 6e 73 20 66 6f 72 20  ons for                           ║
   wri║00040: 64 65 61 6c 69 6e 67 0d  dealing                           ║
      ║00048: 0a 09 77 69 74 68 20 4d    with M                          ║
   got║00050: 53 44 4f 53 20 61 6e 64  SDOS and                          ║
   get║00058: 20 74 68 65 20 49 6e 74   the Int                          ║
   set║00060: 65 6c 20 69 41 50 58 38  el iAPX8                          ║
   clr║00068: 36 20 6d 69 63 72 6f 70  6 microp                          ║
   ini║00070: 72 6f 63 65 73 73 6f 72  rocessor                          ║
   mem╚═══════════════════════════════════════════════════════════════════╝
└────────────────────────────────────────────────────────────────────────────┘
┌Watches───────────────────────────────────────────────────────────────────2┐
└────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 8.4: 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 is 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.



```
  File  View  Run  Breakpoints  Data  Window  Options                 MENU
┌Module: MCALC  File: MCALC.C 161───────────────────────────────────────1┐
│ void main(int argc, char *argv[])                                      │
│ {  ╔File c:\tc\dos.h═══════════════════════════════════════════4╗    │
│ win║00000: 2f 2a 09 64 6f 73 2e 68  /* dos.h                      ║    │
│ ini║00┌──────────────────┐ 65 66      Def                        ║    │
│ ini║00│ Goto             │ 74 72  ines str                        ║    │
│ set║00│ Search           │ 75 6e  ucts, un                        ║    │
│ set║00│ Next             │ 6d 61  ions, ma                        ║    │
│ clr║00├──────────────────┤ 61 6e  cros, an                        ║    │
│ wri║00│ Display as   Hex │ 74 69  d functi                        ║ER) │
│    ║00│ File...          │ 72 20  ons for                         ║    │
│ wri║00│ Edit             │ 67 0d  dealing                         ║    │
│    ║00└──────────────────┘ 20 4d    with M                        ║    │
│ got║00050: 53 44 4f 53 20 61 6e 64  SDOS and                       ║    │
│ get║00058: 20 74 68 65 20 49 6e 74   the Int                       ║    │
│ set║00060: 65 6c 20 69 41 50 58 38  el iAPX8                       ║    │
│ clr║00068: 36 20 6d 69 63 72 6f 70  6 microp                       ║    │
│ ini║00070: 72 6f 63 65 73 73 6f 72  rocessor                       ║    │
│ mem╚═════════════════════════════════════════════════════════════╝    │
└────────────────────────────────────────────────────────────────────────┘
┌Watches───────────────────────────────────────────────────────────────2┐
│                                                                        │
└────────────────────────────────────────────────────────────────────────┘
F1-Help Esc-Abort
```

Figure 8.5: The File Window Local Menu

Goto

Positions you at a new line number or offset in the file. If you are viewing the file as ASCII text, enter the new line number to go to. If you are viewing the file as hex bytes, enter the offset from the start of the file at which to start displaying. You can use the full expression parser when 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 prompt 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 prompt. 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.

If the file is diplayed in hex, you 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. See Chapter 9 for complete information about byte lists.

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 prompt box exactly as if you had specified the Search command.

Next

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

This is useful when your Search command didn't find the instance of the string you wanted. You can keep issuing this command until you find what you want.

Display As

Toggles between displaying the file as ASCII text or hex bytes. When you select ASCII display, the file appears as you are used to seeing it on the screen in an editor or word processor. If you select Hex display, each line starts with the hex offset into the file of the bytes on the line. Either 8 or 16 bytes of data are displayed on a line, depending on how wide the pane is. 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

Allows you to switch to view 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 on the screen. If you wish to view two different files or two parts of the same file simultaneously, issue the **View/Another/File** 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.

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; Chapter 9 has a complete description of byte lists.

CHAPTER

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, and assembler.

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 response to any prompt that asks for a value or address in memory. (Note that each language evaluates an expression differently.)

You use the **Data/Evaluate/Modify** command from the main menu bar to find the value of an expression you type in. You can also use this command as a simple calculator, as well as to examine the value of data objects in your program.

In this chapter, you'll learn how Turbo Debugger chooses which language to use when 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 accessing 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 Compiler User's Guide and Reference 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 based on the source file-name extension for the current module. This is the module where your program is stopped. You can override this by using the **O**ptions/**L**anguage command to choose **C**, **P**ascal, or **A**ssembler. If you choose the **S**ource option, expressions are evaluated according to the source-file language. (If Turbo Debugger can't determine the source language, it uses C's expression rules.)

Usually, you let Turbo Debugger choose which language to use. Sometimes you may find it useful to explicitly set the language, such as when debugging an assembler module that is called from one of the other languages. By explicitly setting expression evaluation to use a language, you can access your data as you refer to it with that language, even though your current module uses a different language.

Code Addresses, Data Addresses, and Line Numbers

Normally, when you want to access a variable or 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.

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 try and 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, and/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 number 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 number 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 *myfunc2* accessible from routine *myfunc* in module *mymodule*

`#mymodule#file2#myvar2` Variable *myvar2* accessible from *file2* that is included in *mymodule*

`#mymodule#file2#myfunc#myvar2` Variable *myvar2* accessible from *myfunc* defined in file *file2* that is included in *mymodule*

Turbo Debugger also supports Pascal's unit override syntax:

```
unitname.symbolname
```

Implied Scope for Expression Evaluation

Whenever an expression gets evaluated by Turbo Debugger, it must decide where in your program the "current scope" is that is used for any symbol names without an explicit scope override. This is important in many languages because 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. For example, 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 Origin local 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. This includes the Search and Change local commands in the Data pane of the CPU window, as well as the Search and Change local commands of the File window displaying a file in hex format.

A byte list can be any mixture of scalar (non-floating point) numbers and strings, using the syntax of the current language determined by the Options/Language command. Both strings and scalars use the same syntax used in expressions. Scalars are converted into a corresponding byte sequence; for example, a Pascal longint value of 123456 becomes a 4-byte hex quantity 56 34 12 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. An 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 a name for a data item or routine in your program. A symbol name starts with a letter or underscore (_). Subsequent characters in the symbol may contain the digits 0 through 9, as well as these characters. 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. Since the compiler normally puts an underscore at the start of your symbol names, this saves you having to remember to add it.

C Register Pseudovariables

Turbo Debugger lets you access the processor registers using the same technique used by 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	instruction pointer

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 constants.

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 wish 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 may use decimal or scientific notation, for example,

```
1.234 4.5e+11
```

C Character Strings and Escape Sequences

Strings are a sequence of characters enclosed in quote characters (""). You can also use the standard C convention of backslash (\) as an escape character.

Sequence	Value	Character
\\		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
\xnn	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 and Operator Precedence

Turbo Debugger uses the same operators as C, with the same precedence. The debugger has one new operator that is not part of the C language set of operators: the double colon (::). This operator has a higher priority than any of the C operators and 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 greater than the binary operators, grouped from right to left. The binary operators have decreasing priority as indicated next; operators on the same line have the same priority:

<table>
<tr><td><i>highest</i></td><td>* / %</td></tr>
<tr><td></td><td>+ −</td></tr>
<tr><td></td><td>>> <<</td></tr>
<tr><td></td><td>< > <= >=</td></tr>
<tr><td></td><td>== !=</td></tr>
<tr><td></td><td>&</td></tr>
<tr><td></td><td>^</td></tr>
<tr><td></td><td>|</td></tr>
<tr><td></td><td>&&</td></tr>
<tr><td><i>lowest</i></td><td>||</td></tr>
</table>

The single ternary operator, ?:, has a priority less than any of the binary operators.

The assignment operators are all of equal priority below the ternary operator 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.

If your program contains the following function that raises one number to a power (x^y),

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

then the following table shows the result of evaluating 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 described 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 Keywords and Casting

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

Casts are useful if you wish 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 cast 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 cast 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 use the following C keywords when forming casts for Turbo Debugger:

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

Pascal Expressions

Turbo Debugger supports the Pascal expression syntax, with the exception of string concatenation and set operators. An *expression* consists of a mixture of operators, strings, variables, and constants. The following sections describe each of the components that make up an expresion.

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 (0-9) 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 wish to access symbols in other scopes. For more details, see the section "Accessing Symbols outside the Current Scope" on page 138.

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

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 and are listed here in decreasing priority:

```
* / div  mod  and  shl  shr

in + - or  xor

< <= > >= = <>
```

The := assignment operator has the lowest precedence; for your convenience, this 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()      call MyProc
MyProc        reports MyProc's address, etc.
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 *expression* consists of a mixture of 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 0-9, 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 constants. 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
digits	Hexadecimal
digitsO	Octal
digitsQ	Octal
digitsD	Decimal
digitsB	Binary

You must always start a hexadecimal number with one of the digits 0-9. If you want to enter a number that starts with one of the letters *A-F*, you must first precede it with a zero (0).

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. If you wish 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. Note that if you use a format control on the wrong data type, it has no effect.

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 pointer to array of characters as a quoted character string. The string is terminated with a null.
x or h	Displays an integer as a hex number.

CHAPTER

10

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 capabilities described 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'll explain when you might want to use assembler-level debugging. Then we describe the CPU viewer 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

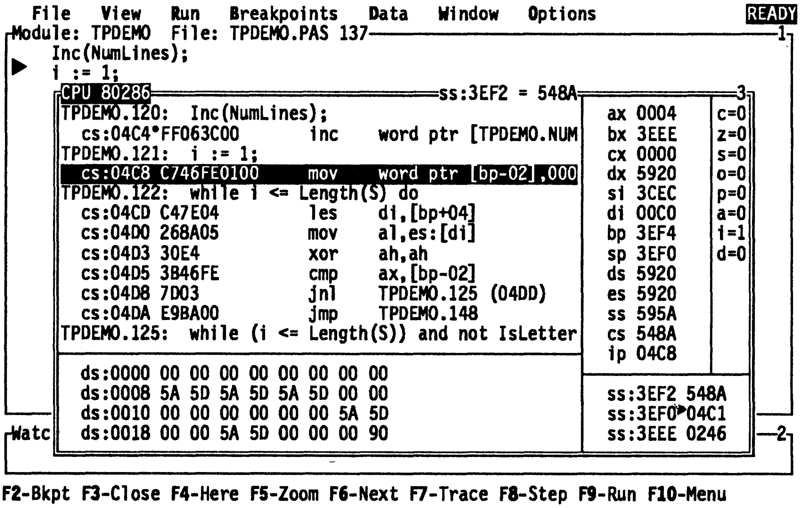
Most of the time when you are debugging a program, you refer to data and code in your program at the source level; you refer to symbol names exactly as you typed them in your source code, and you proceed through your program by executing pieces of source code.

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. Since there are many excellent books available on 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, for example, lets you write lines of source code that perform many actions at once. Since the debugger lets you step one source line at a time, not one C expression at a time, 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 temporarily patch your program 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.



```
 File  View  Run  Breakpoints  Data  Window  Options                  READY
Module: TPDEMO  File: TPDEMO.PAS 137                                       1
   Inc(NumLines);
▶  i := 1;
 CPU 80286                         ss:3EF2 = 548A                        3
 TPDEMO.120:  Inc(NumLines);                             ax 0004  c=0
   cs:04C4•FF063C00       inc    word ptr [TPDEMO.NUM    bx 3EEE  z=0
 TPDEMO.121:  i := 1;                                    cx 0000  s=0
   cs:04C8 C746FE0100     mov    word ptr [bp-02],000    dx 5920  o=0
 TPDEMO.122:  while i <= Length(S) do                    si 3CEC  p=0
   cs:04CD C47E04         les    di,[bp+04]              di 00C0  a=0
   cs:04D0 268A05         mov    al,es:[di]              bp 3EF4  i=1
   cs:04D3 30E4           xor    ah,ah                   sp 3EF0  d=0
   cs:04D5 3B46FE         cmp    ax,[bp-02]              ds 5920
   cs:04D8 7D03           jnl    TPDEMO.125 (04DD)       es 5920
   cs:04DA E9BA00         jmp    TPDEMO.148              ss 595A
 TPDEMO.125:  while (i <= Length(S)) and not IsLetter    cs 548A
                                                         ip 04C8
   ds:0000 00 00 00 00 00 00 00 00
   ds:0008 5A 5D 5A 5D 5A 5D 00 00                       ss:3EF2 548A
   ds:0010 00 00 00 00 00 00 5A 5D                       ss:3EF0▶04C1
Watc ds:0018 00 00 5A 5D 00 00 00 90                     ss:3EEE 0246   2

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

Figure 10.1: The CPU Window

You create a CPU window by choosing the View/CPU command from the main menu bar. Depending on what you are viewing in the current window, the new CPU window will come 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 CPU command:

Current Window	CPU Window Pane	Positioned At
Stack window	Stack	Current SS:SP
Module window	Code	Current CS:IP
Action window	Code	Action address
Variable window	Data*	Address of item
Inspector	Data	Address of item
Breakpoint (if not a Global)	Code	Breakpoint address

*Code pane, if item in window is a routine.

The line at the top of the CPU window shows what processor type you have (8086, 80186, 80286 or 80386). CPU windows have five panes. To go from one pane to the next, press *Tab* or *Shift-Tab*. 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*). You can also directly type over values in the Stack pane.

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. 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

As with all local menus, 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 the desired command.

In the Code, Data, and Stack panes you can press *Ctrl-Left arrow* and *Ctrl-Right arrow* to shift the starting display address of the pane by 1 byte up or down. This is easier than using the Goto command if you just want to slightly adjust the display.

The Code Pane

This pane shows the disassembled instructions at an address that you choose. The Mixed local command 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're debugging an assembler module and you want to see the original source code line, instead of the corresponding disassembled instruction. The pane is set to this display mode if your current module is an assembler source module.

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 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 **Mixed** display mode is set to Yes. Also, if there is a line of source code that corresponds to the symbol address, it is displayed after the symbol.

Global symbols appear simply as the symbol name. Static symbols appear as the module name, followed by a # or a period (.), followed by the static symbol name. Line numbers appear as the module name, followed by a # 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, a word immediate has 4 digits.

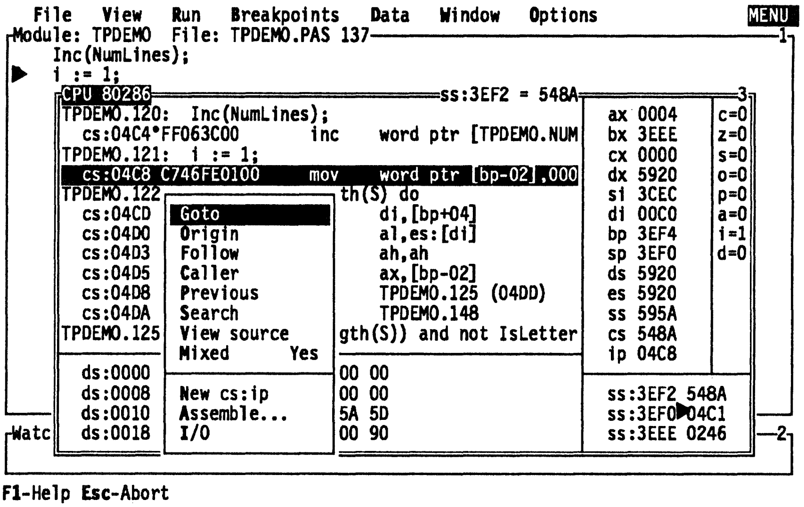
Turbo Debugger can detect the 8087/80287/80386/80387 numeric coprocessor and disassemble those instructions if a floating-point chip or emulator is present.

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.



```
   File   View   Run   Breakpoints   Data   Window   Options                MENU
Module: TPDEMO  File: TPDEMO.PAS 137                                              1
    Inc(NumLines);
▶   i := 1;
  CPU 80286                         ss:3EF2 = 548A                                3
  TPDEMO.120:  Inc(NumLines);                          ax 0004    c=0
    cs:04C4•FF063C00       inc    word ptr [TPDEMO.NUM  bx 3EEE    z=0
  TPDEMO.121:  i := 1;                                 cx 0000    s=0
    cs:04C8 C746FE0100     mov    word ptr [bp-02],000  dx 5920    o=0
  TPDEMO.122                th(S) do                   si 3CEC    p=0
    cs:04CD   Goto                di,[bp+04]           di 00C0    a=0
    cs:04D0   Origin              al,es:[di]           bp 3EF4    i=1
    cs:04D3   Follow              ah,ah                sp 3EF0    d=0
    cs:04D5   Caller              ax,[bp-02]           ds 5920
    cs:04D8   Previous            TPDEMO.125 (04DD)    es 5920
    cs:04DA   Search              TPDEMO.148           ss 595A
  TPDEMO.125  View source   gth(S)) and not IsLetter   cs 548A
              Mixed      Yes                           ip 04C8
    ds:0000                 00 00
    ds:0008   New cs:ip     00 00                      ss:3EF2 548A
    ds:0010   Assemble...   5A 5D                      ss:3EF0▶04C1
Watc ds:0018  I/O           00 90                      ss:3EEE 0246       2

F1-Help Esc-Abort
```

Figure 10.2: The Code Pane 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, which lets you examine code in the BIOS ROM, inside DOS, and in resident utilities. See Chapter 9 for complete information on entering addresses.

The Previous command restores the Code pane to the position it had before the Goto 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 Previous command restores the Code pane to the position it had before the Origin 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 indicated by where the currently highlighted instruction will transfer control to. 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 **Previous** command restores the Code pane to the position it had before the **Follow** 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 **Previous** command restores the Code pane to the position it had before the **Caller** 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 the *PgUp* and *PgDn* keys does not cause the position to be remembered.

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

Search

Lets you enter an instruction or byte list that you want to search for. Enter an instruction exactly as you would when using the **Assemble** 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 where they are assembled in memory. For example, searching for the following instructions is no problem:

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

but trying to search for the following instructions can cause unpredictable results:

```
JE    123
CALL  MYFUNC
LOOP  $-10
```

You can also enter a byte list instead of an instruction. See Chapter 9 for more information 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 line, 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 will start 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.

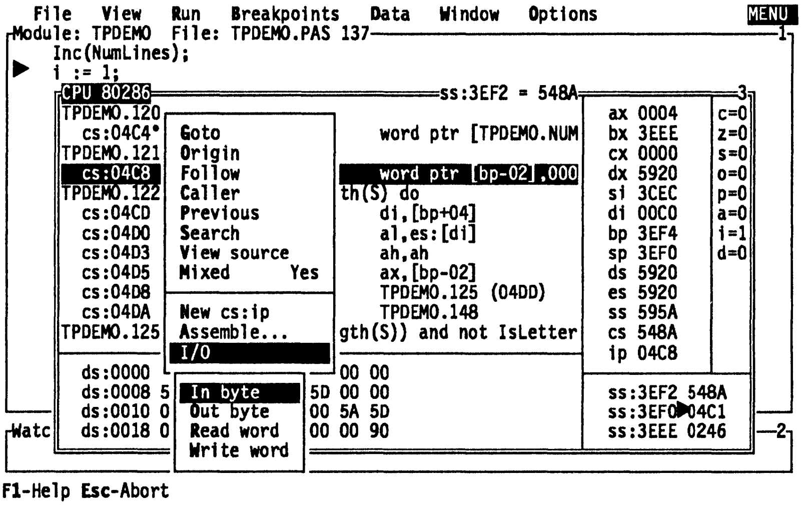
Assemble

Assembles an instruction, replacing the one at the currently highlighted location. You are prompted for the instruction to assemble. See the section in this chapter called "The Assembler" (page 172) 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 prompt box will appear exactly as if you had specified the Assemble command.

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. It pops up the menu shown in Figure 10.3.



```
  File   View   Run   Breakpoints   Data   Window   Options                MENU
 Module: TPDEMO  File: TPDEMO.PAS 137─────────────────────────────────────────1
    Inc(NumLines);
 ►  i := 1;
   CPU 80286══════════════════════════ss:3EF2 = 548A══════════════════════3
   TPDEMO.120                                          ax 0004   c=0
     cs:04C4•  Goto             word ptr [TPDEMO.NUM  bx 3EEE   z=0
   TPDEMO.121  Origin                                 cx 0000   s=0
     cs:04C8   Follow           word ptr [bp-02],000  dx 5920   o=0
   TPDEMO.122  Caller         th(S) do                si 3CEC   p=0
     cs:04CD   Previous         di,[bp+04]            di 00C0   a=0
     cs:04D0   Search           al,es:[di]            bp 3EF4   i=1
     cs:04D3   View source      ah,ah                 sp 3EF0   d=0
     cs:04D5   Mixed      Yes   ax,[bp-02]            ds 5920
     cs:04D8                    TPDEMO.125 (04DD)     es 5920
     cs:04DA   New cs:ip        TPDEMO.148            ss 595A
   TPDEMO.125  Assemble...    gth(S)) and not IsLetter cs 548A
               I/O                                    ip 04C8
     ds:0000               00 00
     ds:0008 5  In byte   5D 00 00                   ss:3EF2 548A
     ds:0010 0  Out byte  00 5A 5D                   ss:3EF0►04C1
 Watc ds:0018 0 Read word 00 00 90                   ss:3EEE 0246        2
                Write word
 F1-Help Esc-Abort
```

Figure 10.3: The I/O Menu

In Byte

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

Write Word

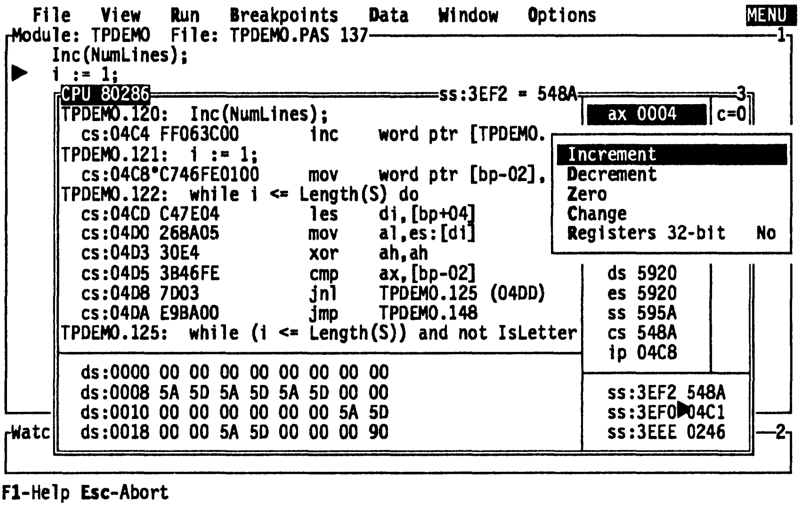
Writes a word to an I/O port. You will be prompted for the I/O port to write to and the value you want to write. Use the **O**ut Byte option to write to a byte-sized I/O port.

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 indiscriminant use of these commands.

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.



```
  File   View   Run   Breakpoints   Data   Window   Options                 MENU
 Module: TPDEMO  File: TPDEMO.PAS 137                                           1
    Inc(NumLines);
 ►  i := 1;
    CPU 80286                            ss:3EF2 = 548A                       3
    TPDEMO.120:  Inc(NumLines);                            ax 0004    c=0
      cs:04C4 FF063C00        inc    word ptr [TPDEMO.
    TPDEMO.121:  i := 1;                                  Increment
      cs:04C8•C746FE0100      mov    word ptr [bp-02],    Decrement
    TPDEMO.122:  while i <= Length(S) do                  Zero
      cs:04CD C47E04          les    di,[bp+04]           Change
      cs:04D0 268A05          mov    al,es:[di]           Registers 32-bit   No
      cs:04D3 30E4            xor    ah,ah
      cs:04D5 3B46FE          cmp    ax,[bp-02]             ds 5920
      cs:04D8 7D03            jnl    TPDEMO.125 (04DD)      es 5920
      cs:04DA E9BA00          jmp    TPDEMO.148             ss 595A
    TPDEMO.125:  while (i <= Length(S)) and not IsLetter    cs 548A
                                                            ip 04C8
      ds:0000 00 00 00 00 00 00 00 00
      ds:0008 5A 5D 5A 5D 5A 5D 00 00                       ss:3EF2 548A
      ds:0010 00 00 00 00 00 00 5A 5D                       ss:3EF0►04C1
 Watc ds:0018 00 00 5A 5D 00 00 00 90                       ss:3EEE 0246     2

F1-Help Esc-Abort
```

Figure 10.4: The Register Pane Local Menu

Increment

Adds one 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 one from the value in the currently highlighted register.

Zero

Sets the value of the currently highlighted register to zero.

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 when entering a new value.

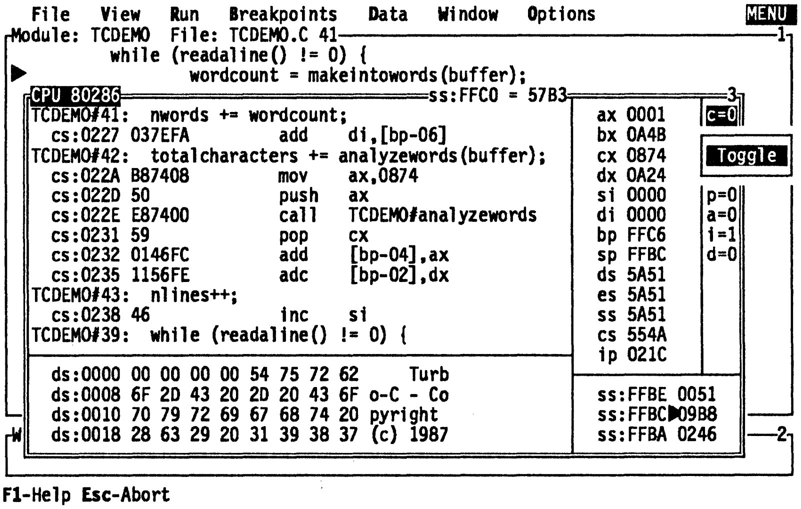
You can also invoke this command by simply starting to type the new value for the register. When you do this, a prompt box will appear exactly as if you had specified the Change command.

Registers 32-bit

Toggles between displaying the CPU registers as 16-bit or 32-bit values. If you are running on an 80386 processor, you will usually see 32-bit registers, unless you use this command to set the display to 16-bit registers. You only really need to see 32-bit registers 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 only uses the normal 16-bit addressing, you can select 16-bit register display.

The Flags Pane Local Menu

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.



```
  File   View   Run   Breakpoints   Data   Window   Options                  MENU
Module: TCDEMO  File: TCDEMO.C 41                                               1
        while (readaline() != 0) {
▶           wordcount = makeintowords(buffer);
 CPU 80286                          ss:FFC0 = 57B3                         3
 TCDEMO#41:  nwords += wordcount;                       ax 0001    c=0
   cs:0227 037EFA         add    di,[bp-06]             bx 0A4B
 TCDEMO#42:  totalcharacters += analyzewords(buffer);   cx 0874     Toggle
   cs:022A B87408         mov    ax,0874                dx 0A24
   cs:022D 50             push   ax                     si 0000    p=0
   cs:022E E87400         call   TCDEMO#analyzewords    di 0000    a=0
   cs:0231 59             pop    cx                     bp FFC6    i=1
   cs:0232 0146FC         add    [bp-04],ax             sp FFBC    d=0
   cs:0235 1156FE         adc    [bp-02],dx             ds 5A51
 TCDEMO#43:  nlines++;                                  es 5A51
   cs:0238 46             inc    si                     ss 5A51
 TCDEMO#39:  while (readaline() != 0) {                 cs 554A
                                                        ip 021C
   ds:0000 00 00 00 00 54 75 72 62     Turb
   ds:0008 6F 2D 43 20 2D 20 43 6F o-C - Co             ss:FFBE 0051
   ds:0010 70 79 72 69 67 68 74 20 pyright              ss:FFBC▶09B8
W  ds:0018 28 63 29 20 31 39 38 37 (c) 1987             ss:FFBA 0246       2

F1-Help Esc-Abort
```

Figure 10.5: The Flags Pane Local Menu

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 mode 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.

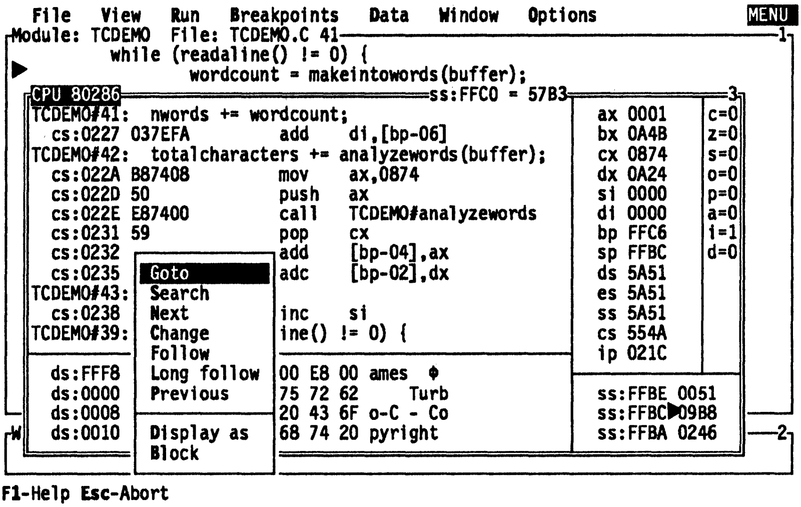
The rightmost part of each line shows the display characters that correspond to the data bytes displayed. Turbo Debugger displays all byte values as their display equivalents, so don't be surprised if you see funny symbols displayed to the right of the hex dump area—these are just the display equivalents of the hex byte values.

The number of bytes displayed on each line varies with the format set with the **D**isplay As command.

Note: 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 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 the command.



```
 File   View   Run   Breakpoints   Data   Window   Options              MENU
Module: TCDEMO  File: TCDEMO.C 41                                          1
          while (readaline() != 0) {
▶                 wordcount = makeintowords(buffer);
 CPU 80286                               ss:FFC0 = 57B3                   3
 TCDEMO#41:  nwords += wordcount;                         ax 0001   c=0
   cs:0227 037EFA          add    di,[bp-06]              bx 0A4B   z=0
 TCDEMO#42:  totalcharacters += analyzewords(buffer);     cx 0874   s=0
   cs:022A B87408          mov    ax,0874                 dx 0A24   o=0
   cs:022D 50              push   ax                      si 0000   p=0
   cs:022E E87400          call   TCDEMO#analyzewords     di 0000   a=0
   cs:0231 59              pop    cx                      bp FFC6   i=1
   cs:0232                 add    [bp-04],ax              sp FFBC   d=0
   cs:0235   Goto          adc    [bp-02],dx              ds 5A51
 TCDEMO#43:  Search                                       es 5A51
   cs:0238   Next          inc    si                      ss 5A51
 TCDEMO#39:  Change        ine() != 0) {                  cs 554A
             Follow                                       ip 021C
   ds:FFF8   Long follow   00 E8 00 ames   Φ
   ds:0000   Previous      75 72 62     Turb              ss:FFBE 0051
   ds:0008                 20 43 6F o-C - Co              ss:FFBC▶09B8
W  ds:0010   Display as    68 74 20 pyright               ss:FFBA 0246   2
             Block
F1-Help Esc-Abort
```

Figure 10.6: The Data Pane Local Menu

Goto

Positions you at an address in your data. Enter the new address you wish 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 Search command.

Change

Allows you to change the bytes at the current cursor location. If you're over an ASCII display or the format is byte, 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 prompt box exactly as if you had chosen the Change command.

Follow

Allows you to 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.

Long Follow

Allows you to 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.

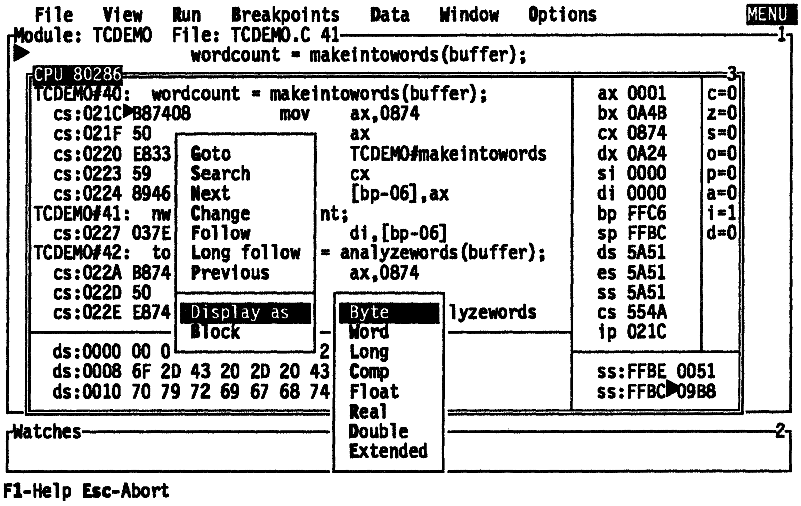
Previous

Restores the Data pane address to the address before the last command that explicitly changed the display address. Using the arrow keys and the *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 Follow, Long Follow, or Goto commands.

Display As

Lets you choose how data appears in the Data pane. You can choose between all the data formats used by C, Pascal, and assembler. You can choose one of the options from the menu shown in Figure 10.7.



```
  File   View   Run   Breakpoints   Data   Window   Options                MENU
Module: TCDEMO  File: TCDEMO.C 41                                               1
▶                    wordcount = makeintowords(buffer);
 CPU 80286                                                                    3
 TCDEMO#40:  wordcount = makeintowords(buffer);            ax 0001     c=0
   cs:021C▶B87408          mov    ax,0874                  bx 0A4B     z=0
   cs:021F 50                     ax                       cx 0874     s=0
   cs:0220 E833  Goto             TCDEMO#makeintowords     dx 0A24     o=0
   cs:0223 59    Search           cx                       si 0000     p=0
   cs:0224 8946  Next             [bp-06],ax               di 0000     a=0
 TCDEMO#41:  nw  Change         nt;                        bp FFC6     i=1
   cs:0227 037E  Follow           di,[bp-06]               sp FFBC     d=0
 TCDEMO#42:  to  Long follow    = analyzewords(buffer);    ds 5A51
   cs:022A B874  Previous         ax,0874                  es 5A51
   cs:022D 50                                              ss 5A51
   cs:022E E874  Display as       Byte      lyzewords      cs 554A
                 Block          - Word                     ip 021C
   ds:0000 00 0                 2 Long
   ds:0008 6F 2D 43 20 2D 20 43   Comp                     ss:FFBE 0051
   ds:0010 70 79 72 69 67 68 74   Float                    ss:FFBC▶09B8
                                  Real
Watches                           Double                                        2
                                  Extended
F1-Help Esc-Abort
```

Figure 10.7: The Display As Menu

Byte

Sets the Data pane to display as hexadecimal bytes.

This corresponds to the C **char** data type and the Pascal byte type.

Word

Sets the Data pane to display as word hexadecimal numbers. The 2-byte hex 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 (IEEE) single 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 and the assembler TBYTE type.

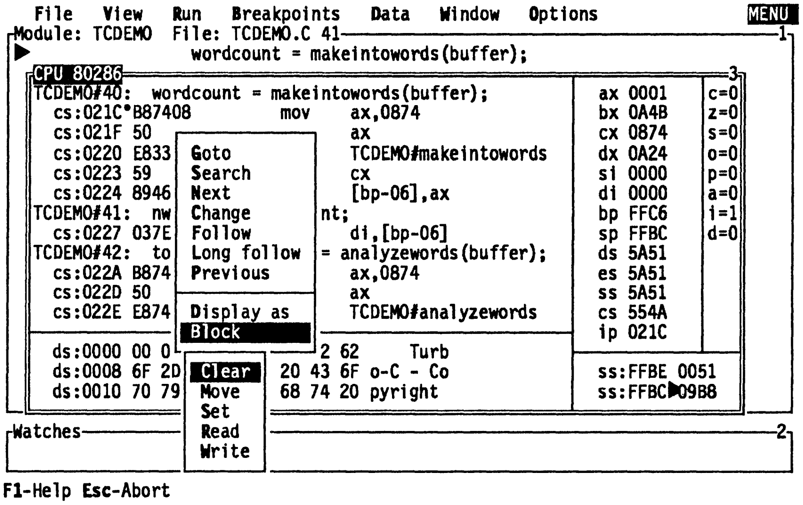
Extended

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 (IEEE) extended 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 in Figure 10.8.



```
  File  View  Run  Breakpoints  Data  Window  Options                    MENU
┌Module: TCDEMO  File: TCDEMO.C 41────────────────────────────────────────────1┐
│▶               wordcount = makeintowords(buffer);                            │
│ ┌CPU 80286═════════════════════════════════════════════════════════════3┐   │
│ │TCDEMO#40:  wordcount = makeintowords(buffer);   │ ax 0001  │c=0│   │
│ │  cs:021C•B87408        mov    ax,0874            │ bx 0A4B  │z=0│   │
│ │  cs:021F 50   ┌─────────────┐ ax                  │ cx 0874  │s=0│   │
│ │  cs:0220 E833 │Goto         │ TCDEMO#makeintowords│ dx 0A24  │o=0│   │
│ │  cs:0223 59   │Search       │ cx                  │ si 0000  │p=0│   │
│ │  cs:0224 8946 │Next         │ [bp-06],ax          │ di 0000  │a=0│   │
│ │TCDEMO#41:  nw │Change       │nt;                  │ bp FFC6  │i=1│   │
│ │  cs:0227 037E │Follow       │ di,[bp-06]          │ sp FFBC  │d=0│   │
│ │TCDEMO#42:  to │Long follow  │= analyzewords(buffer);│ ds 5A51 │   │   │
│ │  cs:022A B874 │Previous     │ ax,0874             │ es 5A51  │   │   │
│ │  cs:022D 50   ├─────────────┤ ax                  │ ss 5A51  │   │   │
│ │  cs:022E E874 │Display as   │ TCDEMO#analyzewords │ cs 554A  │   │   │
│ ├───────────────│Block        │─────────────────────│ ip 021C  │   │   │
│ │  ds:0000 00 0└┌─────┐───────┘2 62    Turb          ├──────────────────┤   │
│ │  ds:0008 6F 2D│Clear│ 20 43 6F o-C - Co          │ ss:FFBE 0051     │   │
│ │  ds:0010 70 79│Move │ 68 74 20 pyright           │ ss:FFBC▶09B8     │   │
└─┴═══════════════│Set  │═══════════════════════════════════════════════┘───┘
┌Watches──────────│Read │─────────────────────────────────────────────────2┐
│                 │Write│                                                  │
└─────────────────└─────┘──────────────────────────────────────────────────┘
F1-Help Esc-Abort
```

Figure 10.8: The Block Menu

Clear

Sets a contiguous block of memory to zero (0). You will be prompted for the address and the number of bytes to clear.

Move

Copies a block of memory from one address to another. You will be 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 will be prompted for the address of the block, how many bytes to set, and the value to set them to.

Read

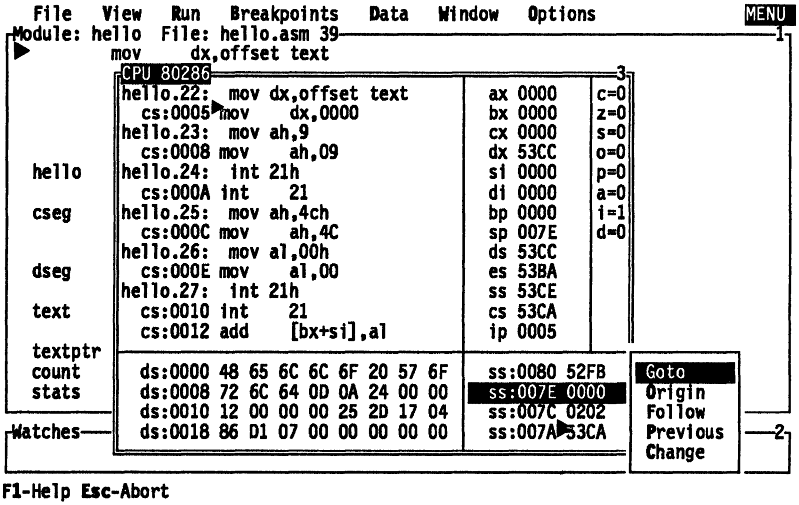
Reads all or a portion of a file into a block of memory. You will be prompted first for the file name to read from, then for the address to read it into, and how many bytes to read.

Write

Writes a block of memory to a file. You will be 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 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.



```
 File   View   Run   Breakpoints   Data   Window   Options                 MENU
Module: hello  File: hello.asm 39                                              1
▶         mov     dx,offset text
         CPU 80286                                         3
         hello.22:  mov dx,offset text           ax 0000    c=0
           cs:0005▶mov    dx,0000                bx 0000    z=0
         hello.23:  mov ah,9                     cx 0000    s=0
           cs:0008 mov    ah,09                  dx 53CC    o=0
 hello   hello.24:  int 21h                      si 0000    p=0
           cs:000A int    21                     di 0000    a=0
 cseg    hello.25:  mov ah,4ch                   bp 0000    i=1
           cs:000C mov    ah,4C                  sp 007E    d=0
         hello.26:  mov al,00h                   ds 53CC
 dseg      cs:000E mov    al,00                  es 53BA
         hello.27:  int 21h                      ss 53CE
 text      cs:0010 int    21                     cs 53CA
           cs:0012 add    [bx+si],al             ip 0005
 textptr
 count     ds:0000 48 65 6C 6C 6F 20 57 6F       ss:0080 52FB    Goto
 stats     ds:0008 72 6C 64 0D 0A 24 00 00       ss:007E 0000    Origin
           ds:0010 12 00 00 00 25 2D 17 04       ss:007C 0202    Follow
Watches    ds:0018 86 D1 07 00 00 00 00 00       ss:007A▶53CA    Previous     2
                                                                 Change
F1-Help Esc-Abort
```

Figure 10.9: The Stack Pane Local Menu

Goto

Positions you at an address in the stack. Enter the new stack address. If you wish, 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 **Previous** command restores the Stack pane to the position it had before the **Goto** 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 **Previous** command restores the Stack pane to the position it had before the **Origin** 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 **Previous** command restores the Stack pane to the position it had before the **Follow** 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 the *PgUp* and *PgDn* keys does not cause the position to be remembered.

Repeated use of the **Previous** 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. When you do this, a prompt box will appear exactly as if you had specified the **Change** command.

The Assembler

Turbo Debugger lets you assemble instructions for the 8086, 80186, and 80286 processor 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 Run/**Program Reset command, or if you load another program using the File/Load 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 go and change your source code and recompile and link your program.

The following section 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**, etc.) 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 just use 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. If your program contained the data definition

```
A    DW 4
```

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 may use the following overrides when assembling 8087/80287 numeric processor 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 raw data dump of any area of memory. It works exactly like the Data pane in the CPU window.

See "The Data Pane Local Menu" section earlier in this chapter (page 165) 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. You can use the View/Dump command to make 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 that 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.

See "The Register Pane Local Menu" (page 162) and "The Flags Pane Local Menu" (page 164) 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. You can shrink the size of your Module window and put up a Registers window alongside it.

Turbo C Code Generation

The Turbo C compiler does a number of predictable things when generating machine code. Once you get 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, first using the SI register, then using the DI register.

Your auto-variables 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

CHAPTER

11

The 80x87 Coprocessor Chip and Emulator

If your program uses floating-point numbers, Turbo Debugger allows you to examine and change the state of the math coprocessor or software emulator. This chapter is for programmers who are familiar with the operation of the 80x87 math coprocessor. 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'll discuss the differences between the 80x87 chip and the software emulator. We'll also describe the Numeric Processor window and teach you how to examine and modify the floating-point registers, the status bits, and the control bits.

The 80x87 Chip vs. Emulator

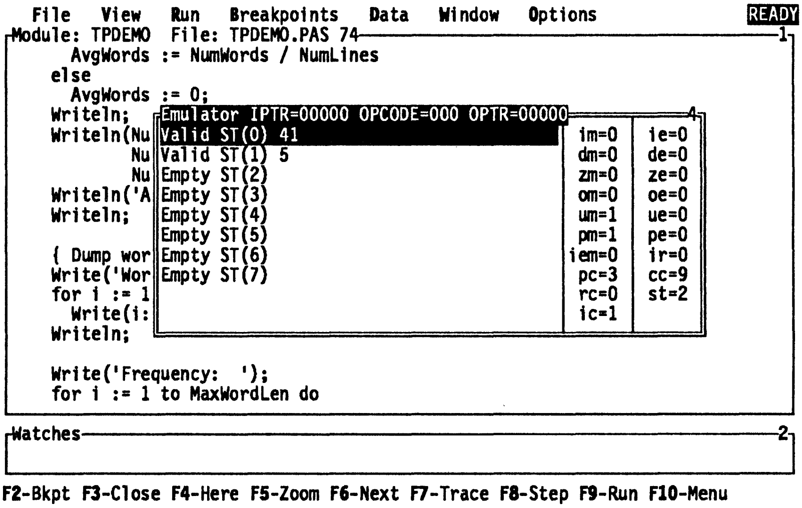
Turbo Debugger automatically detects whether your program is using the math chip or the emulator and adjusts its behavior accordingly.

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 View/Numeric Processor command from the main 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 wish 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.



```
  File   View   Run   Breakpoints   Data   Window   Options               READY
┌Module: TPDEMO  File: TPDEMO.PAS 74─────────────────────────────────────────────1┐
│    AvgWords := NumWords / NumLines
│  else
│    AvgWords := 0;
│  Writeln;   ┌Emulator IPTR=00000 OPCODE=000 OPTR=00000──────────────4┐
│  Writeln(Nu│Valid ST(0) 41                         │  im=0  │ ie=0  │
│          Nu│Valid ST(1) 5                          │  dm=0  │ de=0  │
│          Nu│Empty ST(2)                            │  zm=0  │ ze=0  │
│  Writeln('A│Empty ST(3)                            │  om=0  │ oe=0  │
│  Writeln;  │Empty ST(4)                            │  um=1  │ ue=0  │
│            │Empty ST(5)                            │  pm=1  │ pe=0  │
│  { Dump wor│Empty ST(6)                            │ iem=0  │ ir=0  │
│  Write('Wor│Empty ST(7)                            │  pc=3  │ cc=9  │
│  for i := 1│                                       │  rc=0  │ st=2  │
│    Write(i:│                                       │  ic=1  │       │
│  Writeln;  └───────────────────────────────────────────────────────┘
│
│  Write('Frequency:  ');
│  for i := 1 to MaxWordLen do
└─────────────────────────────────────────────────────────────────────────────────┘
┌Watches───────────────────────────────────────────────────────────────────────────2┐
└─────────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 11.1: The Numeric Processor Window

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 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 Window/Move/Resize, you will also see the floating-point registers displayed as raw hex bytes.

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

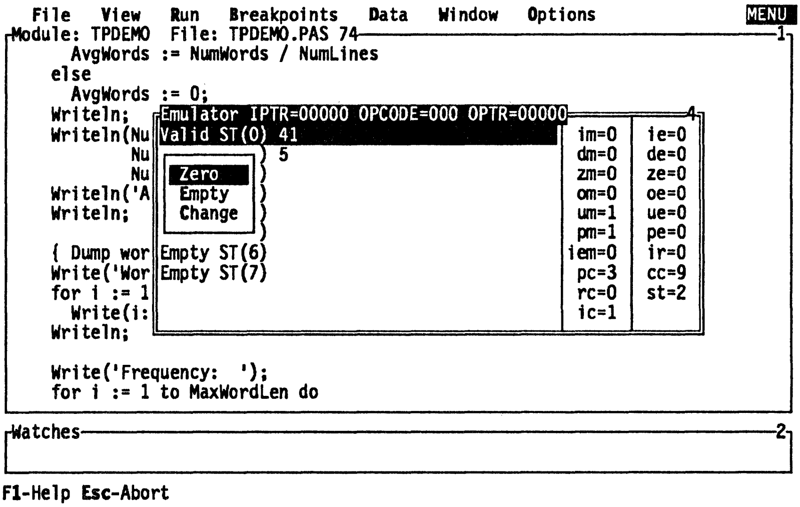
The Control Bits

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 Register Pane Local Menu

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.



```
  File   View   Run   Breakpoints   Data   Window   Options                MENU
Module: TPDEMO  File: TPDEMO.PAS 74                                           1
      AvgWords := NumWords / NumLines
    else
      AvgWords := 0;
    Writeln;   Emulator IPTR=00000 OPCODE=000 OPTR=00000                   4
    Writeln(Nu Valid ST(0) 41                          im=0    ie=0
            Nu             ) 5                          dm=0    de=0
            Nu  Zero       )                            zm=0    ze=0
    Writeln('A  Empty      )                            om=0    oe=0
    Writeln;    Change     )                            um=1    ue=0
                           )                            pm=1    pe=0
    { Dump wor Empty ST(6)                             iem=0    ir=0
    Write('Wor Empty ST(7)                              pc=3    cc=9
    for i := 1                                          rc=0    st=2
      Write(i:                                          ic=1
    Writeln;

    Write('Frequency:  ');
    for i := 1 to MaxWordLen do
Watches                                                                       2

F1-Help Esc-Abort
```

Figure 11.2: The Register Pane Local Menu

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.

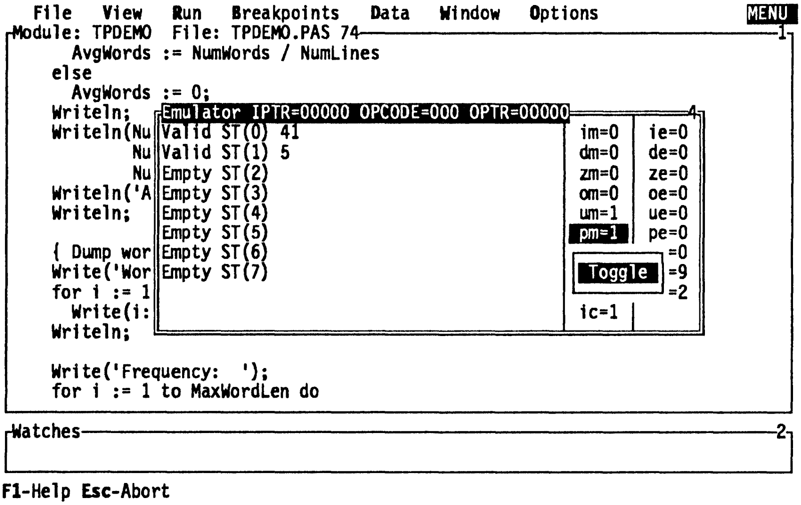
Change

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 full C expression parser. The value you enter will automatically be converted to the 80-bit temporary real format used by the numeric processor.

You can also invoke this command by simply starting to type the new value for the floating-point register. When you do this, a prompt box will appear exactly as if you had specified the Change command.

The Status Pane Local Menu

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 directly access the command.)



```
  File   View   Run   Breakpoints   Data   Window   Options                 MENU
Module: TPDEMO  File: TPDEMO.PAS 74                                            1
    AvgWords := NumWords / NumLines
  else
    AvgWords := 0;
  Writeln;  Emulator IPTR=00000 OPCODE=000 OPTR=00000                 4
  Writeln(Nu Valid ST(0) 41                           im=0   ie=0
          Nu Valid ST(1) 5                            dm=0   de=0
          Nu Empty ST(2)                              zm=0   ze=0
  Writeln('A Empty ST(3)                              om=0   oe=0
  Writeln;   Empty ST(4)                              um=1   ue=0
             Empty ST(5)                              pm=1   pe=0
  { Dump wor Empty ST(6)                                       =0
  Write('Wor Empty ST(7)                              Toggle   =9
  for i := 1                                                   =2
    Write(i:                                          ic=1
  Writeln;

  Write('Frequency:  ');
  for i := 1 to MaxWordLen do

Watches                                                                        2

F1-Help Esc-Abort
```

Figure 11.3: The Status Pane Local Menu

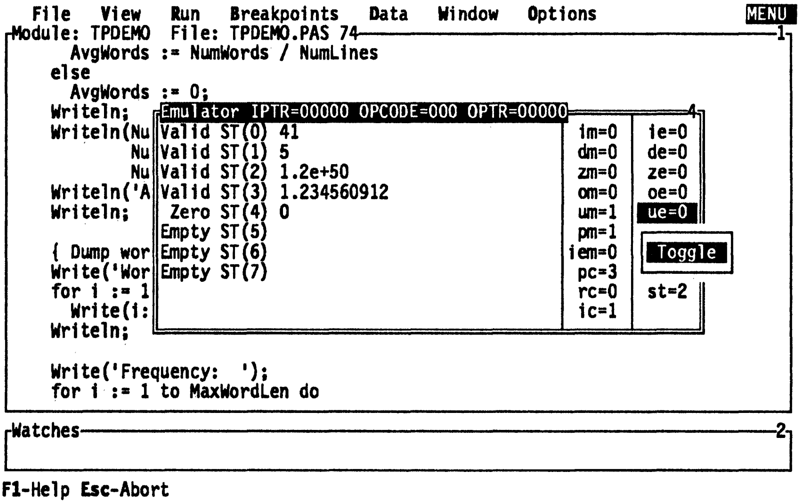
Toggle

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 it sets it back to zero.

You can also toggle the status flag values by pressing *Enter.*

The Control Pane Local Menu

Again, press *Shift-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 directly access the command.)



```
  File   View   Run   Breakpoints   Data   Window   Options              MENU
┌Module: TPDEMO  File: TPDEMO.PAS 74──────────────────────────────────────1┐
│     AvgWords := NumWords / NumLines                                      │
│   else                                                                   │
│     AvgWords := 0;                                                       │
│   Writeln;  ┌Emulator IPTR=00000 OPCODE=000 OPTR=00000═══════════4┐      │
│   Writeln(Nu│Valid ST(0) 41                    │ im=0 │ ie=0 ║      │
│           Nu│Valid ST(1) 5                     │ dm=0 │ de=0 ║      │
│           Nu│Valid ST(2) 1.2e+50               │ zm=0 │ ze=0 ║      │
│   Writeln('A│Valid ST(3) 1.234560912           │ om=0 │ oe=0 ║      │
│   Writeln;  │ Zero ST(4) 0                     │ um=1 │ ue=0 ║      │
│             │Empty ST(5)                       │ pm=1 │┌──────┐     │
│   { Dump wor│Empty ST(6)                       │iem=0 ││Toggle│     │
│   Write('Wor│Empty ST(7)                       │ pc=3 │└──────┘     │
│   for i := 1│                                  │ rc=0 │ st=2 ║      │
│     Write(i:│                                  │ ic=1 │      ║      │
│   Writeln;  └══════════════════════════════════════════════════┘      │
│                                                                          │
│   Write('Frequency:  ');                                                 │
│   for i := 1 to MaxWordLen do                                            │
└──────────────────────────────────────────────────────────────────────────┘
┌Watches──────────────────────────────────────────────────────────────────2┐
│                                                                          │
└──────────────────────────────────────────────────────────────────────────┘
F1-Help Esc-Abort
```

Figure 11.4: The Control Pane Local Menu

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 pressing *Enter.*

CHAPTER

12

Command Reference

Now that you've read about all the commands, here's a quick summary. We'll list and describe

- all the single-keystroke commands available on the function and other keys
- all the main menu commands and the commands for the local menu of each window type
- keystrokes used in the two types of panes, when responding to a prompt for text, and when responding to a prompt for a new window size and position

Hot Keys

A hot key is a key that performs its action no matter where you are in the Turbo Debugger environment. Table 12.1 on page 184 lists all the hot keys.

Table 12.1: The Function Key and Hot Key Commands

Key	Menu Command	Function
F1		Brings up context-sensitive help
F2	Breakpoints/Toggle	Sets breakpoint at cursor position
F3	Window/Close	Closes current window
F4	Run/Go to Cursor	Runs to cursor position
F5		Zooms/unzooms current window
F6	Window/Next	Goes to next window
F7	Run/Trace Into	Executes single source line or instruction
F8	Run/Step Over	Executes single source line or instruction, skipping calls
F9	Run/Run	Runs program
F10		Invokes the main menu bar, takes you out of menus
Alt-F1		Brings up last help screen
Alt-F2	Breakpoints/At	Sets breakpoint at an address
Alt-F3	View/Module	Module pick list
Alt-F4	Run/Animate	Steps continuously updating display
Alt-F5	View/User Screen	Shows your program's screen
Alt-F6	Window/Undo Close	Reopens the last-closed window
Alt-F7	Run/Instruction Trace	Executes a single instruction
Alt-F8	Run/Until Return	Runs until return from function
Alt-F9	Run/Run To	Runs to a specified address
Alt-F10		Invokes the window local menu
Alt-0	Window/Window Pick	Displays a list of all open windows
Alt-1-9		Switch to numbered window
Alt-F		Takes you to the File menu
Alt-V		Takes you to the View menu
Alt-R		Takes you to the Run menu
Alt-B		Takes you to the Breakpoints menu
Alt-D		Takes you to the Data menu
Alt-W		Takes you to the Window menu
Alt-O		Takes you to the Options menu
Alt-X		Quits Turbo Debugger and returns you to DOS
Alt-=	Options/Macros/Create	Defines a keystroke macro
Alt--	Options/Macros/Stop Recording	Ends a macro recording
Ctrl-F2	Run/Program Reset	Stops debug session and resets the program to start again
Ctrl-F4	Data/Evaluate	Evaluates an expression
Ctrl-F7	Data/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-Right arrow		Shifts the starting address in a Code, Data, or Stack pane in a CPU window 1 byte up
Ctrl-Left arrow		Shifts the starting address in a Code, Data, or Stack pane in a CPU window 1 byte down
Ctrl-S		Moves left one column
Ctrl-D		Moves right one column
Ctrl-E		Moves up one line
Ctrl-X		Moves down one line
Ctrl-R		Scrolls up one screen
Ctrl-C		Scrolls down one screen
Ctrl-F		Moves to next word

Table 12.1: The Function Key and Hot Key Commands (continued)

Ctrl-A	Moves to previous word
Esc	Closes an Inspector window, takes you out of menus
Ins	Starts text block selection (highlight); use *Left arrow* and *Right arrow* to highlight
Scroll-Lock Window/Move/Resize	Moves and resizes windows
Tab	Moves cursor to next window pane
Shift-Tab	Moves cursor to previous window pane
Shift-Arrow key	Moves cursor between the panes in a window. The pane in the direction of the arrow becomes the active pane.

Commands from the Main Menu Bar

You invoke the main menu bar by pressing the *F10* key; you can also go directly to one of the individual menus by cursoring to the menu title and pressing *Enter* or by pressing the first 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 File Menu

Load	Loads a new program to debug
Change Dir	Changes to new disk and/or directory
Get Info	Displays program info
OS Shell	Starts a DOS command processor
Quit	Returns to DOS

The View Menu

Breakpoints	View breakpoints
Stack	View function-calling stack
Log	View log of events and data
Watches	View variables being watched
Variables	View global and local variables
Module	View program source module
File	View disk file as ASCII or hex
CPU	View CPU instructions, data, stack
Dump	View raw data dump
Registers	View CPU registers and flags
Numeric Processor	View coprocessor or emulator

User Screen	View your program screen
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
Program Reset	Reloads current program
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
Instruction Trace	Executes a single instruction

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
Watch	Adds variable to Watches window
Function Return	Inspects current routine's return value

The Window Menu

Window Pick	Pick window from list of open windows
Next Pane	Goes to next pane in window
Move/Resize	Moves or changes current window size
Close	Erases current window
Undo Close	Undoes last erase command

Dump Pane to Log	Writes current pane to Log window
Restore Standard	Standard window layout
Screen Repaint	Redisplays entire screen

The Options Menu

Language	
Source Module	Sets expression language from source module
C	Uses C for expressions
Pascal	Uses Pascal for expressions
Assembler	Uses assembler for expressions
Macros	
Create	Defines a keystroke macro
Stop Recording	Ends the recording session
Remove	Removes a keystroke macro
Delete All	Removes all keystroke macros
Environment	
Integer Format	Hex/Decimal/Both: Number display format
Display Swapping	None/Smart/Always: User screen swapping mode
Screen Size	25 line/43/50 line: Debugger screen size
Tab Size	Tab width when displaying text files
Path for Source	Directory list for source files
Arguments	Sets program command-line arguments
Save Options	Saves options, macros, windows to disk
Restore Options	Restores options from disk

The Local Menu Commands

You can invoke the pop-up, or "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. (You can use the installation program TDINST to enable Control-key shortcuts.)

Each type of window (Breakpoint, Module, etc.) and each pane within a window has a different local menu. 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 prompt box for one of the local menu items pops up to accept your input.

The Breakpoints Window Local Menu

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 Action			
	Break		Sets breakpoint to stop program
	Log		Sets breakpoint to log an expression
	Execute		Sets breakpoint to execute an expression
Condition			
	Always		Unconditional breakpoint
	Changed Memory		When memory area changes
	Expression True		When an expression is true
Hardware			
	Cycle Type		
		Read Memory	Match memory reads
		Write Memory	Match memory writes
		Access Memory	Match memory read or write
		Input I/O	Match I/O input
		Output I/O	Match I/O Output
		Both I/O	Match I/O input or output
		Fetch Instruction	Match instruction fetch
	Address		
		Above	Match above an address
		Below	Match below an address
		Range	Match within address range
		Not Range	Match outside address range
		Less or Equal	Match below or equal to address
		Greater or Equal	Match above or equal to address
		Equal	Match a single address
		Unequal	Match all but a single address
		Match All	Match any address
Data			
		Above	Match above a value
		Below	Match below a value
		Range	Match within a range of values
		Not Range	Match outside a range of values
		Less or Equal	Match below or equal to value
		Greater or Equal	Match above or equal to value

Equal	Match a single value
Unequal	Match all but a single value
Match All	Match all values
Pass Count	Number of times to skip breakpoint
Enable/Disable	Toggles breakpoint enabled
Add	Adds a new breakpoint
Global	Adds a new global breakpoint
Remove	Removes highlighted breakpoint
Delete All	Deletes all breakpoints
Inspect	Looks at code where this breakpoint is set

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.

The Code Pane Local Menu

Goto	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	No/Yes/Both: Mixes source code with disassembly
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 **Remove** local menu command in this pane.

The Data Pane Local Menu

Goto	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	Follows near pointer chain
Long Follow	Follows far pointer chain
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 numbers (Pascal singles)
Real	Displays 6-byte floating-point numbers (Pascal reals)
Double	Displays 8-byte floating-point numbers
Extended	Displays 10-byte floating-point numbers (C long double)
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 Change local menu command in this pane.

The Stack Pane Local Menu

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 Change local menu command in this pane.

The Register Pane Local Menu

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	No/Yes: Toggles 32-bit register display

Typing any character is a shortcut for the Change local menu command in this pane.

The Flags Pane Local Menu

Toggle	Sets or clears highlighted flag

Pressing *Enter* is a shortcut for the local menu command in this pane.

The File Window Menu

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	Ascii/Hex: Set file display mode
File	Switches to view new file
Edit	Edits file or changed bytes at cursor

Typing any character is a shortcut for the Search local menu command.

The 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	No/Yes: Toggles logging
Add Comment	Writes user comment to log
Erase Log	Clears all log messages

Typing any character is a shortcut for the Add Comment local menu command.

The Module Window Menu

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 line number 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
Edit	Starts editor to edit source file

Typing any character is a shortcut for the Goto local menu command.

The Numeric Processor Window Menus

The Numeric Processor window has three panes: The Register pane, the Status pane and the Control pane.

The Register Pane Local Menu

The following keys are shortcuts to 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 Change local menu command in this pane

The Status Pane Local Menu

The following keys are shortcuts to local menu commands in this pane:

Toggle	Cycles through valid flag values

Pressing *Enter* is a shortcut for the local menu command in this pane.

The Control Pane Local Menu

Toggle	Cycles through valid flag values

Pressing *Enter* is a shortcut for the local menu command in this pane.

The Stack Window Menu

The Stack window shows the currently active functions.

Inspect	Shows source code for highlighted function
Locals	Shows argument types for function

Pressing *Enter* is a shortcut for the Inspect local menu command.

The Variables Window Menus

The Variables window has two panes, each with a local menu: The Global Symbol pane and the Local Symbol pane.

The Global Symbol Pane Local Menu

Inspect	Shows contents of highlighted symbol
Change	Changes value of highlighted symbol

Pressing *Enter* is a shortcut for the Inspect local menu command in this pane.

The Local Symbol Pane Local Menu

Inspect	Shows contents of highlighted symbol
Change	Changes value of highlighted symbol

Pressing *Enter* is a shortcut for the Inspect local menu command in this pane.

The Watches Window Menu

The Watches window has a single pane that shows the names and values of the variables you're watching.

Watch	Adds a variable to watch
Edit	Lets you edit a variable
Remove	Deletes highlighted variable
Delete All	Deletes all watch variables
Inspect	Shows contents of highlighted variable
Change	Changes contents of highlighted variable

The following keys are shortcuts to local menu commands in this window:

any character	Watch
Enter	Watch

The Inspector Window Local Menu

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 for highlighted item
Descend	Expands highlighted item into this Inspector
New Expression	Inspects a new expression in this Inspector

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.

Key	Function
Ins	Marks text block
Up arrow	Moves up one line
Down arrow	Moves down one line
Right arrow	Moves right one column
Left arrow	Moves left one column
Ctrl-Right arrow	Moves to next word
Ctrl-Left arrow	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:

Key	Function
Ctrl-S	Moves left one column
Ctrl-D	Moves right one column
Ctrl-E	Moves up one line
Ctrl-X	Moves down one line
Ctrl-R	Scrolls up one screen
Ctrl-C	Scrolls down one screen
Ctrl-F	Moves to next word
Ctrl-A	Moves to previous word

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.

Key	Function
Up arrow	Moves up one item
Down arrow	Moves down one item
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	Incremental search (select by typing)

You can also use the WordStar-style Control keys for moving around a List pane:

Key	Function
Ctrl-E	Moves up one line
Ctrl-X	Moves down one line
Ctrl-R	Scrolls up one screen
Ctrl-C	Scrolls down one screen

Commands in Prompt Boxes

The following table shows the commands available when you're inside a prompt box.

Key	Function
Up arrow	Moves up one history item
Down arrow	Moves down one history item
Right arrow	Moves right one character
Left arrow	Moves left one character
Ctrl-Right arrow	Moves to next word
Ctrl-Left arrow	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 after the cursor
Esc	Cancels the prompt and returns to menu

Window Movement Commands

Key	Function
Scroll Lock	Toggles window-positioning mode
Up arrow	Moves window up one line
Down arrow	Moves window down one line
Right arrow	Moves window right one column
Left arrow	Moves window left one column
Shift-Up arrow	Resizes window; moves bottom up
Shift-Down arrow	Resizes window; moves bottom down
Shift-Right arrow	Resizes window; moves right side toward left
Shift-Left arrow	Resizes window; moves left side toward right
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 entering a file name to load or examine
- when entering 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.

File Lists

When you are prompted for a file name and you supply one of the following responses, you will get a file list to pick from:

- a file name containing the * or ? wildcard characters
- a disk drive letter, like `C:`
- a directory name, like `/MYDIR`

The list of files has two panes. On the left appears a list of the files in the directory you specified. On the right appears a list of the directories in the directory you specified, along with the special entry "..\" indicating the parent directory. You can use *Tab* and *Shift-Tab* to switch between the two panes, just as with other multi-paned windows.

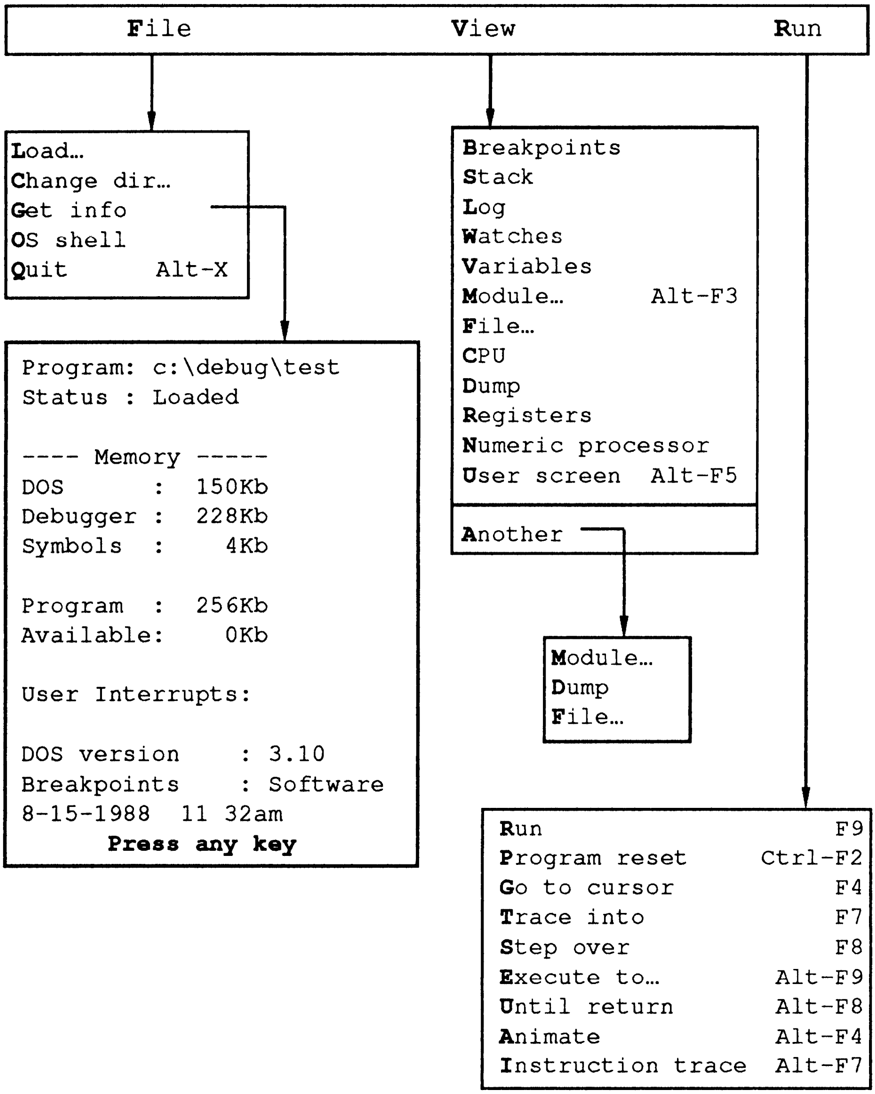
By pressing *Enter* while the highlight is on a directory in the right pane, you switch to that directory. The matching files in that directory then appear in the left pane.

If you want to enter a new directory or wildcard file name, you can press *Ins* and edit or replace the current directory and wildcard mask.

Remember that a file list is like any other list, meaning that you can incrementally match on a file name by starting to type the name of the file that you want. When the highlight is over the correct file name, press *Enter* to accept the file name.

Complete Menu Tree

Figures 12.1, 12.2, and 12.3 show the complete structure of Turbo Debugger's pull-down menus.



```
Program: c:\debug\test
Status : Loaded

---- Memory -----
DOS       :  150Kb
Debugger  :  228Kb
Symbols   :    4Kb

Program   :  256Kb
Available:     0Kb

User Interrupts:

DOS version     : 3.10
Breakpoints     : Software
8-15-1988  11 32am
      Press any key
```

Figure 12.1: The File, View, and Run Menus

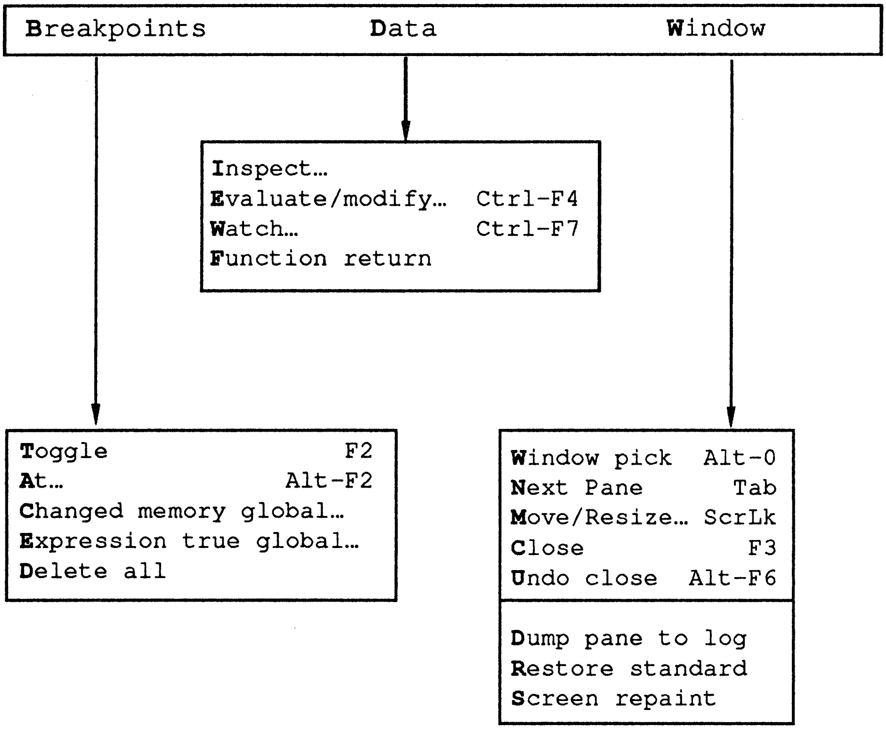


Figure 12.2: The Breakpoints, Data, and Window Menus

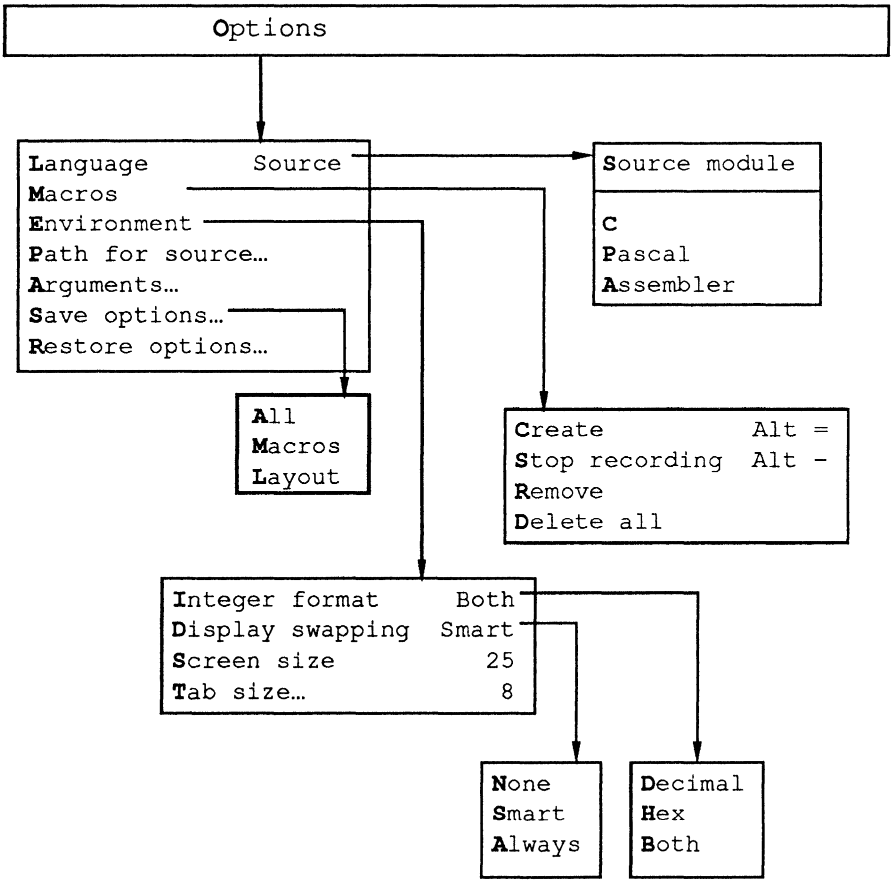


Figure 12.3: The Options Menu

CHAPTER

13

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, while 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 will get at rapidly locating the source of problems in your code. You will learn techniques that suit you well, plus learn to correct methods that may cause you problems time and time again.

Let's begin by looking at where to start when you have a program that doesn't work correctly.

In this chapter, we'll 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.

When Things Don't Work

First and foremost, don't panic! Seldom does even the most expert programmer write 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. A better technique is to use a universally tried-and-true technique: 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 is 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 when debugging it. Some programs are for your own use, or will only be used once or twice to perform a specific task. For these programs, a full scale testing of all the components is probably a waste of time, particularly if you can determine the program is working correctly by inspecting its output. For a program that will be distributed to other people, or that performs a task whose accuracy is hard to determine by inspection, you will want your testing program to be far more rigorous.

Run the Whole Thing

For simple or throw-away programs, the best approach is often just to run it and "see what happens." If your test case has problems, you can then step back and run the program with the simplest possible input and then 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 check that it 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 the debugger to step through the execution of each routine when the whole program is finished.

Types of Bugs

Bugs in your program will 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 will 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 try to avoid making the same mistakes in the future, and to give you a good starting point when looking for bugs in future programs that you write.

The key point here is to try to understand how each bug is an instance of a general family of bugs or misunderstandings, and thereby to 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.

C-Specific Bugs

The *Turbo C User's Guide* has a section on pitfalls in C programming, but what better place to reiterate and expand on those pitfalls than in a lesson on how to debug.

The Turbo C compiler is very good at finding a number of C-specific bugs that other compilers do not 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.)

The following is by no means an exhaustive list of some ways to 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 because they may be telling you of a bug in the making.

Using Uninitialized Auto-Variables

In C, an auto-variable declared inside a function has an undefined value until you load it with something:

```
do_ten_times()
{
   int n;
   while (n < 10)
      {
      ...
      n++;
   }
}
```

This function will execute the while loop an unpredictable number of times since *n* is not initialized to zero before being used as a counter.

Confusing = and ==

C allows you to 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 zero. You almost certainly meant to say

```
if (x == y)
   ...
```

Confusing Operator Precedence

C has so many operators that it is sometimes easy to mix up which ones get applied first when an expression contains many different ones. One of the most common combinations to cause grief is the mixture of shift operators with addition or subtraction. For example,

```
x = 3 << 1 + 1
```

evaluates to 12, not the 7 you might expect if << took place before the +.

Bad Pointer Arithmetic

When you start getting fancy with pointers and use them to step through arrays, be careful of addition and subtraction on pointers. For example,

```
int *intp;
intp += sizeof(int);
```

does not do the hoped-for thing: Increment *intp* to point to the next element of an integer array. In fact, *intp* is advanced by two array elements. When adding to or subtracting from a pointer, C takes into account the size of the item the pointer is pointing to, so all you have to do to move the pointer to the next element is say

```
intp++
```

Unexpected Sign Extension

You must be careful when 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 its consequences. C lets you assign freely between scalar values (**char**, **int**, and so on). When you copy an integer scalar into a larger one, 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 example is sort of 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*.

Superfluous Semicolon

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 not do anything. 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 since the side effect is hidden within the macro definition.

Repeated Auto-Variable 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 auto-variable name *n* is reused in an inner block, hiding access to the one declared in the outer block. You must be careful when re-using variable names in this manner. You can get into this type of trouble easier than you might think, since most programmers use a limited number of variable names for local loop counters (for example, *i*, *n*, and so forth).

Misuse of Auto-Variables

```
int *divide_by_3(int n)
{
   int i;
   i = n / 3;
   return(&i);
}
```

This function means to return a pointer to the result. 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, an indeterminate value will be returned. 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 only exits from a single level of **do, for, switch,** or **while:**

```
for (...)
{
   while (...)
   {
      if (...)
         break;                              /* we want to exit from for loop */
   }
}
```

Here, the **break** only exits from the **while** loop. This is one of the few cases where it is vaguely excusable to use the **goto** statement.

Code Has No Effect

Sometimes a typo results in perfectly compilable source code that doesn't do what you want:

```
a + b;
```

Here, the intended line of code was *a* += *b*.

General Bugs

The following examples barely scratch the surface of the kinds of problems you can have in your programs.

Hidden Effects

Sometimes, a call to a function can leave things in an unexpected way:

```
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

Sometimes, you presume that another routine has already set something up for you:

```
char *workbuf;
addworkstring(char *s)
{
   strcpy(workbuf, s);                                    /* oops */
}
```

You could code this routine defensively by adding the statement:

```
if (workbuf == 0) workbuf = (char *)malloc(20);
```

Not Cleaning Up When Done

This sort of bug can take a long time to finally crash your program by running out of heap space:

```
crunch_string(char *p)
{
   char *work = (char *)malloc(strlen(p));
   strcpy(work, p);
   ...
   return(p);                         /* whoops--work still allocated */
}
```

Fence-Post 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 fence posts 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 will go from one to ten. You'd better make that < into a <= for it to work.

Pascal-Specific Bugs

Because of the strong type- and error-checking features of Pascal, there are few bugs specific to the language itself. However, since 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 still 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. First, as mentioned above, don't use them before assigning 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; Val : integer);
var
  Temp : NodePtr;
begin
  Temp := Head;
  while (Temp^.Key <> 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 will try 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 : 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'll 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 which 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 210.

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 65535 (because words range from 0 to 65535), 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'll return whatever happens to be in *List*[1]. Likewise, if *Count* > *NLMax*, you'll end up either with a runtime 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 to never 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 of their allowed range, or indexing non-existent elements in arrays, as shown in the example above.

The first step in finding such errors is to turn range checking back on by inserting the {$*R*+} compiler option into your program, compiling the program, and running it again. If you know (or suspect) where the error is, you can put this directive above that section and a corresponding {$*R*-} directive afterwards, thus enabling range-checking for that section only. If a range error does occur, your program will stop with a runtime error, and Turbo Pascal will show 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 runtime 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* will equal *Count*+1. If *Count* = *NLMax*, you're beyond the limits for *List*.

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 RETurn to the calling code occurs. When 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 will 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 Needed Registers

When 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 (and 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 if 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 will do 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 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 string instructions to behave differently, depending on code that executed much earlier.

Using the Wrong Sense for a Repeated String Comparison

The **CMPS** instruction compares two areas of memory, while the **SCAS** instruction compares the accumulator to an area of memory. When prefixed by **REPE**, either of these instructions can perform a comparison until either CX becomes zero or a not-equal comparison occurs. When prefixed by **REPNE**, either instruction can perform a comparison until either CX becomes zero or an 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, though. 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 do not work reliably 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 will actually be accessed.

Wiping Out a Register with Multiplication

Multiplication—whether it be 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 either SI or 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, since 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 explicitly preserve only those registers that the handler modifies, it's good insurance to just 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 allow you to logically partition data 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, you should 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 and Limiting Cases

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 that 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 "fence-post" errors (described on page 211).

Erroneous Data Input

Once you are sure that a routine works with a full range of valid input, you should check that it behaves correctly when given invalid input. You should 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 when testing and when 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 inform you of an error condition but continue 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 point of the original error.

Most programs end up being a mixture of these two techniques. Often, you 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 will be 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 debugging a Turbo Pascal program, you'll need TPDEMOB.PAS and TPDEMOB.EXE. If you're a C programmer, you'll need TCDEMOB.C and TCDEMOB.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 Options/Compiler/Optimization/Use 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 230 for a 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 will be 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, and then it is 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 is a good bet that something is wrong during the initial breaking down and counting phase.

Now is the time to start debugging, *after* you have 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 to see if *analyzewords* is correctly counting the analyzed line.

Starting Turbo Debugger

To start the debugging sample session, type

```
TD TCDEMOB
```

Turbo Debugger will load the buggy demo program and then display its startup screen. If you wish to exit from the tutorial session and return to DOS, you can 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 things. There are two approaches you can use: You can either *step* through *makeintowords* as it executes, making sure that it does the right thing, or you can 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 merely by inspecting the output buffer it produces, let's opt for the second approach. To do this, move down to line 41 and press *F4* to run to this line. The program screen will appear and you should type

```
one two three
```

followed by pressing the *Enter* key.

Inspecting

You are now stopped at the source line after *makeintowords* was called. Look at the contents of *buffer* to see if the right thing happened. Move the cursor up a line and 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 (0) at the end of each word as it is meant to. This means that you should execute some more of the program and see if *analyzewords* is doing the right thing. First, remove the inspector by pressing *Esc*. Then, press *F7* twice to execute to the start of *analyzewords*.

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. Since there seems to be a problem with counting characters and words, let's put a breakpoint at the places that a character and a word are counted. Move to line 92 and press *F2* to set a breakpoint. Move to line 96 and set another breakpoint. Finally, set a breakpoint on line 98 so you can look at the character count this function will return. Setting multiple breakpoints like this is a typical way to learn about whether things are happening in the right order in a program, and to allow you to check on important data values each time the program stops at a breakpoint.

Run the program by pressing *F9*. The program stops when it reaches the breakpoint on line 92. 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 indeed shows that *charcount* is 1.

Run the program again by pressing *F9*. You are now back at line 92 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 92 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 98 is if the **while** loop that started on line 82 no longer has a true test value. This means that **bufp* != 0 must evaluate to false. To check this, move back to line 82 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 typing *Alt-D E* and pressing *Enter* to accept the marked expression. The value is indeed 0. Press *Esc* twice 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 85 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 96. 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 96 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 by pressing *Alt-F10/S/E* and then entering `bufp++` at the prompt. Now run the program by pressing *F9*. Enter the usual two input lines

```
one two three
four five six
```

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 anew. Then, go to line 103 and press *F4* to execute to there. Move the cursor to the *nlines* argument and press *Alt-F10 I* to look at its value. It says 6 where it should say 2. Now go back to where it is called from in **main** and look at the value of *nlines* there. Move the cursor to line 35 and 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 45, 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 then run correctly. File TCDEMO.EXE is a corrected version that you may run if you are curious.

Pascal Debugging Session

The rest of this chapter is devoted to a sample debugging session using a Turbo Pascal program. If you're a C programmer, you should look at the preceding sections, which take 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. If you want to recompile it, you can use the command-line compiler, TPC.EXE, and enter this:

```
TPC /v TPDEMOB
```

If you're using the integrated environment (TURBO.EXE), make sure the Debug/Standalone Debugging command is set to On and that the Compile/Destination command is set to 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 letter(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: firsπ 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, you should probably 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 will need 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're done scrutinizing 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 have 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 will load the buggy demo program and then display the startup screen, menus, etc. If you wish to exit from the tutorial session and return to DOS, you can 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*: You can either step through it line-by-line as it executes and make sure it does the right thing, or you can stop the program immediately after *ProcessLine* has done its stuff and then 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.

Okay, so 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 230, press *F4* (Go to Cursor command) and then press *F7* once to step into *ProcessLine*.

Believe it or not, the list is even longer, but try this one: Press *Alt-F9* and a box will pop 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. Enter the same data as before when you are prompted to enter a string (that is, `ABC DEF GHI`).

There are several loops here. An outer one scans the entire string. Inside that loop, there's one to skip over non-letters, and a second one to process words and letters. Move the cursor to the **while** loop on line 132 and press *F4* (Go to Cursor).

This loop keeps scanning until it reaches the end of the string or until it finds a letter. The latter condition 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 it returns a value of False. A not-very-close look reveals that it checks only for uppercase letters. It should either check for characters in the range 'A'..'Z' *and* 'a'..'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 (hindsight is 20/20). You entered both upper- and lowercase letters from 'A' to 'I,' but only half of the letters entered were displayed in the totals. Now you can see why.

Get back to the line that called *IsLetter* by using 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.

By the way, there's another powerful way, to verify *IsLetter's* misbehavior. Display the evaluate window by pressing *Alt-D E* and enter the following expression:

```
  IsLetter('a') = IsLetter('A')
```

They're both letters, but the function result of False confirms that they're not treated the same by *IsLetter*. (You can use the evaluate and watch windows 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 233 and press *F4* to run to the cursor position.

TPDEMOB will prompt you for a string. Type `abc def ghi` and press *Enter* the first time, 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 a *NumWords*. Press *Esc* to close the Inspector window, then move the cursor forward to *NumWords* and press *Alt-F10 I* again (you can also use the shortcut, *Ctrl-I*). Thankfully, *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 performing the division. If you run the program from the beginning and enter no data at all (just press *Enter* when prompted), the program will crash (even after you reverse the divisor and the dividend).

To confirm this, press *Esc* to close the Inspector, 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 will terminate and an error box will display a runtime 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 the it to "walk" through a data structure. Move the cursor up to the declaration of *LetterTable* on line 49. 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

Anyway, 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 now be stopped at *ShowResults.*

ShowResults uses a nested procedure, *ShowLetterInfo,* to display the letter tables. Move the cursor down to line 102, 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 62. 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 stepping through a loop like this, the Watch 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 63. If you look at the Watch 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 63, the current value of *ch,* 'M,' is output and the program moves on. Removing that extra semicolon will eliminate 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 Watch 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 Watch 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 206.

Rather than step through line-by-line, just keep an eye on the Watch window at the end of each iteration. Move the cursor to line 207 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 Watch 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 202 and press *Alt-F10 I*.

Look at the value referenced by *Parm* by pressing *Enter*. You're looking at the first node in the list, and its string data is already corrupted. If you press *Esc, Down arrow,* and then press *Enter* again, you'll open an Inspector window onto the second node in the list. Press *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 Watch window while you use *F7* to step through the loop. The call to *GetMem* on line 198 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 198 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 199 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 and then recompile (they are marked with two asterisks (**) for your convenience), or you can run TPDEMO.PAS, the bug-free version of this program that is discussed in Chapter 3.

CHAPTER

14

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 your program loads and runs at exactly the same address that it does when not being debugged. This 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 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 all the power of a hardware debugger at no additional cost.

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 700K of available extended memory. If you have used up your extended memory for RAM disks, caches, etc., 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 [programoptions]
```

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 K of extended memory used by 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.

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 along with the **–e** option to specify that you want the –**e** value to be permanently set in the TD386 executable program file.

You will then be prompted for the name of the executable program. If you are running on DOS 3.0 or later, the prompt will indicate 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:

–e#### Specifies the number of K of extended memory being used by other programs.

–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.

–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 enulation with a page fram or D000, and modifies TD386.EXE with these values.

For a list of all the command-line options available for TD386.EXE, just type the program name `TD386` 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 whether you are using the 80386 virtual debugging capability. The following items behave differently:

- When you use the *F10*/File/OS 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 while doing 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
    - Windows-386 operating environment
    - 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.

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 will be able to 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 1 megabyte. 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 238.

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 currupted 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

APPENDIX

A

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 form 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	Function
–c<filename>	Startup configuration file
–do	Other display
–dp	Page flipping
–ds	Swap user screen contents
–h	Display help screen
–?	List all the command-line options
–i	Process ID switching
–l	Assembler startup
–m<#>	Heap size (Kb)
–r	Debug on remote system, COM1, slow
–rp<#>	COM port for remote link
–rs<#>	Link speed: 1=slow, 2=med, 3=fast
–sd<dir>	Source file directory
–sc	No case-checking
–vn	43/50 line display not allowed
–vg	Complete graphics save

APPENDIX

B

Turbo Debugger Utilities

Your Turbo Debugger package comes with several utility programs. One utility lets you debug programs developed with Microsoft compilers. This is the CodeView to Turbo Debugger utility, TDCONVRT.EXE.

Another utility works in conjunction with remote debugging and lets you issue basic file-maintenance commands to a remote system. This is the Remote File Transfer utility, TDRF.EXE.

A third program lets you strip the debugging information (the "symbol table") from your programs without relinking. This is the Symbol Table Stripping utility, TDSTRIP.EXE.

There is also a utility that let you pack the debugging information and one that lets you append it from a .MAP file: TDPACK.EXE and TDMAP.EXE.

Finally, we provide a generic object module disassembler program called TDUMP.EXE.

Additionally, we give you a small TSR program, TDNMI.COM that resets the breakout-switch latch if you are using a Periscope I board.

Note: For a list of all the command-line options available for TDCONVRT.EXE, TDRF.EXE, TDSTRIP.EXE, TDPACK.EXE, TDMAP.EXE, and 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
```

CodeView to Turbo Debugger Symbol Table Converter

The TDCONVRT.EXE utility program converts your programs linked with Microsoft Link into a format suitable for use with Turbo Debugger. Before you use this utility, you must prepare your program for debugging exactly as if you were going to use CodeView.

Running from DOS

After you have compiled and linked your program, convert it to Turbo Debugger format by typing at the DOS prompt:

```
TDCONVRT oldname [newname]
```

where *oldname* is the name of the executable program you made with Microsoft Link. You can specify a *newname* for the Turbo Debugger executable program by supplying that name after *oldname*. If you don't specify a new name, the *oldname* file is converted to Turbo Debugger format.

The utility automatically detects whether your program is in CodeView version 1 or version 2 format and adjusts its behavior accordingly.

Error Messages

The conversion utility sometimes encounters a problem that prevents it from converting your program to Turbo Debugger format. If that happens, you will see one of the following error messages.

Can't convert CodeView version 1 symbol tables
You have attempted to convert a program that has a symbol table that can only be used with CodeView version 1. You must recompile and link your program with later versions of Microsoft tools that produce CodeView version 2 symbol tables.

Can't create file: ___
TDCONVRT couldn't create the indicated file. You probably don't have enough room on your disk. Delete a few files that you don't need and try again.

Can't translate a packed symbol table
You are attempting to convert a symbol table that has been processed by the CVPACK utility. TDCONVRT can only work on symbol tables that

have not been packed. You will have to relink your program and not run the packing utility, and then use TDCONVRT.

Error reading from file: ___
An error occurred while reading from the indicated file. This usually means that the file was written on a bad disk sector. Try relinking the program you want to convert so that it gets written to disk again.

Error writing to file: ___
An error occurred while writing to the indicated file. This usually means your disk has filled up. You will have to delete a few files and try the conversion again.

File not found: ___
TDCONVRT couldn't find the file you want to convert. If you don't supply an extension to the file name you want to convert, TDCONVRT assumes that you meant it to be .EXE.

Not enough memory to perform conversion
TDCONVRT has run out of memory while translating the symbol table to Turbo Debugger format. You can try unloading any resident utilities that you have loaded and try the conversion again. If you still don't have enough memory, you can try reducing the amount of debug information in your program. Do this by compiling or assembling only some of its modules with debugging information included.

Program does not have a valid symbol table
You have attempted to convert a program that does not have a symbol table created by MS-LINK. TDCONVRT can only convert symbol tables generated by MS-LINK.

Symbol table contains untranslatable fields
The program you're trying to convert has a symbol table that contains fields that TDCONVRT can't translate. TDCONVRT works with all Microsoft C and assembler programs, but it may have trouble with symbol tables from Microsoft BASIC or Pascal.

Symbol table has invalid format
Your program's symbol table is not formatted correctly. It may have become corrupted or have been generated by an old version of MS-LINK. Make sure that you are using the latest MS-LINK and then try relinking to create a new .EXE file and symbol table on disk.

Remote File Transfer Utility

The Remote File Transfer utility (TDRF) works in conjunction with TDREMOTE running on another system. With TDRF you can perform most DOS file maintenance operations on the remote system. You can:

- copy files to the remote system
- copy files from the remote system
- make directories
- remove directories
- display directories
- change directories
- rename files
- delete files

(See Appendix G, "Remote Debugging," for information on how to start TDREMOTE on a remote system, and for a discussion of remote debugging.)

Once you have started TDREMOTE on the remote system, you can use TDRF at any time. You can start it directly from the DOS prompt, or you can access DOS from inside Turbo Debugger by using the *F10*/File/OS Shell command, and then start TDRF, even while debugging a program on the remote system. This can be useful if you've forgotten to put some files on the remote system that are required by the program you're debugging.

When describing TDRF in the following sections, we refer to the system you are typing at as the "local system" and any files there as "local files," and the other system connected by a cable as the "remote system" and any files there as "remote files."

Starting TDRF from the DOS Command Line

The general form of the command line for TDRF is:

```
TDRF [options] command [arguments]
```

The *options* control the speed of the remote link and which port it runs on. The *options* are described in more detail in the next section.

Command indicates the operation you wish to perform. The command can either be typed the way you are used to doing it with DOS—like COPY, DEL, MD, and so on—or it can be a single-letter abbreviation.

For example, to get a directory display of all files starting with ABC in the current directory on the remote system, you could type:

```
TDRF DIR ABC*
```

All the commands are described fully after the next section.

TDRF Command-Line Options

You must start an option with either a hyphen (-) or a slash (/). Here are the possible command-line options for the Remote File Transfer utility:

–rs*N* Sets the remote link speed.

The **–rs** option sets the speed at which the remote link operates. You must make sure you use the same speed with TDRF that you specified when you started TDREMOTE on the remote system. *N* can be 1, 2, or 3, where 1 signifies a speed of 9600 baud, 2 signifies 40,000 baud, and 3 signifies 115,000 baud.

In other words, the higher the number, the faster the data transfer rate across the link. Normally, TDRF defaults to **–rs3** (the highest speed).

–rp*N* Sets the remote link port.

The **–rp** options specifies which port to use for the remote link. *N* can be either 1 or 2, where 1 stands for COM1 and 2 stands for COM2.

–w Writes options to the executable program file.

You can make the TDRF command-line options permanent by writing them back into the TDRF executable program image on disk. Do this by specifying the **–w** command line option along with the other options you wish to make permanent. You will then be prompted for the name of the executable program.

If you're running on DOS 3.0 or later, the prompt will indicate the path and file name that you executed TDRF 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 TDRF program that you have already made.

If you are running on DOS 2.x, you will have to supply the full path and file name of the executable program.

TDRF Remote File Transfer Utility

You can enter a new executable file name that does not have to already exist. TDRF will create the new executable file.

TDRF Commands

Here are the command names you can use with the Remote File Transfer utility. The wildcards * and ? can be used with the COPY, COPYFROM, DEL, and DIR commands that follow:

COPY

Copies files from the local system to the remote system. You can also type COPYTO instead of COPY. The single letter abbreviation for this command is T.

If you supply a single file name after the COPY command, that file name will be copied to the current directory on the remote system. If you supply a second file name after the name of the file on the local system, the local file will be copied to that destination on the remote system. You can specify either a new file name, a directory name, or a drive name on the remote system. For example,

```
TDRF COPY TEST1 \MYDIR
```

copies file TEST1 from the local system to file \MYDIR\TEST1 on the remote system.

COPYFROM

Copies files from the remote system to the local system. The single letter abbreviation for this command is F.

If you supply a single file name after the COPYFROM command, that file name will be copied from the current directory on the remote system to the current directory on the local system. If you supply a second file name after the name of the file on the remote system, the remote file will be copied to that destination on the local system. You can specify either a new file name, a directory name, or a drive name on the local system. For example,

```
TDRF COPYFROM MYFILE ..
```

copies file MYFILE from the remote system to the parent directory of the current directory on the local system.

```
TDRF F TC*.* A:\TCDEMO
```

copies all files beginning with TC on the current directory of the remote system to the local system's drive A, subdirectory TCDEMO.

DEL

Erases a single file from the remote system. The single letter abbreviation for this command is E.

If you just give a file name with no directory or drive, the file is deleted from the current directory on the remote system. For example,

```
TDRF DEL \XYZ
```

removes file XYZ from the root directory of the remote system.

DIR

Displays a listing of the files in a directory on the remote system. The single letter abbreviation for this command is D.

This command behaves very similarly to the equivalent DOS command. If you don't specify a wildcard mask, it shows all the files in the directory; if you do specify a mask, only those files will be listed. You can interrupt the directory display at any time by pressing *Ctrl-Break.*

The directory listing is displayed in exactly the same format as that used by the DOS DIR command. For example,

```
TDRF DIR \SYS\*.SYS
```

results in a display like:

```
 Directory of  C:\SYS

.            <DIR>       6-08-88   8:01a
..           <DIR>       6-08-88   8:01a
ANSI     SYS      1678   3-17-87  12:00a
VDISK    SYS      3455   3-17-87  12:00a
```

REN

Renames a single file on the remote system. The single letter abbreviation for this command is R.

You must supply two file names with this command: the original file name and the new file name. The new name can specify a different directory as part of the name, but not a different drive. For example,

```
TDRF REN TEST1 \TEST2
```

renames file TEST1 in the current directory in the remote to TEST2 in the root directory. This effectively "moves" the file from one directory to another. You can also use this command to simply rename a file within a directory, without moving it to another directory.

MD

Makes a new directory on the remote system. The single letter abbreviation for this command is M.

You must supply the name of the directory to be created. If you don't supply a directory path as part of the new directory name, the new directory will be created in the current directory on the remote system. For example,

```
TDRF MD TEST
```

creates a directory named TEST in the current directory on the remote system.

RD

Removes an existing directory on the remote system. The single letter abbreviation for this command is K.

You must supply the name of the directory to be removed. If you don't supply a directory path as part of the new directory name, the directory will be removed from the current directory on the remote system. For example,

```
TDRF RD MYDIR
```

removes a directory named MYDIR from the current directory on the remote system.

CD

Changes to a new directory on the remote system. The single letter abbreviation for this command is C.

You must supply the name of the directory to change to. You can also supply a new drive to switch to, or even supply a new drive and directory all at once. For example,

```
TDRF CD A:ABC
```

makes drive A the current drive on the remote system, and switches to directory ABC as well.

TDRF Messages

Here is a list of the messages you might encounter when working with the Remote File Transfer utility:

Can't create file on local system: ___
When copying a file from the remote system using the COPYFROM command, the file could not be created on the local system. This will happen if the disk is full on the local system, or if the file name on the remote system is the same as a directory name on the local system.

Can't modify exe file
The file name you specified to modify is not a valid copy of the TDRF utility. You can only modify a copy of the TDRF 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.

Error opening file: ___
The file you wanted to transfer to the remote system could not be opened. You probably specified a nonexistent or invalid file name.

Error writing file: ___
An error occurred while writing to a file on the local system. This usually happens when there is no more room on your disk. You will have to delete some files to make room for the file that you want to copy from the remote system.

Error writing file ___ on remote system
An error occurred while writing a file to the disk on the remote system. This usually happens if the remote disk is full. You must delete a few files to make room for the file that you want to transfer.

File name is a directory on remote
You have tried to copy a file from the local to the remote system, but the local file name exists as a directory on the remote system. You will have to rename the file by giving a second argument to the COPY command.

Interrupted
You have pressed *Ctrl-Break* while waiting for communications to be established with the remote system.

Invalid command: ___
You have entered a command that TDRF does not recognize. For each command, you can use the DOS-style command word or the single letter abbreviation.

Invalid command line option: ___
You have given an invalid command-line option when starting TDRF from the DOS command line.

Invalid destination disk drive
You have specified a nonexistent disk drive letter in your command. Remember that the remote system might have a different number of disk drives than the local system.

No matching files on remote
You have done a directory command but either there are no files in the directory on the remote system, or no files match the wildcard specification that you gave as an argument to the DIR command.

No remote command specified
You have not specified any command on the DOS command line; TDRF has nothing to do.

Too few arguments
You have not supplied enough arguments for the command that you requested. Some commands require an argument, like DEL, MD, CD, RD, and so on.

Too many arguments
You have specified too many arguments for the command that you requested. No command requires more than two arguments, and some require only one.

Wrong version of TDREMOTE
You are using an incompatible version of TDRF and TDREMOTE. Make sure that you are using the latest version of each utility.

Symbol Table Stripping Utility

The Symbol Table Stripping utility lets you remove the symbol table from an executable program generated by TLINK with the /v option. This is a faster way of removing the symbol table than relinking without the /v option.

You can also use this utility to remove the symbol table and put it in a separate file. This is useful when you want to convert the .EXE format program to a .COM file, and still retain the debugging symbol table. This utility puts the symbol table in a file with the extension .TDS. Turbo Debugger will look for this file when it loads a program to debug that does not have a symbol table.

TDSTRIP Command Line

Here is the general form of the DOS command line used to start the Symbol Table Stripping utility, TDSTRIP:

```
TDSTRIP [-s][-c] exename [outputname]
```

If you don't specify the **–s** option, the symbol table is removed from the .EXE file *exename*. If you specify an *outputname*, the original .EXE file is left unchanged and a version with no symbol table is created as *outputname*.

If you do specify the **–s** option, the symbol table will be put in a file with the same name as *exename* but with the extension .TDS. If you specify an output file, the symbol table will be put in *outputname*.

If you specify the **–c** option, the input .EXE file is converted into a .COM file. If you use **–c** in conjunction with **–s**, you can convert an .EXE file with symbols into a .COM file with a separate .TDS symbol file. This lets you debug .COM files with Turbo Debugger while retaining full debugging information.

You can only convert certain .EXE files into .COM files. The same restrictions apply to the **–c** option of TDSTRIP as to the **/t** option of TLINK: Your program must start at location 100 hex, and it can't contain any segment fixups.

If you don't supply an extension with *exename*, .EXE is presumed. If you don't supply an extension with *outputname*, .EXE is added when you don't use **–s**, and .TDS is added when you do use **–s**.

Here are some sample TDSTRIP command lines:

```
TDSTRIP MYPROG
```

removes the symbol table from MYPROG.EXE.

```
TDSTRIP -s MYPROG.OLD
```

removes the symbol table from MYPROG.OLD and places it in MYPROG.TDS.

```
TDSTRIP MYPROG MYPROG.NEW
```

leaves MYPROG.EXE unchanged but creates another copy of it named MYPROG.NEW without a symbol table.

```
TDSTRIP -s MYPROG MYSYMS
```

removes the symbol table from MYPROG.EXE and places it in MYSYMS.TDS

TDSTRIP Error Messages

Here is a list of the possible error messages you might encounter when working with the Symbol Table Stripping utility:

Can't create file: ___
TDSTRIP could not create the output symbol or .EXE file. Either there is no more room on your disk, or you specified an invalid output file name.

Can't open file: ___
TDSTRIP could not locate the .EXE file that you wish to remove the symbol table from.

Error reading from input exe file
An error occurred while reading from the input executable program file. Your disk may be unreadable. Try the operation again.

Error writing to output file: ___; disk may be full
TDSTRIP could not write to the output symbol or executable file. This usually happens when there is no more room on your disk. You will have to delete some files to make room for the file created by TDSTRIP.

Input file is not an .exe file
You have specified an input file name that is not a valid executable program. You can strip symbols only from .EXE programs, since these are the only ones that TLINK can put a symbol table on. Programs in .COM file format do not have symbol tables and cannot be processed by TDSTRIP.

Invalid command line option: ___
You have given an invalid command line option when starting TDSTRIP from the DOS command line.

Invalid exe file format
The input file appears to be an .EXE format program file, but something is wrong with it. You should relink the program with TLINK.

Not enough memory
Your system does not have enough free memory for TDSTRIP to load and process the .EXE file. This only happens in extreme circumstances (TDSTRIP has very modest memory requirements). You should probably try rebooting your system and trying again. You may have previously run a program that allocated some memory that won't be freed until you reboot.

Program does not have a symbol table
You have specified an input file that is a valid exe file, but it does not have a symbol table.

Program does not have a valid symbol table
The symbol table at the end of the .EXE file is not a valid TLINK symbol table. This can happen if you try and use TDSTRIP on a program created by a linker other than TLINK.

Too many arguments
You can supply a maximum of two arguments to TDSTRIP, the first being the name of the executable program, and the second being the name of the output file for symbols or the executable program.

You must supply an exe file name
You have started TDSTRIP without giving it the name of an EXE program file whose symbol table you want to strip.

TDMAP Utility

The TDMAP utility takes a .MAP file—an ASCII file created by the linker containing all public symbols of a program—and appends it to the .EXE file in Turbo Debugger format. This allows you to debug an executable program that you compiled with a non-Borland compiler and linker.

The syntax for using TDMAP is

```
TDMAP filename [/sourceextension]
```

TDMAP reads *filename.MAP* and adds debugging information to *filename.EXE.* For example, if you want to append debugging records to a program called HELLO.PAS, you could enter

```
TDMAP hello /Epas
```

Note that the extension, which is preceded with `/E`, is optional.

TDPACK Utility

The TDPACK utility compresses the debugging information that's appended to an .EXE file, making the executable file smaller. It does this by eliminating duplicate information, such as strings or data type information.

The syntax for using TDPACK is

```
TDPACK filename
```

If no extension is specified, TDPACK assumes the extension is .EXE. If it cannot find *filename*.EXE, TDPACK looks for *filename*.COM and *filename*.TDS (a .TDS file contains Turbo Debugger symbols for use with a .COM file).

TDUMP Utility

The TDUMP utility program is a generic module disassembler you can use to examine the structure of any file.

TDUMP attempts to break apart a file as intelligently as possible. To do this, it first looks to the file's extension, and if it recognizes it, it displays the the file's components according to the type. TDUMP recognizes .EXE, .OBJ, and .LIB files. Any file type other than these results in a straight hex dump of the file.

You can use TDUMP to peek into the inner structure of any file. This not only shows you what's in a file, but also teaches you how files are constructed. Moreover, because TDUMP verifies that a file's structure matches its extension, you can also use TDUMP to test file integrity.

TDUMP Syntax

The syntax for TDUMP is

```
TDUMP [options] inputfile [outputfile]
```

Inputfile is the file whose structure you want to display (or "dump"). *Outputfile* is an optional file name to send the display to (you can also use the standard DOS redirection command ">"). *Options* stands for any of the TDUMP options discussed in the next section.

TDUMP Options

You can use several optional "switches" with TDUMP, all of which start with a hyphen (or you can use a slash instead). The following two examples are equivalent:

```
TDUMP -el -v demo.exe

TDUMP /el /v demo.exe
```

The –a and –a7 Options

TDUMP automatically adjusts its output display according to the file type (it shows an unrecognized file type in hex display). You can, however, force the display to be in ASCII by including the **–a** or **–a7** option.

An ASCII file display shows the offset and the contents in displayable ASCII. If a character is not displayable (like any Control character), it appears as a period.

The **–a7** option works just like the **–a** except that it converts any high-ASCII characters to their low-ASCII equivalents. This is handy if the file you're dumping sets high-ASCII characters as flags (as WordStar files do).

The –e, –el, and –er Options

All three of these options force TDUMP to display the file as if it were an executable (.EXE) file.

An .EXE file display consists of lists of file statistics at the top of the display, followed by the global symbol table and the module table.

The **–el** option works just like **–e** option except that it suppresses line numbers in the display.

The **–er** option works just like **–e** option except that it prevents the the relocation table from displaying.

You can suppress both line numbers and the relocaion table by using **–elr** as an option.

The –h Option

The **–h** option forces TDUMP to display the file in hexadecimal format. The hex format consists of a column of offset numbers, columns of hex numbers, and then ASCII equivalents (with periods appearing where there are no displayable ASCII characters).

If TDUMP does not recognize the file extension of the input file, it defaults to displaying the file in hex format (unless an option forces it to show it in another format).

The –l Option

The –l option forces TDUMP to display the file as if it were a library (.LIB) file. A library file is a collection of object files (see the following section for more on object files). A library file dump shows first some library-specific information, then each of the object files, then each record in each object file.

The –o Option

The **–o** option forces TDUMP to display the file as if it were an object (.OBJ) file. An object file contains descriptions of the command records that pass commands and data to the linker, telling it how to create an .EXE file. The display format shows each record and its associated data, on a record-by-record basis.

The –v Option

The **–v** option affects all file formats equally by suppressing any descriptions TDUMP ordinarily inserts into the dump to improve readability. If you use the **–v** option, the display is a *verbatim* dump of the file's components—as terse as possible. You need to be an advanced programmer to interpret a verbatim display.

The TDNMI Utility

Use TDNMI if you have a Periscope I board and wish to use its breakout switch with Turbo Debugger. TDNMI is a small TSR program that periodically resets the breakout-switch latch on the Periscope board. Use the **/p** command-line option to set the board's base address if it is different from the default address of 300.

If you are using a PC clone that disables the NMI interrupt (such as some PC's Limited systems), you can install the TDNMI resident utility to clear the NMI every half second. You will need to do this if you are using a breakout switch on such a system.

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

```
TDNMI -h
```

APPENDIX

C

Technical Notes

This appendix is for advanced users who wish to understand some of the technical details that underly 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 debugger user.

Some of the information in this chapter will allow you to understand how the debugger interacts with DOS, the hardware, and with your program. This can help you understand how your program's behavior might differ while running under the 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 may appear or disappear that are the result of accessing memory outside your program. By changing the amount of free memory, bugs in your memory allocation or usage 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 14 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 0000:0040 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 the debugger and each time the debugger is reentered from your program. Since DOS is not reentrant, 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 wish 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

The 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 the debugger.

Interrupts Used by Turbo Debugger

The 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
The debugger uses these interrupts to process breakpoints and instruction single-stepping. If these interrupts are modified by your program, the debugger may not be able to regain control at the next breakpoint. Normal applications never use these interrupts, since 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-press and release codes. The debugger chains into this interrupt when the user program is running, so that it can regain control of a program stuck in a loop. The debugger re-installs 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 the debugger to step through your code.

Display-Saving and Mode-Switching

The 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, the debugger may disturb those settings.

Memory Consumption

When you first start the debugger, DOS loads it into the first free memory above DOS and any resident programs. Then, the 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, the 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. The debugger saves and restores the state of the EMS driver, allowing you to 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 use by the debugger. Use the TDINST installation utility to do this.

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 *F10*/File/Quit (or *Alt-X*) command. These vectors are also restored if you use the *F10*/File/OS Shell command to enter a DOS command while debugging a program.

The second set of vectors are Turbo Debuggers vectors. These are in effect whenever Turbo Debugger is running and on the screen. 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, while at the same time allowing Turbo Debugger to use its own version of those same interrupts.

APPENDIX

D

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 D.1: 8086/80186/80286 Instruction Mnemonics

AAA
AAD
AAM
AAS
ADC
ADD
AND
ARPL**
BOUND*
CALL
CLC
CLD
CLI
CLTS**
CMC
CMP
CMPSB
CMPSW
CWD
DAA
DAS
DEC
DIV
ENTER*
ESC
HLT
IDIV
IMUL
IN
INC
INSB*
INSW*
INT
INTO
IRET
JB
JBE
JCXZ
JE
JL
JLE
JMP
JNB
JNBE
JNE
JNLE
JNO
JNP
JO
JP
JS
LAHF
LAR**
LDS
LEA
LEAVE*
LES
LGDT**
LIDT**
LLDT**
LMSW**
LOCK
LODSB
LODSW
LOOP
LOOPNZ
LOOPZ
LSL**
LTR**
MOV
MOVSB
MOVSW
MUL
NEG
NOP
NOT
OR
OUT
OUTSB*
OUTSW*
POP
POPA*
POPF
PUSH
PUSHA*
PUSHF
RCL
REPNZ
REPZ
RET
REFT
ROL
ROR
SAHF
SAR
SBB
SCASB
SCASW
SGDT**
SHL
SHR
SLDT**
SMSW**
STC
STD
STI
STOSB
STOSW
STR**
SUB
TEST
WAIT
VERR**
VERW**
XCHG
XLAT
XOR

* = Available only when running on the 186 and 286 processor

** = Available only when running on the 286 processor

Table D.2: 8087/80287 Numeric Processor Instruction Mnemonics

FABS
FADD
FADDP
FBLD
FBSTP
FCHS
FCLEX
FCOM
FCOMP
FCOMPP
FDECSTP
FDISI
FDIV
FDIVP
FDIVR
FDIVRP
FENI
FFREE
FIADD
FICOM
FICOMP
FIDIV
FIDIVR
FILD
FIMUL
FINCSTP
FINIT
FIST
FISTP
FISUB
FISUBR
FLD
FLDCW
FLDENV
FLDLG2
FLDLN2
FLDL2E
FLDL2T
FLDPI
FLDZ
FLD1
FMUL
FMULP
FNOP
FNSTS**
FPATAN
FPREM
FPTAN
FRNDINT
FRSTOR
FSAVE
FSCALE
FSETPM*
FSQRT
FST
FSTCW
FSTENV
FSTP
FSTSW*
FSUB
FSUBP
FSUBR
FSUBRP
FTST
FWAIT
FXAM
FXCH
FXTRACT
FYL2X
FYL2XP1
F2XM1

* = Available only when running on the 287 numeric processor

** = On the 80287, the **fstsw** and **fnstsw** instructions can use the AX register as an operand, as well as the normal memory operand.

Table D.3: CPU Registers

Byte registers	ah, al, bh, bl, ch, cl, dh, cl
Word registers	ax, bx, cx, dx, si, di, sp, bp
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 D.4: Special Keywords

WORD PTR
BYTE PTR
DWORD PTR
QWORD PTR
TBYTE PTR
NEAR
FAR
SHORT

Turbo Debugger supports all 80386 and 80387 instruction mnemonics and registers:

Table D.5: 80387 Registers

EAX
EBX
ECX
EDX
ESI
EDI
EBP
ESP

Table D.6: 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 D.7: 80387 Instruction Mnemonics

FCOS	FUCOM
FSIN	FUCOMP
FPREM1	FUCOMPP
FSINCOS	

APPENDIX

E

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 options by running the installation program called TDINST.EXE. 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 installation program.

The installation program lets you set the following items:

- Window colors
- Display parameters: Beginning display, screen swapping mode, integer display format, log list length, tab column width, maximum tiled Watches size, snow control, 43-/50-line mode, and graphics saving
- Your editor startup command
- Directories to search for source files
- Keyboard parameters: program interrupt key and control-key shortcuts
- Prompting parameters: beep on error, *Esc* required to clear error and history list length
- Lowercase symbols
- DOS process ID-switching
- expanded memory specification (EMS) for symbol table

Running TDINST

To run the installation program, enter `TDINST` at the DOS prompt. You can either press the highlighted first letter of a menu option or use the *Up* and *Down arrow* 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

Press *C* in the main menu to bring up the color selection menu. You are then given four choices of the colors to choose: Customize, **1** default color set, **2** default color set, and **3** default color set. Since there are so many screen items that can be given different colors, you will probably want to choose one of the default color sets by pressing *1*, *2*, or *3*. If you do not like any of the default color sets, you can design your own screen colors by pressing *C* to customize the colors.

You can also choose one of the default color sets and modify portions of it. To do this, press *1*, *2*, or *3* to select a color set, and then press *C* so you can modify it.

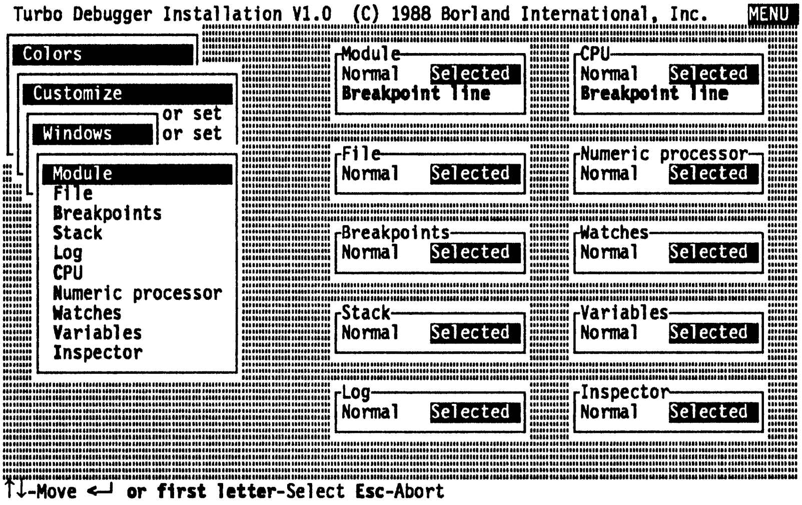
If you choose to customize the colors, a new menu will be displayed with these options:

Windows Lets you change the color of the different windows Turbo Debugger uses to display information about your program.

User Input Allows you to choose the color of the different boxes that prompt you for information, including menus, error messages, help, and history lists.

Screen Lets you choose the pattern and color for the screen background, as well as choose the colors for the help line.

Each of these options leads to another menu from which you can pick the window or box whose colors you wish to change. If you are running on a color display, the full range of 16 foreground and 8 background colors appears in the form of a color palette. On a black-and-white display, the various attribute combinations of underlined, reverse video, and highlight appear.



```
Turbo Debugger Installation V1.0  (C) 1988 Borland International, Inc.  MENU
 Colors            Module                     CPU
  Customize        Normal   Selected          Normal   Selected
          or set   Breakpoint line            Breakpoint line
   Windows or set
                   File                       Numeric processor
    Module         Normal   Selected          Normal   Selected
    File
    Breakpoints    Breakpoints                Watches
    Stack          Normal   Selected          Normal   Selected
    Log
    CPU
    Numeric processor
    Watches        Stack                      Variables
    Variables      Normal   Selected          Normal   Selected
    Inspector
                   Log                        Inspector
                   Normal   Selected          Normal   Selected
↑↓-Move ↵ or first letter-Select Esc-Abort
```

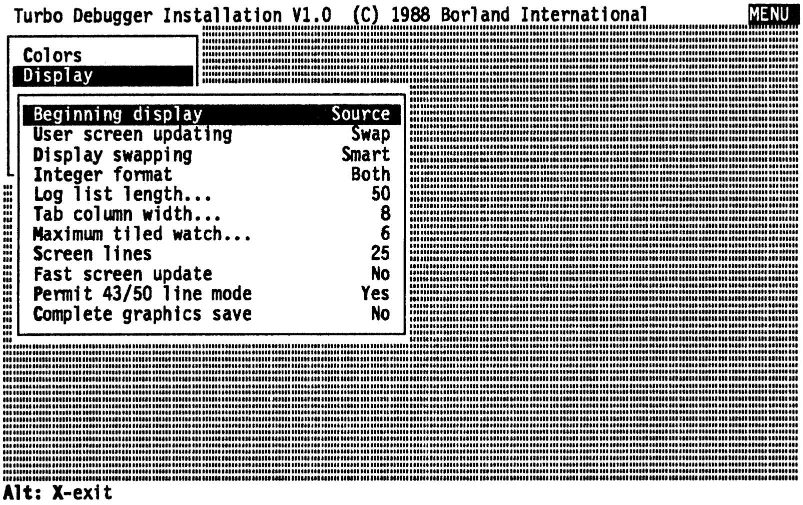
Figure E.1: Customizing Window Colors

Once you've selected the item you wish to change, the palette box will pop up over the menu. You can use the arrow keys to move around in the palette box. To the right of the menu area and the palette, you will see all the windows or boxes in the category that you are changing. As you move the menu highlight through the various color choices, the window whose colors you are changing will be updated to show how your selection appears. When you find the colors you like, press *Enter* to accept that selection.

Note: 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 wish to adjust the black and white table, you would first set your video mode to black and white by typing `MODE BW80` at the DOS prompt, and then run TDINST.

Setting Turbo Debugger Display Parameters

Press *D* from the main menu to bring up the **D**isplay menu. These 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 using TDINST.



```
Turbo Debugger Installation V1.0  (C) 1988 Borland International        MENU
Colors
Display
   Beginning display               Source
   User screen updating              Swap
   Display swapping                 Smart
   Integer format                    Both
   Log list length...                  50
   Tab column width...                  8
   Maximum tiled watch...               6
   Screen lines                        25
   Fast screen update                  No
   Permit 43/50 line mode             Yes
   Complete graphics save              No
Alt: X-exit
```

Figure E.2: Customizing Display Parameters

The Beginning Display Option

For **B**eginning Display, press *B* at the **D**isplay menu to set how the display appears when Turbo Debugger starts. Choosing **B**eginning Display toggles between the following settings:

Assembler Assembler Beginning: None of your program is executed, and a CPU viewer shows the first instruction in your program.

Source Source startup: Your program's compiler Beginning code runs and you start in a Module viewer where your source code begins.

The User Screen Updating Option

Choose **U**ser Screen Updating from the **D**isplay menu to set how the display gets updated when it switches between your program's screen and the Turbo Debugger screen. Choosing this item toggles between the following options:

Flip Pages Puts Turbo Debugger's screen on a separate display page. This option only works 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 will only disturb the operation of programs that use multiple display pages—and these are few and far between.

Swap Uses a single display adapter and display page, and swaps the contents of the user and Turbo Debugger screen 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, use this option. Also use the Swap option if you Shell out 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.

Display Swapping Option

Choose Display Swapping from the Display menu when you want Turbo Debugger to switch between its own display and the display of the program you're debugging. Choosing this command toggles between the following settings:

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

Smart Only swap to the user screen when display output may occur. Turbo Debugger will swap 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's writing to the screen. If you choose this option, the screen will flicker every time you step your program, since Turbo Debugger's screen will be replaced for a short time with your program's screen.

The Integer Format Option

Choose **I**nteger from the TDINST **D**isplay menu to set how integers are displayed. You can toggle between the following options:

Hex Chooses hex number display.

Decimal Chooses decimal number display.

Both Displays both hex and decimal.

The Log List Length Option

Choose **L**og List Length from the TDINST **D**isplay menu to set how many previous responses are saved for each prompt. You will be prompted for the number of responses to save. (The maximum is 200; the minimum is 4.)

The Tab Column Width Option

Choose **T**ab Column Width from the TDINST **D**isplay menu to set the number of columns between tab stops when displaying a text or source file. You will be prompted for the number of columns (a number from 1 to 32).

The Maximum Tiled Watch Option

Choose **M**aximum Tiled Watch from the TDINST **D**isplay menu to set the number of lines that the Watches window can expand to when in tiled mode. You will be prompted for the number of lines (a number from 1 to 20).

The Screen Lines Option

Press *S* to toggle whether Turbo Debugger should start up with a big display screen of 43 or 50 lines; only the EGA and VGA can display more than the usual 25 lines.

The Fast Screen Update Option

Press *F* to toggle whether color displays will be updated quickly. Toggle this option off if you get "snow" on your display with fast updating enabled. You only need to disable this option if the "snow" annoys you. (Some people prefer the snowy screen because it gets updated more quickly.)

Permit 43-/50-Line Mode

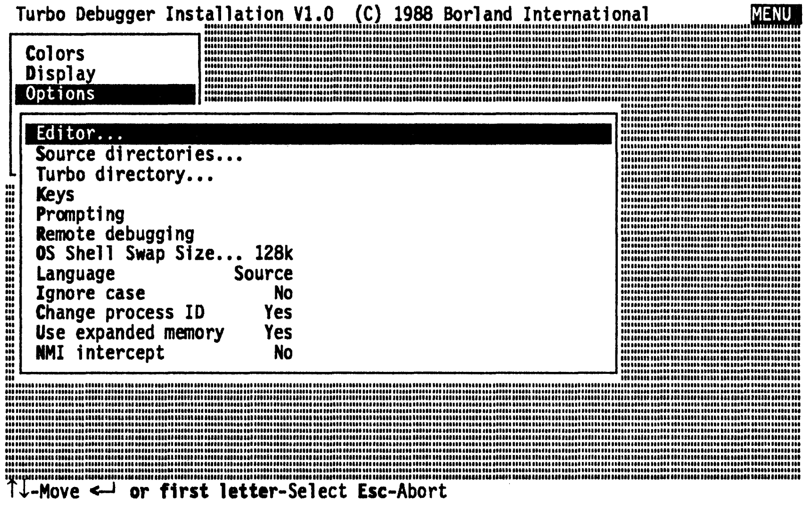
Press *P* to toggle whether Turbo Debugger allows big display modes. If you disable this option, you will save approximately 8K since the large screen modes need more window buffer space in the debugger. This may be helpful if you are debugging a very large program that needs as much memory as possible to execute in. When disabled, you will not be able to switch the display into 43-/50-line mode even if your system is capable of handling it.

Complete Graphics Save

Press *C* to toggle whether the entire graphics display buffer gets saved when switching between your program's screen and the Turbo Debugger screen. If you turn this option off, you will save approximately 12K of memory. This may be helpful if you are debugging a very large program that needs as much memory as possible to execute in. Generally the only price you pay for disabling this option is a small number of corrupted locations on your program's screen that don't usually interfere with debugging.

Setting the Turbo Debugger Options

Press *O* to display the TDINST **O**ptions menu. (You can set some of these options from the Turbo Debugger command line, but most of them can be set only using TDINST.)



```
Turbo Debugger Installation V1.0  (C) 1988 Borland International            MENU
 Colors
 Display
 Options
   Editor...
   Source directories...
   Turbo directory...
   Keys
   Prompting
   Remote debugging
   OS Shell Swap Size... 128k
   Language             Source
   Ignore case              No
   Change process ID       Yes
   Use expanded memory     Yes
   NMI intercept            No
↑↓-Move <┘ or first letter-Select Esc-Abort
```

Figure E.3: Customizing the Options

The Editor Option

Choose Editor from the Options menu to specify the DOS command that starts your editor. This allows Turbo Debugger to start up your favorite editor when you are debugging and wish to change something in a file.

Turbo Debugger will add to the end of this command the name of the file that it wants to edit, separated by a space.

The Source Directories Option

Choose Source Directories from the Options menu to change the list of directories Turbo Debugger searches for source files.

The Turbo Directory Option

Choose Turbo Directory from the Options menu to set the directory that Turbo Debugger will look in for its help file and configuration file.

The Keys Option

Selecting **K**eys from the **O**ptions menu lets you choose how Turbo Debugger interprets certain keys. Here are the options you can choose from:

Key for Interrupt

Lets you specify which key to press with the *Ctrl* key to interrupt your program. You can choose between the *Break, Esc,* or *NumLock* keys, or you can enter a scan code number to choose any of the keys on the standard IBM or compatible keyboard. Make sure that you choose a key combination that does not conflict with any that your program currently uses. You choose the key for interrupt by directly pressing the key combination that you wish to use. You can use any combination of the *Shift-Left arrow, Shift-Right arrow, Alt,* and Ctrl keys, along with a normal keyboard character like a letter, function key, etc. For example,

Shift-Alt-F1

Left arrow Shift-Right arrow Shift-Space

Extended 101-key keyboards, Turbo Debugger does not distinguish between the left and right Alt keys or the left and right Ctrl keys. The left and right shift keys are still treated distinctly.

Control-Key Shortcuts

Toggles between enabling and disabling 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, you then can't use those Control keys as WordStar-style cursor movement commands.

The Prompting Options

Choose **P**rompting from the **O**ptions menu to adjust how you are prompted for information and how you must respond. Press the first letter of the menu item you wish to change:

Beep on Error	Toggles between beeps enabled and disabled. With beeps enabled, error messages and invalid key presses make a beep.
Error Message Clearing	Forces you to press *Esc* to clear error messages. Selecting Next Key Clears removes error messages automatically when you press a key to issue a new command.
History List	Lets you specify how many old input lines to keep for each prompt.

OS Shell Swap Size

Choose **O**S Shell from the TDINST **O**ptions menu to specify the amount of room Turbo Debugger will make sure is available when you want to enter a DOS command from within Turbo Debugger.

Remote Debugging

Choosing **R**emote Debugging from the TDINST Options menu lets you choose how Turbo Debugger communicates with the remote system. Press the first letter of the menu item you wish to change:

Remote Debugging

Toggles between enabling and disabling the remote link. You usually won't want to set this to Yes, since that will mean that Turbo Debugger will start up every time using the remote link.

Port

Toggles between using the COM1 or COM2 serial port for the remote link.

Speed

Toggles between the three speeds that are available for the remote link: 9600 baud, 40,000 baud, or 115,000 baud.

Language

Choosing Language lets you specify what language Turbo Debugger will use for evaluating expressions. You can choose between:

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.

The Ignore Case Option

Pressing *I* toggles between treating lowercase differently from uppercase, and treating uppercase and lowercase the same.

The Change Process ID Option

Press *C* to control whether Turbo Debugger uses process ID-switching. Do not disable this unless you are tracing through DOS and have a good understanding of the technical issues discussed in Appendix C.

The Use Expanded Memory Option

Press *U* to toggle whether Turbo Debugger uses EMS memory for symbol tables. You can enable this option even if your program uses EMS as well.

The NMI Intercept Option

If your computer is a Tandy 1000A, IBM PC Convertible, or NEC MultiSpeed, or if Turbo Debugger hangs loading your system, run TDINST and change the default setting for NMI Intercept to No. Some computers use the NMI (Non-Maskable 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.

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 the debugger command-line options and the installation program command that permanently sets that option.

Command-Line Option	TDINST Menu Selection
–do	Display/User Screen Updating/Other
–dp	Display/User Screen Updating/Flip
–ds	Display/User Screen Updating/Swap
–i	Options/Change Process ID/Yes
–i-	Options/Change Process ID/No
–l	Display/Beginning Display/Assembler
–l-	Display/Beginning Display/Source
–vn	Display/Permit 43/50 Line Mode/No
–vn-	Display/Permit 43/50 Line Mode/Yes
–vg	Display/Complete Graphics Save/Yes
–vg-	Display/Complete Graphics Save/No
–r	Options/Remote Debugging/Remote Debugging/Yes
–r-	Options/Remote Debugging/Remote Debugging/No
–rp#	Options/Remote Debugging/Port/#
–rs#	Options/Remote Debugging/Speed/#
–sd	Options/Source Directories
–sc	Options/Ignore Case/Yes
–sc-	Options/Ignore Case/No

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

```
TDINST -h
```

Quitting the Program

When you have finished making changes to the configuration, return to the main menu by pressing *Esc* as many times as needed to return to the main menu. Then choose Quit.

If you have made any changes to the configuration, you will be prompted for whether you want to save them into the Turbo Debugger executable program file TD.EXE. If you want to install the changes, press *Y*. If you don't want to change the configuration options, press *N*. You will then be returned to the DOS prompt.

TCINST also has separate **S**ave command on the main menu. If you select **S**ave, a menu appears that lets you choose between saving the configuration directly to the Turbo Debugger executable program file TD.EXE, or to a configuration file.

If you choose to save to a configuration file, a prompt appears 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 **O**ptions/**R**estore Configuration command to load a configuration once you have started Turbo Debugger.

If at any time, you want to return to the standard configuration that Turbo Debugger is shipped with, copy TD.EXE from your master disk onto your working system disk.

APPENDIX

F

Hardware Debugger Interface

Hardware debugging boards greatly speed up certain types of breakpoints, in particular those that watch for an area of memory or a program variable to change. Turbo Debugger has a general interface for accessing these boards.

This appendix describes how to install the device driver supplied with Turbo Debugger and how to write a device driver that Turbo Debugger can communicate with in order to make use of the capabilities of a particular hardware debugger.

80386 Hardware Device Driver

This information is intended for vendors of hardware debuggers who want to let Turbo Debugger make use of their boards. (Note that you must know the general architecture of DOS device drivers. Refer to the DOS Technical Reference for more information on how to write device drivers.)

The Turbo Debugger distribution disk contains the file TDH386.SYS, which is a hardware device driver that lets Turbo Debugger use the debug registers on the 80386 processor. You can use this driver by copying it to your DOS disk and putting the following line in your CONFIG.SYS file:

```
DEVICE = TDH386.SYS
```

Turbo Debugger will then use the hardware assistance of the 80386 whenever it can to speed up breakpoint processing.

This means, of course, that you can only use this device driver if your system uses the 80386 processor.

Setting Hardware Breakpoints

There are three ways of setting a hardware-assisted breakpoint:

- Use the *F10*/Breakpoints/Changed Memory Global command
- Use the Condition/Changed Memory command in the Breakpoints window local menu
- Use the Condition/Hardware command in the Breakpoints window local menu

When you set a breakpoint using one of the Changed Memory commands, Turbo Debugger automatically determines whether that breakpoint can make use of the available hardware. If it can, Turbo Debugger sets a hardware breakpoint for you, and indicates that the breakpoint is set in hardware by putting an asterisk (*) after the global breakpoint number in the left pane of the Breakpoints window.

The following two sections describe how to set generalized hardware breakpoints. First, we'll tell you the types of breakpoints you can set with the TDH386.SYS device driver, and then we'll tell you about all the hardware breakpoint options that may work with other hardware debugger device drivers. Consult the vendor's documentation for more information about your particular device driver.

Hardware Conditions Permitted with TDH386.SYS

When you are using TDH386.SYS with the ordinary version of Turbo Debugger, you can set the following types of hardware breakpoints from the Condition/Hardware command in the Breakpoints window local menu:

- instruction fetch
- read from memory
- read/write memory

You can't set any type of data matching when you use TDH386.SYS; you must always set the data match option to All. You can also only match a single memory address or a range of memory addresses. A range can encompass from 1 to 16 bytes, depending on how many other hardware breakpoints you have set and the address of the beginning of the range.

You can use data watching with breakpoints that are not longer than 4 bytes.

The other options on the **Hardware** menu in the Breakpoints window are for other hardware debuggers and device drivers that might support more matching modes.

You can also use TDH386.SYS with Turbo Debugger (TD) and with the virtual debugger (TD386). In this case, you can use many more types of hardware breakpoints, including matching on any size ranges of memory or I/O access.

See Chapter 14 for more information on using the virtual debugger.

Breakpoints Window Hardware Conditions Menu

This section describes the **Hardware** menu that can be selected from the Condition option in the Breakpoints window. Remember that your hardware most likely doesn't support all combinations of matching that you can specify from this menu. The previous section describes the combinations that are allowed for the TDH386.SYS device driver supplied with Turbo Debugger.

The **Hardware** menu lets you set the three matching criteria that make up a hardware breakpoint:

- the bus cycle type that will be matched
- the range of addresses that will be matched
- the range of data values that will be matched

For example, a hardware breakpoint might say "Watch for an I/O write anywhere from address 3F8 to 3FF as long as the data value is equal to 1." This breakpoint will then be triggered any time a byte of 1 is written to any of the I/O locations that control the COM1 serial port.

Usually, you set far more simple hardware breakpoints than this, such as "Watch for I/O to address 200."

The **Cycle Type** command of the **Hardware** menu has the following settings:

Read Memory	Match memory reads
Write Memory	Match memory writes
Access Memory	Match memory read or write
Input I/O	Match I/O input
Output I/O	Match I/O Output

Both I/O	Match I/O input or output
Fetch Instruction	Match instruction fetch

The Access Memory option is a combination of the Read Memory and Write Memory options—it matches either memory reads or writes. Likewise, the Both I/O option matches I/O reads or writes.

Some hardware debuggers are capable of distinguishing between simple data reads from memory and instruction fetches. In this case, if you set a breakpoint to match on read memory, an instruction fetch from that location will not trigger the hardware breakpoint. Instruction cycles include all the bytes that the processor reads in order to determine the instruction operation to perform, including prefix bytes, operand addresses, and immediate values. The actual data read or written to memory referenced by an operand's address is not considered to be part of the instruction fetch. For example:

```
MOV    AX,[1234]
```

fetches 3 instruction bytes from memory, and reads 2 data bytes. If you use instruction fetch matching, remember that the 80x86 processor family prefetches instructions to be executed, so you may get false matches, depending on whether your hardware debugger can sort out prefetched instructions from ones that are really executed.

The Address option of the Hardware menu has the following settings:

Above	Match above an address
Below	Match below an address
Range	Match within address range
Not Range	Match outside address range
Less or Equal	Match below or equal to address
Greater or Equal	Match above or equal to address
Equal	Match a single address
Unequal	Match all but a single address
Match All	Match any address

The Data command of the Hardware menu has the following settings:

Above	Match above a value
Below	Match below a value
Range	Match within value range
Not Range	Match outside value range
Less or Equal	Match below or equal to value
Greater or Equal	Match above or equal to value
Equal	Match a single value
Unequal	Match all but a single value
Match All	Match any value

If you choose a Data or Address option that involves any less-than or greater-than condition, a single address match range either starts at zero and extends to the value you specified, or starts at the value you specified and extends to the highest allowed value for addresses or data.

Hardware Debugger Overview

The device driver interface provides device-independent access to the capabilities of different hardware debuggers. To accomplish this, the common features of several hardware debuggers have been combined into one generic hardware debugger. Turbo Debugger then uses this abstract model when making requests to the device driver. Depending on the capabilities of a particular board, it may not be able to support all the operations specified by the abstract interface. In this case, the device driver can inform Turbo Debugger that a requested operation cannot be performed. A hardware board may also offer more capabilities than the abstract interface defines. In this case, Turbo Debugger can't make use of the added features of the board.

Since we expect the device driver interface to encompass new features in future releases, we have defined an "implementation level" status field that the device driver returns when requested. This lets Turbo Debugger know what the device driver is capable of doing, and provides compatiblity with older drivers, while allowing new drivers to take advantage of capabilities in future releases of the interface.

The hardware debugger interface breaks the capabilities of debugger boards into three main areas of functionality:

1. Memory and I/O access breakpoints
2. Instruction trace-back memory
3. Extra onboard memory for symbol tables

This version of the interface supports only the first category. Future releases will define an interface that accesses the other features. When you write a device driver, keep in mind that these other capabilities will be supported at a later date.

Device Driver Interface

The device driver is an ordinary character-type device driver named TDH386.SYS. You must put the following statement in your CONFIG.SYS file in order for the driver to be loaded when you boot the system:

```
DEVICE = drvrname.ext
```

where *drvrname.ext* is the name of the device driver file you have created.

The device driver must support the following function calls.

INIT Command code = 0

Called once when the device driver is first loaded. Your code for this function must make sure that the hardware board is disabled and in a quiescent state.

READ Command code = 4

Called by Turbo Debugger to read the status block from the last command sent to the device driver. You should keep the last status in a data area inside the driver, and return as many bytes as requested. Each time a read is issued, you must start sending from the beginning of the status block, even if the previous read request was not long enough to send the entire block.

The next section describes the various status blocks the device driver can return in response to different command blocks.

READNOWAIT Command code = 5

Returns the first byte of the status block. The busy bit should always be set to 0, indicating that data is available at all times.

READSTATUS Command code = 6

Always sets the busy bit to 0, indicating that a subsequent read request would complete immediately.

READFLUSH Command code = 7

Simply sets the done bit in the return status.

WRITE Command code = 8

Called by Turbo Debugger to send a command to the device driver. The command will have a variable length, depending on the command type. You can either copy the data into a work area inside the device driver, or you can access it directly using the data pointer that is part of the device driver request.

The next section describes the various command blocks Turbo Debugger can send to the device driver.

WRITEVERIFY Command code = 9

Does the same thing as **WRITE** (command code 8).

WRITESTATUS Command code = 10

Simply sets the done bit in the return status.

WRITEFLUSH Command code = 11

Simply sets the done bit in the return status.

All other function calls should set the error bit (bit 15) in the return status word, and put an "Unknown command" error code (3) in the low byte of the status word.

Command Blocks Sent to Device Driver

All command blocks sent to the device driver by the **WRITE** function call start with a byte that describes the operation to perform. The subsequent bytes provide additional information for the particular command.

When setting a hardware breakpoint, it is the device driver's responsibility to check that it has been handed an acceptable parameter block. It can't just ignore fields that request an operation it can't perform. You must check each field to make sure that the hardware can support the requested operation, and if it can't, you must set the appropriate return code.

The first byte can contain one of the following command codes:

- **0—Installs vectors**
- **1—Gets hardware capabilities**
- **2—Enables the hardware breakpoints**
- **3—Disables the hardware breakpoints**
- **4—Sets a hardware breakpoint**
- **5—Clears a hardware breakpoint**
- **6—Sets I/O base address, resets hardware**
- **7—Removes vectors**

The following commands send additional data after the command code.

Install vectors (code 0)

4 bytes *Far pointer to vector routine*

This is a 32-bit pointer, the first word being the offset, and the second being the segment. You must save this address so that the device driver can jump to it when a hardware breakpoint occurs. This routine should also install any interrupt vectors that the device driver needs. Turbo Debugger calls this routine once when it first starts up. The Remove Vectors (code 10) function is called once by Turbo Debugger when it no longer needs to use the hardware debugger device driver. At this time, you should replace any vectors that you took over when function 0 was called and make sure the hardware is disabled.

Set a hardware breakpoint (code 4)

1 byte *Breakpoint type*

0	Memory read
1	Memory write
2	Memory read or write
3	I/O read
4	I/O write
5	I/O read or write
6	Instruction fetch

1 byte *Address matching mode*

0	Match any address, don't care
1	Equal to test value
2	Not equal to test value
3	Above test value
4	Below test value
5	Below or equal to test value

6 Above or equal to test value
7 Within inclusive range
8 Outside range

4 bytes *Low address*

A memory address, in 32-bit linear form. If the address-matching mode requires one or more addresses to test against, the only or low value comes here.

4 bytes *High address*

A memory address, in 32-bit linear form. If the address matching mode requires two addresses to test against a range, the second and higher value comes here.

2 bytes *Pass count*

1 byte *Data-matching size,* 1 = byte, 2 = word, 4 = doubleword

1 byte *Source of matched bus cycle:*

1 CPU
2 DMA
3 Either CPU or DMA

1 byte *Data-matching mode*

0 Match any data, don't care
1 Equal to test value
2 Not equal to test value
3 Above test value
4 Below test value
5 Below or equal to test value
6 Above or equal to test value
7 Within inclusive range
8 Outside range

4 bytes *Low data value*

If the data-matching mode requires one or more data values, this field supplies the first or only value. The data-matching size determines how many bytes of this field are significant.

4 bytes *High data value*

If the data-matching mode requires two data values, this field supplies the second value. The data-matching size determines how many bytes of this field are significant.

4 bytes *Data mask*

If the hardware supports it, this field controls which bits in the data are examined for the match condition.

Clear a hardware breakpoint (code 5)

1 byte The handle of the breakpoint to clear. The handle was given to Turbo Debugger by the Set Hardware Breakpoint command (code 4).

Set I/O board base address (code 9)

2 bytes The base address of the hardware debugger board.

Status Blocks Returned by Device Driver

The READ function call returns the status block from the device driver. Different commands written to the device driver result in various status blocks being built to report on what happened. All the status blocks start with a single byte that describes the overall result of the requested operation. The subsequent bytes return additional information particular to the command that generated the status block. The following status codes can be returned in the first byte:

0 **Command was successful.**

1 **Invalid handle supplied.**

2 **Full, can't set any more breakpoints.**

3 **Breakpoint was too complex for the hardware.** The breakpoint could never be set; the hardware is not capable of supporting the combination of bus cycle, address, and data-matching that Turbo Debugger requested.

4 **Command can't be performed due to restrictions imposed by a previous command.** The command could have been performed if it weren't for some previous operation preventing it. This could happen, for example, if the hardware only permits a single data match value, but Turbo Debugger tries to set a second hardware breakpoint with a different data match value than the first breakpoint.

5 **The device driver can't find the hardware board.**

6 **A hardware failure has occurred.**

7 **An invalid command code was sent to the driver.**

8 **The driver has not been initialized with function code 0, so nothing can be done yet.**

The following commands return additional status information after the status code byte:

Get hardware capabilities (code 1)

2 bytes *Device driver interface (this specification) version number.* The current version is 1, subsequent versions will increase this number.

2 bytes *Device driver software version number.* For each released version of your device driver that behaves differently, this field should contain a different number. This lets Turbo Debugger take special measures if necessary, based on this field.

1 byte *Maximum number of hardware breakpoints that this driver/board combination can support.*

1 byte *Configuration bits*

Bit	**Function**
0	1 for can distinguish between CPU and DMA accesses.
1	1 for can detect DMA transfers
2	1 for has data mask
3	1 for breakpoints have hardware pass counter
4	1 for can match on data as well as address

1 byte *Breakpoint types supported (bit mask)*

Bit	**Function**
0	Memory read
1	Memory write
2	Memory read or write
3	I/O read
4	I/O write
5	I/O read or write
6	Instruction fetch

2 bytes *Addressing match modes supported (bit mask)*

Bit	Function
0	Match any address, don't care
1	Equal to test value
2	Not equal to test value
3	Above test value
4	Below test value
5	Below or equal to test value
6	Above or equal to test value
7	Within inclusive range
8	Outside range

2 bytes *Data match modes supported (bit mask)*

Bit	Function
0	Match any data, don't care
1	Equal to test value
2	Not equal to test value
3	Above test value
4	Below test value
5	Below or equal to test value
6	Above or equal to test value
7	Within inclusive range
8	Outside range

1 byte *Maximum data match length*

Set to 1, 2, or 4 depending on the widest data match or mask that the hardware can perform.

2 bytes Size of onboard memory in Kbytes.

2 bytes Maximum number of trace-back events that can be recalled.

2 bytes *Address of hardware breakpoint enable byte*

Specifies the segment address where Turbo Debugger must write a byte with a value of 1 to enable hardware breakpoints. The field must contain 0 if the device driver does not or cannot support this capability. If it is supported, this byte allows Turbo Debugger to inform the device driver that it has finished writing things to the address space of the program being debugged, and that subsequent accesses can cause hardware breakpoints.

Set a hardware breakpoint (code 4)

1 byte	A handle that Turbo Debugger will use to refer to this breakpoint in the future. The device driver also uses this handle when calling back into Turbo Debugger after a hardware breakpoint has occurred. The handle must be greater than or equal to zero (0). Negative values (top bit on) indicate a special condition when the device driver calls Turbo Debugger with a hardware breakpoint.

Recursive entry (code –2)

1 byte	The special value FE (hex) can be returned by the hardware device driver if it has been recursively entered while processing a hardware breakpoint. This can happen if a hardware breakpoint has been set in the 6 bytes below the current top of stack in the program being debugged. If Turbo Debugger receives this entry code, it displays a message that the device driver can't proceed because of a breakpoint being set near the top of the stack.

Device Driver Call into Turbo Debugger

When the hardware board and the device driver software have determined that a hardware breakpoint has occurred, control must be transferred to the address inside Turbo Debugger that was specified with command code 0 (Set hardware breakpoint vector).

The vector address must be jumped to with the CPU state exactly as it was when the hardware breakpoint occurred, but with the program's AX register pushed on the stack, and an entry code now in the AH register. The entry code can be

>= 0	The handle of the triggered breakpoint
–1 (FF)	The breakpoint button was pressed

Turbo Debugger will never return to the device driver once it is jumped into from a hardware breakpoint.

APPENDIX

G

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. The TDREMOTE program 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.

If you're experiencing memory problems debugging a program and your system has EMS memory, make sure you're using it for symbol tables. Usually, Turbo Debugger will use any EMS memory it finds for symbol tables. You can use the installation utility (TDINST) to control whether Turbo Debugger uses EMS for symbol tables.

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.

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 Appendix B.

You can, if you wish, 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 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 wish 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 TD from DOS:

- For serial port COM1: `-rs3`
- For serial port COM2: `-rp2 -rs3`

When the link is successfully started, the message "Turbo Debugger online" will appear on the remote system, and the message "TDREMOTE online" will appear on the Turbo Debugger screen. This will be quickly replaced with Turbo Debugger's normal window display.

Notice that both TD and TDREMOTE use the same command-line options to set the speed and serial port. Both TD 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 disk. 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 and/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 (/).

–?	Display a help screen
–h	Display a help screen
–rs1	Slow speed, 9600 baud
–rs2	Medium speed, 40,000 baud
–rs3	High speed, 115,000 baud (default)
–rp1	Port 1, (COM1) (default)
–rp2	Port 2, (COM2)
–w	Write 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 by using the **–w** option.

You can make the TDREMOTE command-line options permanent by writing them back into the TDREMOTE executable program image on disk. Do this by specifying the **–w** command-line option along with the other options that you wish to make permanent. You will then be 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.EXE, enter the program name followed by -h:

```
TDREMOTE -h
```

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 are running on a version 2 of DOS, you will have to supply the full path and file name of the executable program.

Remote Debugging Sessions

Once you've started TDREMOTE and TD in remote mode, you debug your program much as if you were doing it on a single system. Turbo Debugger commands work exactly as you are used to; 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 *F10*/View/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 wish to send files over to the remote system while you are running Turbo Debugger, you can go to DOS using the *F10*/File/OS 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 operation is described in Appendix B.

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 succesfully 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 version 2 of DOS, 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 commmmunicate 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 in the program has become corrupted or truncated. You should delete the program

file from disk. This will 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 since 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 receivedan 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 hookups. Next, try running the link at the slowest speed by using the **–rs1** command-line option when starting up both TDREMOTE and TD. 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.

APPENDIX

H

Prompts and Error Messages

Turbo Debugger displays error messages and prompts at the current cursor location.

You can alter how you respond to error messages with a command-line option. You can also set the option permanently by using the TDINST configuration program described in Appendix E.

You can alter some aspects of error messages (for example, whether they cause an audible beep) either for one debugging session or permanently. To alter the error messages for one session, use the command-line options; to alter the error messages permanently, use TDINST.

This chapter describes the prompts and error and information messages Turbo Debugger generates.

We'll tell you how to respond to both prompts and error messages. All the prompts and error messages (including the startup fatal error messages) are listed in alphabetical order, with a description provided for each one.

Prompts

Turbo Debugger displays a prompt when you must supply additional information to complete a command. The title of the prompt describes the information that's needed. The contents may show a history list (previous responses) that you have given to this prompt.

You can respond to a prompt in one of two ways:

- Enter a response and accept it by pressing *Enter.*
- Press *Esc* to cancel the prompt and return to the command menu that preceded the prompt.

Some prompts only present a choice between two items (like yes/no). You can use the arrow keys 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 prompt is active, refer to Chapter 2.

Here's a listing of all the prompts in alphabetical order.

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 prompt is initialized to the currently highlighted watch expression.

Enter address, count, byte value
Enter the address of the block of memory you wish to set to a particular byte value, then the number of bytes you wish to set, followed by the value to fill the block with.

Enter animate delay (10ths of sec)
Specify how fast you want the Animate command to proceed. The higher the number, the longer between sucessive steps during animation.

Enter breakpoint pass count
Enter the number of times you want the breakpoint to be passed over before finally being triggered. If you don't enter a pass count, a default

value of 1 is used, which triggers the breakpoint the first time it is encountered.

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 viewer. 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 wish 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 execute
Enter an expression that will be executed each time the breakpoint is triggered. In order to be useful, the expression should have some type of side effect, such as executing a procedure or function in your program or changing the value of a variable.

Enter expression to log
Enter an expression whose value will be logged in the Log viewer each time the breakpoint is triggered.

Enter go to label or address
Enter the address you wish to view in your program. You can enter a function name, a line number, an absolute address, or a memory pointer expression.

Enter go to line number
Enter the line number you wish to see in the current module. If you enter a line number that is past the end of the file, you will 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 viewer.

Enter go to offset
You are viewing a disk file as hex data bytes. Enter the offset from the start of the file where you wish to view the data bytes. The file will be positioned at the line that contains the offset you specified.

Enter inspect start index, range
Enter the index of the first item in the array you wish to view, followed by the number of items you wish 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 D has a condensed listing of all assembler keywords and Chapter 10 discusses the assembler language in more detail.

Enter log file
Enter the name of the file you wish to write the log to. Until you issue a Close log file command, all lines sent to the log will be written to the file as well as displayed in the window. The default file name has the extension .LOG and is the same file name as the program you are debugging. You can accept this name by pressing *Enter* or type a new name instead.

Enter memory address
Enter a single memory address. 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 processor 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 and/or directory name that you want to become the current drive and directory.

Enter new file mask
Enter a DOS-style wildcard file specification that matches the files you wish to see in the file list. You can enter a new disk letter or directory path as part of the specification.

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 wish to read from; valid port numbers are from 0 to 65535.

Enter port number, value to output
Enter the I/O port number you wish to write to, and the value to write; separate the two expressions with a comma. Valid port numbers are from 0 to 65535.

Enter position address
Enter an address in your program where you wish to view the code or data. See Chapter 9 for more information on entering addresses.

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 from 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 run to code address
Enter the address in your program where you wish execution to stop. See Chapter 9 for more information on entering addresses.

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 expression
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 0 or more of any character, and ? to match any single character.

Enter search instruction or bytes
Enter an instruction, as you would for the Assemble local menu command, or enter a byte list as you would for a Search command in a data pane.

Enter set breakpoint at code address
Enter the address in your program where you wish to set a breakpoint. See Chapter 9 for more information on entering addresses.

Enter source address, destination, count
Enter the address of the block you wish 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 list
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 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 window or Module window.

Enter variable to inspect
Enter the name of a variable or expression whose contents you wish to examine. If the prompt is initialized from a text pane, you can accept the entry by pressing *Enter* or change it and enter something else entirely.

Enter variable to range inspect
Enter the name of a variable or expression whose contents you wish to examine as an array. If the prompt is initialized from a text pane, you can accept the entry by pressing *Enter* or change it and enter something else entirely.

Enter watch expression
Enter a variable name or expression whose value you wish to watch in the Watches viewer. If you wish, you can enter an expression that does not refer to a memory location, such as `x * y + 4`. If the prompt is initialized from a Text pane, you can accept the entry by pressing *Enter* or change it and enter something else entirely.

Enter write file name
Enter the name of the file you wish to write the block of memory to.

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 wish to overrwrite the existing macro, press *Y*; otherwise, press *N* to cancel the command.

Pick a module
Select a module name to view in the Module viewer. You are presented with a list of all the modules in your program. If you wish 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.

Press key for macro assign
Press the key that you want to assign the macro to. Then, press the keys to do the command sequence that you wish 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 for macro delete
Press the key for the macro that you wish 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 reply *Y*, your program will be reloaded. If you reply *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 wish 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 answer yes, 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, 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 both from the DOS command line when you start Turbo Debugger or from the installation program (TDINST).

When an error message is displayed, the next key that's pressed will clear the error from the screen. However, the key you press is also taken as the next command, so the message simply goes away when you do something next. You can configure the debugger so that pressing *Esc* clears the error. You can do this from the DOS command line, or set it permanently from within the installation 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 the 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 the debugger from the DOS prompt.

Bad configuration file
The configuration file is either corrupted, not a Turbo Debugger configuration file, or is an out-of-date configuration file for a different version of the debugger.

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 memory 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 processing your command.

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 G, 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 TD and TDREMOTE together.

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 : to separate the second and third expressions was found.

For example,

```
x < 0 ? 4 6
```

should have been

```
x < 0 ? 4 : 6
```

']' expected
While evaluating an expression, a left bracket ([) starting an array index expression was encountered without a matching right bracket (]) to end the index expression.

For example,

```
table[4
```

should have been

```
table[4]
```

This error can also occur when entering an assembler instruction using the built-in assembler. In this case, a left bracket was encountered that introduced a base or index register memory access and there was no corresponding right bracket.

For example,

```
mov ax,4[si
```

should have been

```
mov ax,4[si]
```

Already logging to a file
You issued an Open log file command after having already issued the same command without an intervening Close log file command. If you wish to log to a different file, first close the current log by issuing the Close log file command.

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.

Can't change that symbol
You tried to change a symbol that can't be changed. The only symbols that can be changed directly are scalars (int, long, etc. in C; byte, integer, longint, etc. in Pascal) and pointers. If you wish 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 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, non-existent memory, or in segment 0. The only way to view a program executing in ROM is to use the **Run/Trace** 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 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/OS** Shell menu command and the Edit command 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 viewer 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.

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 viewer, 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 Log To 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 progam, 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/Load 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 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 sort of an expression whenever Turbo Debugger might need to repeatedly evaluate an expression, such as when it is in an Inspector 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 be

```
add ax,45
```

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.

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 cast syntax
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 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 string after an expression, but it is not a valid format string. See Chapter 9 for a description of format strings.

Invalid function parameter
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 command 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, attempting 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, 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 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 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 viewer using the View/Numeric Processor command from the main menu bar, but there is no numeric processor chip installed on your system, nor does the program you're debugging use the software emulator.

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 the *F1* key 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 allow you to 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 327 for more information on symbol tables.

No previous search expression
You attempted to perform a Next command from the local menu of a Text pane, but you had not previously issued a Search command to specify what to search for. You can only use Next after issuing a Search command in a pane.

No program loaded
You attempted to issue a command that requires a program to be loaded. There are many commands that can only be issued when a program is loaded. For example, none of the commands in the Run main menu can be performed without having a program loaded. Use the File/Load command to load a program before issuing these commands.

No source file for module ___
No source file can be found for the module you wish to view. If the source file is not in the current directory, you can use the Options/Code Directories 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
You 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 Expression/Inspect command from the main menu bar 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 a structure 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 the debugger is set to use it for the symbol table. You can use the **–se** command-line option to do this or set it permanently using TDINST.

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 G for more information on how to do this.

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 wish to remove them and then restart the 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.

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 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
(Pascal only) You've tried to set a pane in the CPU viewer to a location in your program that is not presently loaded into memory. You can use a Module viewer 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 prompt first appears. The names in the list that end with a backslash (\) are directories, allowing you to move up and down the directory tree through the lists.

Press Esc
This message appears as part of any error message when you have chosen to reply to errors by explicitly pressing a key. Pressing the *Esc* key causes the error message to disappear.

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 viewer 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 Assembler, you must link your program with TLINK, using the **/v** option, in order to get debug symbol information.

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 as negative address
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 wish to search the entire file, press the *Ctrl-PgUp* key combination 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.

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/OS 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 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.

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.

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). The 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 the *Ctrl-Break* key combination.

APPENDIX

I

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 Options/ Compiler/Optimization command with the Turbo C integrated development 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 will show 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 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, you can do this by pressing *F7* to trace into a function, then put the cursor on the closing *}* at the end of the function and press *F4* to run to that point. Then, choose the *F10*/**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 and 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 *F10*/Data/Watch and *F10*/Data/Evaluate 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 see use a CPU window and see your source code as well. To do this, open a CPU window, and then choose the Code pane's Mixed 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 *F10*/Data/Evaluate/Modify command. You can enter an assignment expression that directly modifies a register's contents, for example:

```
SI=99
```

will load the SI register with 99.

Likewise, you can examine registers using the same technique, for example:

```
Alt-D E AX
```

will show you the value of the AX register.

Turbo Pascal Tips

Stepping through Initialization Code

When you first load your program into the 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 shortcut 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 shortcut 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 you program terminates, control is passed down a chain of exit procedures (refer to the chapter titled "Inside Turbo Pascal" in the *Turbo Pascal Owner's Handbook*). When you step past the **end** of the main program, the 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 wish to debug.

Constants

Constant identifiers are only recognized 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) and *Digits* (a typed constant), but you can't inspect *B* (a floating point constant), or *Message* or *Caps* (string and set constants).

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, etc.) 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.

APPENDIX

J

Glossary

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 gets done 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 top window, with the 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: The human-readable form of machine instructions. The Code pane of a CPU window allows you to assemble instructions directly into memory.

auto-variable: 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.*

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.

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.

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 prompt a window can issue. This allows you to select a previous entry from a history list.

inspector: A window used to examine or change the values in a data element, array, or structure.

local menu: See *pop-up menu.*

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 first letter of a main menu 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 (pop-up) menu of commands.

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: The menu of commands that apply only to a particular pane in a window. Press *Alt-F10* to pop up the local menu for the current pane. Also called "local menu."

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: The menu of commands that apply to all windows. Also known as "global menus."

record: See *structure.*

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 allows you to 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 and * 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.*

Index

USER'S GUIDE

TURBO DEBUGGER®

BORLAND

BORLAND INTERNATIONAL, INC., 1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95066-0001 PART# 15MN-ASD03-10 BOR 0847