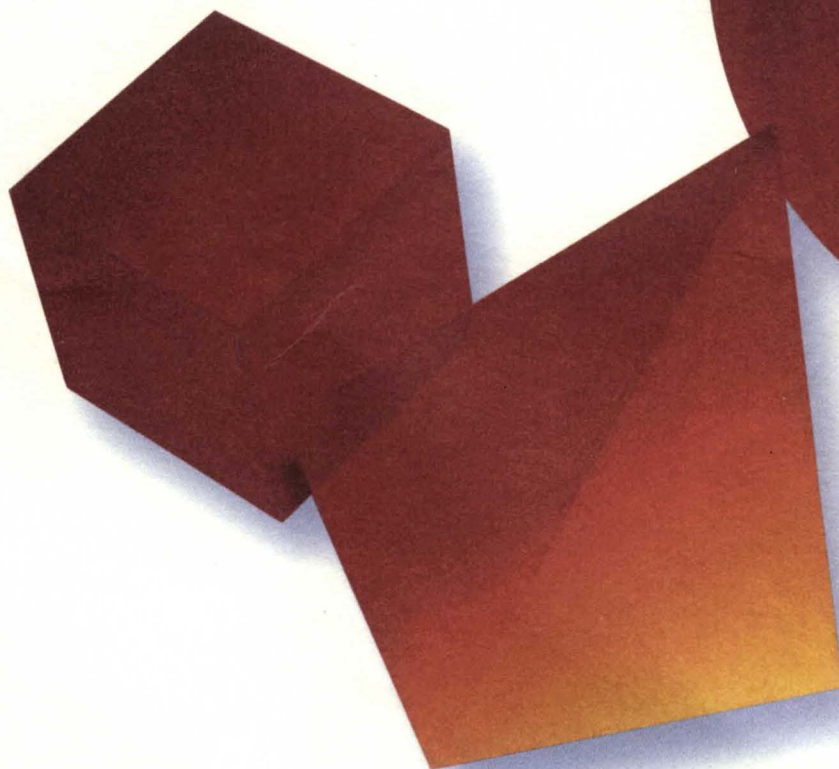User's Guide

Borland®
Turbo Debugger®

User's Guide

VERSION 4.5

Borland®
Turbo Debugger®

Borland International, Inc., 100 Borland Way
P.O. Box 660001, Scotts Valley, CA 95067-0001

Contents

Chapter 4
Debugging a simple example   51

Chapter 5
Setting and using breakpoints   67

Chapter 9
Assembly-level debugging   115

Chapter 10
Windows debugging features   129

Chapter 11
Debugging object-oriented programs 147

Tables

Figures

Introduction

Turbo Debugger is a set of tools designed to help you debug the programs you write with Borland's line of compilers. The Turbo Debugger package consists of a set of executable files, utilities, online text files, example programs, and this manual.

Turbo Debugger lets you debug the programs you're writing for Win16, Win32, and DOS. When you load your program into Turbo Debugger, you can use the debugger to control your program's execution and to view the different aspects of your program (including your program's output, source code, data structures, and program values) as it runs.

Turbo Debugger uses menus, multiple windows, dialog boxes, and online context-sensitive help system to provide you with an easy-to-use, interactive debugging environment. In addition, Turbo Debugger provides a comprehensive set of debugging features:

- Full C, C++, Object Pascal, and assembler expression evaluation.
- Full program execution control, including program animation.
- Low-level access to the CPU registers and system memory.
- Complete data inspection capabilities.
- Powerful breakpoint and logging facilities.
- Windows message tracking, including breakpoints on window messages.
- Full object-oriented programming support, including class browsing and object inspecting.
- Reverse execution.
- Remote debugging support.
- Macro recording of keystrokes to speed up repeated series of commands.
- Copying and pasting between windows and dialog boxes.
- Incremental matching, automatic name completion, and select-by-typing (to minimize keyboard entries).

- Context-sensitive SpeedMenus throughout the product.
- Dialog boxes that let you customize the debugger's options.

New features and changes for version 4.x

Turbo Debugger 4.x provides the following enhancements over version 3.x:

- Ability to debug both 16- and 32-bit Windows programs (debugging Win32 programs is provided with the addition of TD32, the 32-bit debugger).
- Ability to debug larger programs.
- Support for remote debugging on Windows systems (described in Appendix B).
- Operating-system exception handling (described in section "Tracking operating-system exceptions" on page 142).
- C++ and C exception handling (described in section "Exceptions" on page 152).
- Session-state saving (described in section "Restarting a debugging session" on page 22).
- Thread support for multithreaded Windows NT programs (described in section "Debugging multithreaded programs" on page 139).
- Ability to attach to processes that are already running in Windows NT (described in section "The Attach command" on page 46).
- Ability to shell out to a selected editor while running Windows NT (described in section "Edit" on page 111).
- Ability to choose a Windows international sort order for items displayed in Turbo Debugger (use Turbo Debugger's configuration programs to access this feature).

Hardware requirements

Turbo Debugger's hardware requirements are the same as those of your Borland language compiler.

In addition, Turbo Debugger supports the following graphics modes and adapters: CGA, EGA, VGA, Hercules monochrome-graphics, Super VGA (SVGA), TIGA, and 8514. You can use standard drivers with everything except SVGA, TIGA, and 8514.

Terminology in this manual

For convenience and brevity, several terms in this manual are used in slightly more generic ways than usual:

Argument

The term *argument* is used interchangeably with *parameter* in this manual and applies both to command-line arguments used to invoke a program to be debugged and to arguments passed to routines, functions, and procedures.

Module

This term refers to what is usually called a module in C++ and assembler, and also to what is called a *unit* in Pascal. Modules are executable files such as .EXE files and .DLLs.

Routine

A *routine*, as used in this manual, refers to assembler and C++ functions, and to Pascal functions, procedures, and object methods.

In this manual, the term "Turbo Debugger" refers to the Turbo Debugger programs TD.EXE, TDW.EXE, and TD32.EXE. However, there are times when the text refers to a specific Turbo Debugger program. In these cases, the term "TDW" refers to TDW.EXE, "TD32" refers to TD32.EXE, and "TD" refers to TD.EXE.

Typographic and icon conventions

Boldface	Boldface type indicates language keywords (such as **char**, **switch**, and **begin**) and command-line options (such as **–rn**).
Italics	Italic type indicates program variables and constants that appear in text. This typeface is also used to emphasize certain words, such as new terms.
Monospace	Monospace type represents text as it appears onscreen or in a program. It is also used for anything you must type literally (such as TDW to start up Turbo Debugger for Windows).
Key1	This typeface indicates a key on your keyboard. For example, "Press *Esc* to exit a menu."
Key1+Key2	Key combinations produced by holding down one or more keys simultaneously are represented as *Key1+Key2*. For example, you can execute the Program Reset command by holding down the *Ctrl* key and pressing *F2* (which is represented as *Ctrl+F2*).
Menu| Command	This command sequence represents a choice from the menu bar followed by a menu choice. For example, the command "File|Open" represents the Open command on the File menu.
Note	This indicates material you should take special notice of.
Screen shots	Unless otherwise noted, all screen shots in this manual depict TD32 while running under Windows NT.

This manual also uses the following icons to indicate sections that pertain to specific Windows operating environments:

`TDW`	Windows 3.x	`TD32`	Windows 32s *and* NT
`32s`	Windows 32s	`NT`	Windows NT

Using this manual

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

Chapter 1, "Installing and configuring Turbo Debugger" describes the files that are installed with the Turbo Debugger package and how to customize Turbo Debugger once it is installed.

Chapter 2, "Starting Turbo Debugger and running your program" describes how to prepare your program for debugging, and how to run Turbo Debugger and load your program. This chapter also discusses the different ways to control your program's execution while you are running it in Turbo Debugger.

Chapter 3, "Debugging with Turbo Debugger" introduces you to Turbo Debugger's environment—its global and SpeedMenu system, dialog boxes, and debugging windows. This chapter also discusses the basics of debugging, and the special features that Turbo Debugger provides to make your debugging session run smoothly.

Chapter 4, "Debugging a simple example" leads you through a sample debugging session that demonstrates the capabilities of Turbo Debugger.

Chapter 5, "Setting and using breakpoints" describes Turbo Debugger's breakpoint capability.

Chapter 6, "Examining and modifying data" explains the various ways you can examine and modify the data used by your program.

Chapter 7, "Evaluating expressions" describes the types of expressions that Turbo Debugger accepts, how to specify a display format of the expression results, and how to override the scope in expressions.

Chapter 8, "Examining disk files" describes how to examine program source files, and how to examine other disk files in either a text or binary format.

Chapter 9, "Assembly-level debugging" describes Turbo Debugger's CPU window. Additional information about this window and about assembler-level debugging is in the file TD_ASM.TXT.

Chapter 10, "Windows debugging features" describes the Turbo Debugger features you can use to debug Windows programs.

Chapter 11, **"Debugging object-oriented programs"** explains Turbo Debugger's special features that let you examine object oriented classes and objects.

Chapter 12, **"Debugging TSRs and device drivers"** describes how to use TD.EXE to debug terminate and stay resident (TSR) programs and DOS device drivers.

Appendix A, **"Command-line options"** describes all the command-line options that are available with Turbo Debugger.

Appendix B, **"Remote debugging"** describes the remote debugging capabilities of Turbo Debugger.

Appendix C, **"Turbo Debugger error messages"** lists all the error messages and prompts generated by Turbo Debugger. The list also gives suggestions on how to respond to the prompts and error messages.

Where to now?

The following reading guidelines are proposed to help first-time and experienced Turbo Debugger users.

First-time Turbo Debugger users

New Turbo Debugger users should read the first four chapters of this manual to get a basic understanding of how the debugger works. Once you become familiar with the basics of Turbo Debugger, read Chapters 5, 6, and 7 to become proficient with the debugger's most-often used features: breakpoints, data inspection, and expression evaluation.

The remaining chapters in the book provide information about specific debugger features (such as the CPU window), and provide help when you encounter problems debugging a specific area of your program (such as with an object-oriented class or a Windows DLL). Browse through these chapters to get an overview of the more advanced debugger features.

If, while using Turbo Debugger, you have questions about a certain feature or menu command, press *F1* to access the debugger's context-sensitive help system.

Experienced Turbo Debugger users

Users familiar with Turbo Debugger should read the "New features and changes for version 4.x" section on page 2 to get an overview of items new to this release. Experienced users should also read Chapter 2, "Starting Turbo Debugger and running your program," which lists the files installed with Turbo Debugger. Experienced users should also read "Turbo Debugger's special features" on page 41, which describes the features that make Turbo Debugger especially easy to use. Even experienced Turbo Debugger users might be surprised at some of the features they've previously overlooked.

Software registration and technical support

The Borland Assist program offers a range of technical support plans to fit the different needs of individuals, consultants, large corporations, and developers. To receive help with this product, send in the registration card and select the Borland Assist plan that best suits your needs. North American customers can register by phone 24 hours a day by calling 1-800-845-0147. For additional details on these and other Borland services, see the *Borland Assist Support and Services Guide* included with this product.

Installing and configuring Turbo Debugger

This chapter describes how to install Turbo Debugger and how to customize its default options and display settings. Also described in this chapter are the many files that are installed with the debugger.

Installing Turbo Debugger

The INSTALL.EXE program supplied with your Borland compiler installs the entire Turbo Debugger package, which includes executable files, configuration files, utilities, online text files, and example programs. A detailed listing of all files included with Turbo Debugger starts on page 12.

The install program creates icons for your Borland compiler and language tools, and places them inside a new Windows program group. Directions for using INSTALL.EXE can be found in the *User's Guide* of your Borland language product.

For general installation information, refer to the README file on your compiler's Installation disk. For a complete listing of the files installed by INSTALL.EXE, refer to the FILELIST.DOC text file (this file is copied by the installation program to your main language directory).

Configuring Turbo Debugger

You can configure Turbo Debugger's display options and program settings with customized configuration files and with the debugger's Options menu. Settings in the configuration files become effective when you load Turbo Debugger. To change the debugger's settings after you've loaded it, use the commands on the Options menu.

Turbo Debugger's configuration files

Turbo Debugger uses the following configuration, initialization, and session-state files when it starts:

- TDCONFIG.TD
- TDCONFIG.TDW
- TDCONFIG.TD2
- TDW.INI
- XXXX.TR
- XXXX.TRW
- XXXX.TR2

The configuration files TDCONFIG.TD, TDCONFIG.TDW, and TDCONFIG.TD2 are created and used by TD, TDW, and TD32, respectively. The settings in these files override the default configuration settings of the debuggers. You can modify the configuration files using the installation programs TDINST.EXE, TDWINST.EXE, and TD32INST.

TDW.INI is the initialization file used by TDW.EXE and TD32.EXE. It contains settings for the video driver used with Turbo Debugger, the location of TDWINTH.DLL (the Windows-debugging DLL), and the remote debugging settings you specify using WRSETUP.EXE.

The installation program places a copy of TDW.INI in the main Windows directory. In this copy of TDW.INI, the video driver setting ([VideoDLL]) is set to SVGA.DLL, and the DebuggerDLL setting indicates the path to TDWINTH.DLL. Refer to the online file TD_HELP!.TXT for a complete description of TDW.INI.

Files ending with .TR, .TRW, and .TR2 extensions contain the session-state settings for the debuggers. For information on session-state saving, refer to "Restarting a debugging session" on page 22.

Searching for configuration files

When you start Turbo Debugger, it looks for its configuration files in the following order:

1 In the current directory.

2 In the directory specified in the Turbo Directory setting of Turbo Debugger's installation program.

3 In the directory that contains the debugger's executable file.

If Turbo Debugger finds a configuration file, the settings in that file override any built-in defaults. If you supply any command-line options when you start Turbo Debugger, they override any corresponding default options or values specified in the configuration file.

Setting up the video DLLs

TDW and TD32 use different video DLLs to support the available types of video adapters and monitors. After you've installed Turbo Debugger, run the utility program TDWINI.EXE to help you select or modify the video DLL that's used with the debuggers.

By default, TDW and TD32 use the SVGA.DLL video driver, which supports most video adapters and monitors. For more information on the available video DLLs, refer to the entries for DUAL8514.DLL, STB.DLL, SVGA.DLL, and TDWGUI.DLL in Table 1.1 (which appears later in this chapter), and the online Help system provided with TDWINI.EXE.

Dual-monitor debugging

Turbo Debugger supports dual-monitor debugging with TD and TDW, and with TD32 running on Windows 32s.

To create a dual-monitor system, you need a color monitor and video adapter, and a monochrome monitor and video adapter. When you debug with two monitors, Turbo Debugger appears on the monochrome monitor, and Windows and the program you're debugging appears on the color monitor. The advantage of this system setup is that you can see your program's output while you're debugging it with Turbo Debugger.

Once your hardware setup is complete, use the **–do** command-line option to load TD or TDW in dual-monitor mode. For more information on command-line options, see Appendix A.

If you're running TD32 on Windows 32s, you must use Turbo Debugger's SVGA.DLL video DLL to have access to dual-monitor debugging. On this system, dual-monitor debugging is enabled when the following line is inserted in the TDW.INI file, under the heading [VideoOptions]:

```
mono=yes
```

To properly setup this video DLL, use the TDWINI.EXE utility.

The Options menu

The Options menu contains commands that let you set and adjust the parameters that control the overall appearance of Turbo Debugger.

```
Language...      Source
Macros                ▶
Display options...
Path for source...
Save options...
Restore options...
```
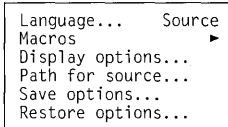
The Language command

Use the Options | Language command to select the programming language evaluator that the debugger uses. Chapter 7 describes how to set the expression evaluator and how it affects the way Turbo Debugger evaluates expressions.

Display Options command

The Option | Display Options command opens the Display Options dialog box. You use the settings in this dialog box to control the appearance of Turbo Debugger's display. While TD, TDW, and TD32 share most display options, TD32 has several additional options to provide support for the Windows NT multitasking operating system.

Figure 1.1   The Display Options dialog box
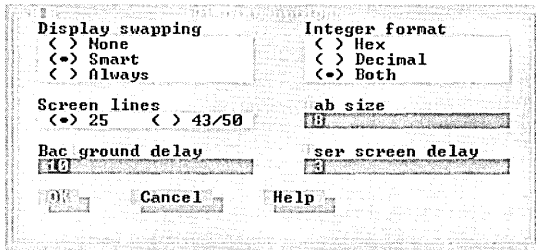
Display Swapping

You can use the Display Swapping radio buttons to control the way Turbo Debugger swaps your applications' screens with the debugger's windows.

None   Found in TD32 only, this option specifies that no screen-swapping is to take place. This option provides the fastest and smoothest screen updating when you're single-stepping through a program and should be used if you're using dual-monitor debugging. Beware, however, that this option can cause your display to become corrupted. If this happens, use the Repaint Desktop on the System menu to repaint the screen.

Smart   Turbo Debugger activates the user screen when it detects that your program is going to display output and when you step over routines.

Always   Turbo Debugger activates the user screen every time your program runs. Use this option if the Smart option isn't finding all the times the program writes to the screen. If you choose this option, the screen flickers every time you step through your program because Turbo Debugger's screen is replaced for a short time with the User screen.

Integer Format

The Integer Format radio buttons let you choose the way integers are displayed in Turbo Debugger.

Hex   Shows integers as hexadecimal numbers, displayed in a format appropriate to the current language evaluator.

Decimal   Displays integers in decimal notation.

Both   Displays integers in both decimal and hexadecimal notation (the hexadecimal numbers are placed in parentheses after the decimal value).

Screen Lines

Use the Screen Lines radio buttons to select either a 25-line display or a 43- or 50-line display available with EGA and VGA display adapters.

Tab Size

The Tab Size input box lets you set the number of columns each tab stop occupies. To see more text in source files that use tab indents, reduce the tab column width. You can set the tab column width from 1 to 32.

Background Delay

Found only in TD32.EXE, the Background Delay input box lets you specify how often Turbo Debugger's screens get updated. When you use this setting in conjunction with the Run | Wait for Child command, you can watch the effects of your program through Turbo Debugger's windows, while your program is running.

User Screen Delay

Found only in TD32, User Screen Delay lets you specify how long your program's screen is displayed when you press *Alt+F5* (the Window | User Screen command). This command is useful when you're using TD32 in full-screen mode, and you need to see your application's windows. By setting the delay, you can specify how long your program's screens will be displayed before Turbo Debugger regains control.

Path for Source command

Use the Path for Source command to specify the directories that Turbo Debugger searches for your program's source files. To enter multiple directories, separate each directory with a semicolon.

Although the Enter Source Directory Path input box holds a maximum of 256 characters, you can use a *response file* to specify longer search paths. A response file contains a single line that specifies the directories that should be searched for source code. Each directory listed must be separated by a semicolon. For example, a response file could contain the following line to specify three different search directories:

```
c:\my_proj\mod1\source;c:\my_proj\mod2\source;c:\my_proj\mod3\source
```

To specify a response file in the Enter Source Directory Path input box, enter an at-character (@) followed by the path and name of the response file. For example, the following entry specifies the SRC_PATH.TXT response file:

```
@C:\my_proj\src_path.txt
```

For more information on how Turbo Debugger conducts its search for source code, refer to "Searching for source code" on page 22.

Save Options command

The Save Options command opens a dialog box that lets you save your Option menu settings to a configuration file. You can save any or all of the following options:

Options   Saves all settings made in the Options menu.

Layout   Saves the current window layout and pane formats.

Macros   Saves the currently defined keyboard macros.

You can specify the name of the configuration file by using the Save To input box. By default, TDW.EXE uses the file name TDCONFIG.TDW, TD.EXE uses the file name TDCONFIG.TD, and TD32.EXE uses the file name TDCONFIG.TD2.

By creating different names for your configuration files, you can have a different debugger setup for each programming project you're working on. Each setup can specify unique macros, window layouts, source directories, and so on.

Restore Options command

The Restore Options command restores a configuration from a disk file. The file loaded must be a configuration file that was created with the Options | Save Options command or with Turbo Debugger's installation program (TDWINST for TDW.EXE and TDINST32 for TD32.EXE).

Debugging ObjectWindows 1.0 programs

If you're using TDW to debug a program that uses ObjectWindows 1.0x, you must configure the debugger so that it recognizes the ObjectWindows DDVT message dispatch system. To configure TDW,

1 Run TDWINST.
2 Choose Options | Source Debugging to access the Source Debugging dialog box.
3 Check the OWL 1.0X Window Messages check box.
4 Save the configuration and exit TDWINST.

Files installed with Turbo Debugger

The following tables list the files installed with Turbo Debugger, arranged into the following categories:

- Turbo Debugger's executable and support files
- Turbo Debugger's utilities
- Turbo Debugger's online text files
- Turbo Debugger's demonstration program

Turbo Debugger's executable and support files

Table 1.1 lists all the executable and support files needed to run TDW and TD32.

Table 1.1   Turbo Debugger's executable and support files

File name	Description
DUAL8514.DLL	Video DLL that supports dual-monitor debugging with 8514 monitors.
STB.DLL	Video DLL that supports video adapters produced by STB.
SVGA.DLL	Video driver that supports most adapters and monitors.
TD.EXE	Executable program used to debug DOS applications.
TDDEBUG.386	TDW.EXE uses the device driver TDDEBUG.386 to access the special debug registers of 80386 (and higher) processors. See page 74 for information on hardware debugging.
TDHELP.TDH	Help file for TD.EXE.
TDKBDW16.DLL	Support file used with Windows 32s.
TDKBDW32.DLL	Support file used with Windows 32s.
TDREMOTE.EXE	Driver used on remote system to support DOS remote debugging.
TD32.EXE	Executable program used to debug 32-bit programs written for Windows NT and Windows 32s.
TD32.ICO	Icon used with TD32.EXE.
TD32HELP.TDH	Help file for TD32.EXE.
TDVIDW16.DLL	Support file used with Windows 32s.
TDVIDW32.DLL	Support file used with Windows 32s.
TDW.EXE	Executable program used to debug 16-bit Windows programs.
TDW.INI	Initialization file used by TDW.EXE and TD32.EXE. This file is created by the install program and placed in your main Windows directory.
TDWGUI.DLL	Video DLL that places Turbo Debugger in a window while using TDW under Windows 3.x or while using TD32 under Windows 32s.
TDWHELP.TDH	Help file for TDW.EXE.
TDWINTH.DLL	Support DLL required by TDW.EXE. TDW.INI is set up to point to TDWINTH.DLL.
WREMOTE.EXE	Driver used on remote system to support Windows remote debugging.

Turbo Debugger's utilities

The Turbo Debugger package includes utilities to help with the debugging process. Table 1.2 lists the utilities and gives a general description of each one. For a more detailed description of these utilities, refer to the online text file TD_UTILS.TXT.

Table 1.2   Turbo Debugger's utilities

File name	Description
TDINST.EXE	Creates and modifies TD's configuration file, TDCONFIG.TD.
TDMEM.EXE	Displays the current availability of your computer's memory, including Expanded and Extended memory. Used for checking the programs and device drivers that are loaded, and the addresses that they're loaded into.
TDRF.EXE	File transfer utility used to transfer files to remote system.
TD32INST.EXE	Creates and modifies TD32's configuration file, TDCONFIG.TD2.

Table 1.2   Turbo Debugger's utilities (continued)

File name	Description
TD32INST.ICO	Icon used with TD32INST.EXE.
TDSTRIP.EXE	Strips Turbo Debugger's debugging information (the *symbol table*) from 16-bit .EXEs and .DLLs, without relinking.
TDSTRP32.EXE	Strips Turbo Debugger's debugging information (the *symbol table*) from 32-bit .EXEs and .DLLs, without relinking.
TDUMP.EXE	Displays the file structure of 16-bit and 32-bit .EXE, .DLL, and .OBJ files. Also displays the contents of the symbolic debug information.
TDWINI.EXE	Lets you change and customize Turbo Debugger's video DLL settings.
TDWINI.HLP	Windows help file for TDWINI.EXE.
TDWINST.EXE	Creates and modifies TDW's configuration file, TDCONFIG.TDW. Configures things like the display options and screen colors of TDW.
WRSETUP.EXE	Configuration file used to configure WREMOTE, the remote driver used with remote debugging.

Specifying utility command-line options

Each Turbo Debugger utility can be started using special command-line options. For a list of the command-line options available for the TDUMP, TDUMP32, and TDSTRIP utility programs, type the program name at the DOS command-line and press *Enter*. To see the command-line options for TDWINST and TDINST32, type the program name followed by -?, then press *Enter*. For example,

```
TDWINST -?
```

Turbo Debugger's online text files

The installation program places several text files in the DOC subdirectory of your main language directory.

Although you might not need to access all online files, it's important for you to look at TD_RDME.TXT, which contains last-minute information not available in the manual.

Table 1.3   Turbo Debugger's online files

File name	Description
TD_ASM.TXT	This file contains information about debugging Turbo Assembler programs. You might also find the information in this file helpful for debugging your inline assembler code.
	This file also contains information on using Turbo Debugger's Numeric Processor window.
TD_HELP!.TXT	Contains answers to commonly encountered problems. Among other things, TD_HELP!.TXT discusses the syntactic and parsing differences between Turbo Debugger and your language compiler, the TDW.INI file, debugging multi-language programs, and common questions and answers concerning Turbo Debugger.
TD_HDWBP.TXT	This file contains information on how to configure Turbo Debugger so that it takes advantage of the hardware debugging registers.
TD_RDME.TXT	Contains last-minute information not contained in the manual.
TD_UTILS.TXT	This file describes the command-line utilities included with Turbo Debugger.

All of Turbo Debugger's online files are unformatted ASCII files, so you can use your program editor to access them.

Turbo Debugger's example program

Chapter 4, "Debugging a simple example," introduces Turbo Debugger by showing how to debug a simple example program. The following table lists the files used in the example debugging chapter:

Table 1.4   Turbo Debugger's example program files

File name	Description
MAKEFILE	Makefile used with the buggy example program.
TDWDEMO.BUG	Buggy example program's source code.
TDWDEMO.H	Header file used by the example program.
TDWDEMO.ICO	Example program's icon.
TDWDEMO.IDE	IDE project file used with the example program.
TDWDEMO.RC	Resource file used with the example program.
S_PAINT.C	Example program's source code.
S_PAINT.EXE	Working example program in executable form.

Starting Turbo Debugger and running your program

A debugging session begins when you load your program into Turbo Debugger. After you load your program, you can run it under the debugger's control, pausing its execution at various places to look for where things have gone wrong. Before you can load your program into Turbo Debugger, however, you must prepare it for debugging.

This chapter describes

- Preparing programs for debugging
- Starting Turbo Debugger
- Loading your program into the debugger
- Controlling program execution
- Interrupting program execution
- Reverse execution
- Program termination
- Exiting Turbo Debugger

Preparing programs for debugging

When you're developing a program, whether it's written in C, C++, Pascal, or assembler, it's best to compile and link it with debug information. Although debugging information adds to the size of your executable files, it lets you see your program's source code and use its symbols to reference values while you're in the debugger.

Unless you have a very large project, it's usually best to compile your entire project with debug information turned on. This way, you'll have access to all your program modules from within the debugger. With larger projects, however, you might want to add debug information only to the modules you intend to load into the debugger.

While you're developing your program, compile your program without compiler optimizations. Even though optimizations create efficient programs, it can be confusing to debug the sections of code that have been optimized by the compiler; the object code that the compiler produces might not exactly match your program's source code. Because of this, you should compile your program with optimizations turned on only after you've fully debugged your program.

Once your program is fully debugged and ready for distribution, compile and link your program without debug information to reduce the size of your final program files.

Compiling from the integrated environment

If you're compiling your program from within the Borland C++ integrated environment, you must be sure to include symbolic debug information in both your .OBJ files and your final executable files.

To include debug information in your .OBJ and .EXE files,

1 Choose the Options | Project command to open the Project Options dialog box.
2 Choose the Compiler | Debugging topic to access the Debugging options.
3 Check Debug Information in OBJs.
4 Choose the Linker | General topics to access the linker options.
5 Check Include Debug Information.

Compiling from the command line

If you compile your programs with Borland's C command-line compiler, use the –v compiler directive to add debug information to each of your modules. Be sure to use the –v linker switch to include the debug information in your final executable files.

Starting Turbo Debugger

After you've compiled and linked your program with debug information, you can begin the debugging process by starting Turbo Debugger and loading your program into the debugger.

The following table describes the appropriate debugger to use for the application you've built:

Table 2.1   Turbo Debugger programs

Turbo Debugger program	Applications debugged
TD.EXE	16-bit DOS applications
TDW.EXE	16-bit Windows applications
TD32.EXE	32-bit Windows applications

While TD must be started from the DOS command line, TDW and TD32 can be started from the Windows locations listed in the following table:

Table 2.2   Starting Turbo Debugger

Starting location	Procedure
Windows	Open your Borland compiler's group from the Program Manager, and choose the TDW or TD32 icon.
The compiler's integrated environment	Choose Tools | Turbo Debugger to debug the program in the active Edit window.
Windows' Program Manager File | Run dialog box	From the Command input box, type TDW or TD32, followed by any command-line options.
Windows File Manager	Double-click either the TDW.EXE or TD32.EXE executable file icon from the directory containing Turbo Debugger.

Specifying Turbo Debugger's command-line options

Turbo Debugger uses command-line options to specify special start-up parameters and debugging modes. The command-line options must be specified before you start Turbo Debugger; you can't specify them once Turbo Debugger is loaded.

The command-line syntax for starting Turbo Debugger is as follows:

```
TD | TDW | TD32 [options] [progname [progargs]]
```

The items enclosed in brackets are optional. The *options* are Turbo Debugger command-line options, and are described in Appendix A. The item *progname* refers to the name of the program you're debugging, and *progargs* are optional arguments supplied to your program. When using this syntax, be sure to supply a correct path for the program you're debugging.

For example, the following command line starts TDW with the **–jp** command-line option, and loads *my_prog* with the arguments *mammal* and *river*:

```
TDW -jp my_prog mammal river
```

Setting command-line options with Turbo Debugger's icon properties

If you start Turbo Debugger using the TDW or TD32 icons, you can specify command-line options using the icon's Property dialog box. This is usually the best way to specify command-line options because the options you specify are saved with the icon's property settings.

You can also specify your program (and optional program arguments) in the command you enter into the Properties dialog box. If you specify your program it'll be loaded into Turbo Debugger when you double-click the debugger's icon. This is the best way to load your program if you're working on an ongoing project.

To specify an icon's Property settings, click the icon, then choose File \| Properties from the Windows Program Manager. In the Command Line input box, type the executable name of the debugger, followed by the desired command-line option(s). Choose OK when you're done.

Setting command-line options from Borland's C++ integrated environment

If you transfer to Turbo Debugger from Borland's C++ for Windows integrated environment, you can specify Turbo Debugger's command-line options using the following procedure:

1 From the C++ integrated environment, choose Options | Tools to access the Tools dialog box.

2 Select TDStartup from the Tools list box.

3 Click the Edit button to open the Tools Options dialog box.

4 In the Command Line input box, enter Turbo Debugger's command-line options after the $TD transfer macro setting.

The $ARG transfer macro in the Command Line input box indicates the arguments that are passed to your program when you transfer to Turbo Debugger from the integrated environment. To specify program arguments,

1 From the integrated environment, choose Options | Environment to open the Environment Options dialog box.

2 Select Debugger in the Topics list box.

3 Enter the program arguments in Run Arguments list box.

Running Turbo Debugger

When you run TDW (or TD32 under Win32s), the debugger opens in full-screen character mode. However, unlike other applications, you cannot access the Windows shortcut keys *Alt+Esc* and *Ctrl+Esc* from Turbo Debugger. Although you can access the Windows task list from your program, you should not change tasks when Turbo Debugger is running because of the special way the debugger uses system resources.

This is different from running TD32 under Windows NT. In this case, Turbo Debugger activates in a command-prompt window, and it has all the features of a normal Windows application.

Loading your program into the debugger

You can load your program into Turbo Debugger using its command-line syntax (which is described on page 19) or from within Turbo Debugger once it has started.

To load a new program into Turbo Debugger (or to change the currently loaded program), use the File | Open command. This command opens a two-tiered set of dialog boxes, the first being the Load a New Program to Debug dialog box.

Figure 2.1   The Load a New Program to Debug dialog box
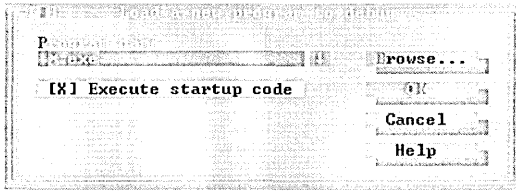
TD and TDWs' Load a New Program to Debug dialog box contains an additional
button, Session, to support its remote debugging feature. For more information on the
Session button, see "Starting and configuring TDW" on page 171.

If you know the name of the program you want to load, enter the executable name into
the Program Name input box and press *Enter*.

To search through directories for your program, click the Browse button to open the
second dialog box (the Enter Program Name to Load dialog box):

Figure 2.2   The Enter Program Name to Load dialog box
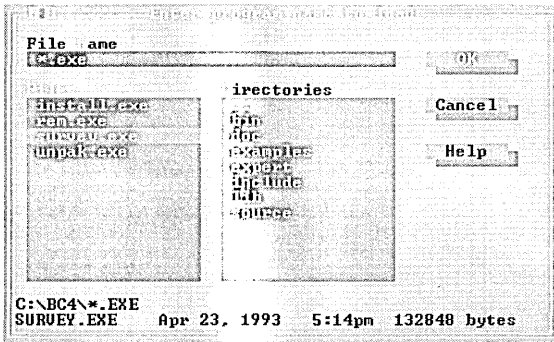
The Files list box displays the files in the currently selected directory. By entering a file
mask into the File Name input box (such as *.*EXE*), you can specify which files should be
listed. You can also use the File Name input box to change disk drives.

To "walk" through disk directories, double-click the entries listed in the Directories list
box (the .. entry steps you back one directory level). Once you've selected a directory,
choose a file to load from the Files list box. To quickly search for a file, type a file name
into the Files list box. Turbo Debugger's incremental matching feature moves the
highlight bar to the file that begins with the letters you type. Once you've selected a file,
press OK. This action returns you to the Load a New Program to Debug dialog box.

After you've specified a program in the Load a New Program to Debug dialog box,
specify whether or not you want the debugger to run its startup code. If you check the
Execute Startup Code check box, Turbo Debugger runs the program to *main* (or its
equivalent) when you load the program. If you leave this box unchecked,
Turbo Debugger will not run any code when you load the program.

To support remote debugging, TDW contains a set of radio buttons in the Load a New
Program to Debug dialog box. The Session radio buttons specify whether or not the

program you're debugging is on a local or remote system. If it's located on a remote system, select the Remote Windows radio button; if it's not on a remote system, select Local. See Appendix B for complete instructions on remote debugging.

Note Before loading a program into the debugger, be sure to compile your source code into an executable file (.EXE or .DLL) with full debugging information. Although you can load programs that don't have debug information, you will not be able to use the Module window to view the program's source code. (The debugger cannot reference the source code of executable modules that lack debug information. If you load a module that doesn't contain debug information, Turbo Debugger opens the CPU window to show the disassembled machine instructions of that module.)

When you run a program under the control of Turbo Debugger, the program's executable files (including all .DLL files) and original source files must be available. In addition, all .EXE and .DLL files for the application must be located in the same directory.

Searching for source code

When you load a program or module into Turbo Debugger, the debugger searches for the program's source code in the following order:

1 In the directory where the compiler found the source files.
2 In the directory specified in the Options | Path for Source command.
3 In the current directory.
4 In the directory where the .EXE and .DLL files reside.

Note Directories specified with the **–sd** command-line option override any directories set with the Options | Path for Source command.

Specifying program arguments

Once your program is loaded into Turbo Debugger, you can use the Run | Arguments command to set or change the arguments supplied to your program. See page 26 for more information on this command.

If you load your program using Turbo Debugger's command-line syntax (as described on page 19), you can supply arguments to the program you're debugging by placing them after the program name in the command line. For instance, the following command loads the program **MyProg** with the command-line arguments **a**, **b**, and **c**:

```
myprog a b c
```

Restarting a debugging session

When you exit Turbo Debugger, it saves to the current directory a *session-state file* that contains information about the debugging session you're leaving. When you reload your program from that directory, Turbo Debugger restores your settings from the last debugging session.

By default, all history lists, watch expressions, Clipboard items, breakpoints, operating-system exception settings, and C++ and C exception settings are saved to the session-state file. Session-state files are named *XXXX*.TR, *XXXX*.TRW, and *XXXX*.TR2 by TD, TDW, and TD32, respectively, where *XXXX* is the name of the program you're debugging. If no program is loaded when you exit Turbo Debugger, then *XXXX* will be the debugger's executable file name (TD, TDW, or TD32).

The Options | Set Restart Options command opens the Restart Options dialog box, from where you can customize how Turbo Debugger handles the session-state files.

Figure 2.3   The Set Restart Options dialog box
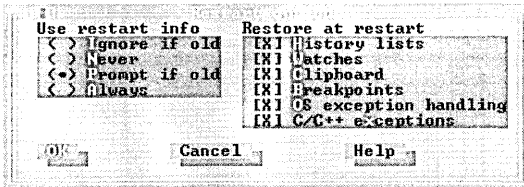
The Restore at Restart check boxes specify which debugger settings you want saved to the session-state file, and the Use Restart Info radio buttons specify when the session-state file should be loaded.

Because breakpoint line numbers and variable names can change when you edit and recompile your source code, the Use Restart Info radio buttons give you the following options for loading the session-state file:

Always   Always use the session-state file.

Ignore if old Don't use the session-state file if you've recompiled your program.

Prompt if old Turbo Debugger asks if you want to use the session-state file if
                   you've changed your program.

Never   Do not use the session-state file.

These options can also be specified using command-line switches.

Controlling program execution

The process of debugging usually entails alternating between times when your program has control, and times when Turbo Debugger has control. When the debugger has control, you can use its features to search through your program's source code and data structures, looking for where things have gone wrong. However, when your program has control, you can't access the debugger's menus and windows; you must pause your program's execution and allow the debugger to regain control. (TD32's Wait for Child command, explained on page 27, is an exception to this rule.)

Using the debugger's execution control mechanisms, you can specify when and where you want the execution of your program to pause. Turbo Debugger offers the following mechanisms to control your program's execution:

- "Single-Step" through machine instructions or source lines.
- Step over calls to functions.
- Run to a specified program location.
- Execute until the current function returns to its caller.
- "Animate" (perform continuous single-stepping).
- Reverse program execution.
- Run until a breakpoint is encountered.
- Run until a specific Windows message is encountered.
- Pause when a C++ or C exception is thrown.

Except for breakpoints, Windows messages, and C++ exceptions, all execution control mechanisms are located on the Run menu.

The Run menu

The Run menu has a number of options for executing different parts of your program. Since these commands are frequently used, most are linked to function keys.

Run

F9 The Run command runs your program at full speed. Control returns to Turbo Debugger when one of the following events occurs:

- Your program terminates.
- A breakpoint with a break action is encountered.
- You interrupt execution with the program interrupt key.
- A program error halts execution.
- A C++ or C exception that you have marked is thrown.

Go to Cursor

F4 The Go to Cursor command executes your program up to the line containing the cursor in the current Module window or CPU Code pane. If the current window is a Module window, the cursor must be on a line of source code.

Trace Into

F7 Known as *single-stepping*, this command executes a single source line or assembly-level instruction at a time.

If the current window is a Module window, a single line of source code is executed; if it's a CPU window, a single machine instruction is executed. If the current source line contains a function call, Turbo Debugger traces into the function, providing that it was compiled with debug information. If the current window is a CPU window, pressing *F7* on a CALL instruction steps to the called routine.

When you single-step through machine instructions (using Trace Into in the CPU window or by pressing *Alt+F7*), Turbo Debugger treats certain sets of machine instructions as a single instruction. This causes multiple assembly instructions to be executed, even though you're single-stepping through the code.

Here is a list of the machine instructions that cause multiple instructions to be executed when you are single-stepping at the instruction level:

CALL   LOOPNZ
INT   LOOPZ
LOOP

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

Turbo Debugger treats a class member function just like any other function; *F7* traces into the source code if it's available.

Step Over

F8 The Step Over command, like the Trace Into command, executes a single line of source code or machine instruction at a time. However, if you issue the Step Over command when the instruction pointer is located at a routine call, Turbo Debugger executes that routine at full speed, and places you at the statement following the routine call.

When you step over a source line that contains multiple statements, Turbo Debugger treats any routine calls in that line as part of the line—you don't end up at the start of one of the routines. If the line contains a return statement, Turbo Debugger returns you to the previously called routine.

The Run | Step Over command treats a call to a class member function like a single statement, and steps over it like any other function call.

Execute To

Alt F9 Executes your program until the address you specify in the Enter Code Address to Execute To dialog box is reached. The address you specify might never be reached if a breakpoint action is encountered first, or if you interrupt execution.

Until Return

Alt F8 Executes your program until the current routine returns to its caller. This is useful in two circumstances: when you've accidentally single-stepped into a routine that you don't need to debug, or when you've determined that the current routine works to your satisfaction, and you don't want to slowly step through the rest of it.

Animate

Performs a continuous series of Trace Into commands, updating the screen after each one. The animate command lets you watch your program's execution in "slow motion," and see the values of variables as they change. Press any key to interrupt this command.

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

Back Trace

If you're tracing through your program using *F7* or *Alt+F7*, you can use Back Trace to reverse the direction of program execution. Reverse execution is handy if you trace beyond the point where you think there might be a bug, and you want to return to that point.

Using the Back Trace command, you can back-trace a single-step at a time or back-trace to a specified point that's highlighted in the Execution History window. Although reverse execution is always available in the CPU window, you can execute source code in reverse only if Full History is turned *On* in the Execution History window.

For complete instructions on the Execution History window, see page 29.

Instruction Trace

The Instruction Trace command executes a single machine instruction. Use this command when you want to trace into an interrupt, or when you're in a Module window and you want to trace into a routine that doesn't contain debug information (for example, a library routine).

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

Arguments

Use the Arguments command to set or change the command-line arguments supplied to the program you're debugging. Enter new arguments exactly as you would following the name of your program on the command line.

Once you've entered the arguments, Turbo Debugger asks if you want to reload your program from disk. You should answer "Yes" because most programs read the argument list only when the program is first loaded.

Program Reset

The Program Reset command terminates the program you're running and reloads it from disk. You might use this command in the following circumstances:

- When you've executed past the place where you think there is a bug.

- When your program has terminated and you want to run it again.

- If you've suspended your application with the program interrupt key and you want restart it from the beginning. Make sure, however, that you don't interrupt the execution of your program if Windows kernel code is executing.

- If you want to debug a DLL that's already been loaded. To do so, set the Debug Startup option in the Load Module Source or DLL Symbols dialog box to *Yes* for the DLL you're interested in, and reset your program.

If you choose the Program Reset command while you're in a Module or CPU window, the Turbo Debugger resets the Instruction Pointer to the beginning of the program. However, the display is not changed from the location where you issued the Program Reset command. This behavior makes it easy for you to resume debugging from the position you were at prior to issuing the Program Reset command.

For example, if you chose Program Reset because you executed a few statements past a bug, press *Ctrl+F2* to rest your program and reposition the cursor up a few lines in your source file. Pressing *F4* will then run to that location.

Next Pending Status

The Next Pending Status command (available when you're debugging with Windows NT) can be used when the Run | Wait for Child command is set to No. When Wait for Child is set to No (and your program is running in the background while you're accessing Turbo Debugger), you can use the Next Pending Status command to retrieve a program status message. To indicate that a status message has been sent, Turbo Debugger's activity indicator displays PENDING. Status messages are sent on the occurrence of events such as breakpoints and exceptions.

Wait for Child

Wait for Child (used exclusively by TD32 for debugging Windows NT programs) can be toggled to either Yes or No. When this option is set to No, you can access Turbo Debugger *while* your program is running; you don't have to wait for your program to hit a breakpoint or exception to access the debugger's views and menus.

This command can be useful when you're debugging interactive programs. For example, if your program reads a lot of information from the keyboard, you can access the debugger while the program is waiting for input. You can set breakpoints and examine your program's data, even though your program has focus. Use the Refresh option in TD32INST to set the rate that TD32 updates the information in its windows.

Interrupting program execution

If your program is running, you can access Turbo Debugger by pressing the *program interrupt key*. The program interrupt key you use depends on the type of application you're debugging:

- Use *Ctrl+Alt+SysRq* when you're debugging a Windows 3.x program.
- Use *Ctrl+Alt+F11* when you're debugging a Win32s program.
- Use *F12* when you're debugging a Windows NT program.
- Use *Ctrl+Break* when you're debugging a DOS program.

Interrupting program execution is useful when you need to access Turbo Debugger, but haven't set any breakpoints that will interrupt your program's execution.

For example, if you single-step through a Windows application, you'll eventually get caught in the message loop, waiting for a message to be sent to your program. When this happens, you must press *F9* to run the program past the message loop. Once the program is running, you can press the program interrupt key to return to Turbo Debugger.

Stopping in Windows code

If, when you return to Turbo Debugger, you see a CPU window without any instructions corresponding to your program, you're probably in Windows kernel code. If this happens, return to Turbo Debugger and set a breakpoint at a location you know your program will execute. Next, run your program (*F9*) until it encounters the breakpoint. You are now out of Windows code, and can resume debugging your program.

Even though you can access the Module window, set breakpoints, and do other things inside Turbo Debugger, there are a few things that you *should not* do while you're stopped in Windows code:
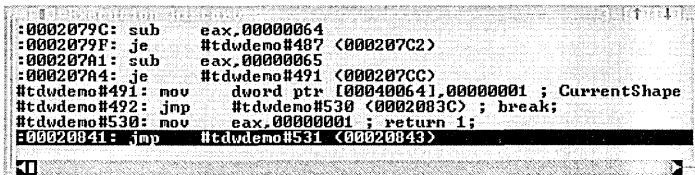
- Don't single-step through your program. Attempting to single-step through Windows kernel code can produce unpredictable effects.

- Don't terminate or reload your application or Turbo Debugger—this might cause a system crash.

  If you attempt to reload your application, Turbo Debugger displays a prompt asking if you want to continue. Select *No* to return to Turbo Debugger.

Reverse execution

Turbo Debugger's *execution history* keeps track of each instruction as it's executed, provided that you're tracing into the code. Using the Execution History window, you can examine the instructions you've executed and, if you like, return to a point in the program where you think there might be a bug. Turbo Debugger can record about 400 instructions.

Figure 2.4   The Execution History window

```
:0002079C: sub     eax,00000064
:0002079F: je      #tdwdemo#487 (000207C2)
:000207A1: sub     eax,00000065
:000207A4: je      #tdwdemo#491 (000207CC)
#tdwdemo#491: mov    dword ptr [00040064],00000001 ; CurrentShape
#tdwdemo#492: jmp    #tdwdemo#530 (0002083C) ; break;
#tdwdemo#530: mov    eax,00000001 ; return 1;
:00020841: jmp     #tdwdemo#531 (00020843)
```

The following rules apply to reverse execution:

- The execution history keeps track only of instructions that have been executed with the Trace Into command (*F7*) or the Instruction Trace command (*Alt+F7*). However, it will also track Step Over commands if the instructions listed on page 24 (in the "Trace Into" section) and page 29 (in the "Reverse Execute" section) aren't executed.

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

- As soon as you use the Run command or execute past an interrupt, the execution history is deleted. (It starts recording again when you resume tracing.)

- If you step over a function call, you won't be able to trace back beyond the instruction following the return.

- Tracing back through a port-related instruction has no effect, because you can't undo reads and writes.

- Turbo Debugger cannot execute in reverse any Windows code called by your program, unless you are in the CPU window and the code is in a DLL you've selected for debugging.

Although reverse execution is always available in a CPU window, you can only execute source code in reverse if Full History is *On*. (Full History is found on the Execution History SpeedMenu.)

The Execution History window SpeedMenu

The SpeedMenu for the Execution History window contains the following commands:

```
Inspect
Reverse execute
Full history
```
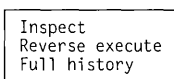
Inspect
Takes you to the command highlighted in the Instructions pane. If it is a line of source code, you are shown that line in the Module window. If there is no source code, the CPU window opens with the instruction highlighted in the Code pane.

Reverse Execute
Alt F4

Reverses program execution to the instruction highlighted in the window, and activates the Code pane of the CPU window. If you selected a line of source code, you are returned to the Module window.

Note   The following instructions don't cause the history to be thrown out, but they cannot have their effects undone. Be prepared for unexpected side effects if you back up over these instructions:

IN	INSB	OUTSB
OUT	INSW	OUTSW

Full History
Toggles from *On* to *Off*. If it's set to *On*, backtracing is enabled. If it's set to *Off*, backtracing is disabled.

The Keystroke Recording pane

TD.EXE has an extra pane in the Execution History window that lets you execute back to a given point in your program if you inadvertently destroy your execution history.

The Keystroke Recording pane at the bottom of the Execution History window becomes active when you have *keystroke recording* enabled. The –**k** command-line option enables

keystroke recording (refer to Appendix A, page 163, for more information on the **-k** command-line option). You can also use TDINST to set keystroke recording to *On*.

When Keystroke Recording is enabled, each line in the Keystroke Recording pane shows how Turbo Debugger gained control from your running program (breakpoint, trace, and so forth) and the location of your program at that time. The program location is followed by the corresponding line of source code or disassembled machine instruction.

Keystroke recording works in conjunction with reverse execution to let you return to a previous point in your debugging session. When keystroke recording is turned on, Turbo Debugger keeps a record of all the keys that you press, including the commands you issue to the debugger and the keys you press when you're interacting with the program you are debugging. The keystrokes are recorded in a file named *XXXX*.TDK, where *XXXX* is the name of the program you're debugging.

The Keystroke Recording pane SpeedMenu

The local menu for the Keystroke Recording pane contains two commands: Inspect and Keystroke Restore.

Inspect

When you highlight a line in the Keystroke Recording pane and choose Inspect from the SpeedMenu, Turbo Debugger activates either the Module window or the CPU window with the cursor positioned on the line where the keystroke occurred.

Keystroke Restore

If you highlight a line in the Keystroke Recording pane, then choose Keystroke Restore, Turbo Debugger reloads your program and runs it to the highlighted line. This is especially useful after you execute a Turbo Debugger command that deletes your execution history.

Program termination

When your program terminates, Turbo Debugger regains control and displays a message indicating the exit code that your program returned. After this, issuing any of the Run menu options causes Turbo Debugger to reload your program.

The program segment registers and stack are usually incorrect after your program has terminated, so don't examine or modify any program variables after termination.

Resetting your program

When you're debugging a program, it's easy to accidentally step past the cause of the problem. If you do, you can restart the debugging session by choosing Run | Program Reset (*Ctrl+F2*) to reload your program from disk. Resetting a program doesn't affect any debugging settings, such as breakpoints and watches.

Reloading the program from disk is the safest way to restart a program after it has terminated. Since many programs initialize variables from the disk image of the program, some variables might contain incorrect data if you restart the program without first resetting it.

Exiting Turbo Debugger

Alt X You can end your debugging session and return to the Windows Program Manager at any time (except when you're in a dialog box or when your program has control) by pressing *Alt+X*. You can also choose File | Quit to exit the debugger.

Debugging with Turbo Debugger

Debugging is the process of finding and correcting errors ("bugs") in the programs you write. Although debugging is not an exact science (the best debugging tool is your own "feel" for where a program has gone wrong), you can always profit from developing a systematic approach to finding and correcting program bugs.

This chapter discusses the basic tasks involved in debugging a program and describes how you can use Turbo Debugger to accomplish these tasks. This chapter also provides an overview of Turbo Debugger, including a section on the debugger's special features.

Debugging basics

The debugging process can be broadly divided into four steps:

1 Discovering a bug
2 Isolating the bug
3 Finding the bug
4 Fixing the bug

These four steps offer a simplified model of an actual debugging session. As a general rule, it's best to divide your program into discrete sections and debug each section separately. By verifying the functionality of each section before moving on, you can debug even the largest and most complicated programs.

Regardless of your personal debugging approach, one thing remains constant: testing and fixing source code is a part of producing software. As your programming projects become more complex, you'll reduce the time you spend debugging by developing a systematic method for testing your software.

Discovering a bug

The first debugging step can be painfully obvious. You run your program and the computer freezes. However, the presence of a bug might not be so obvious. Your

program might seem to work fine, until you enter a negative number or until you examine the output closely. Only then do you notice that the result is off by a factor of .2 or that the middle initials are missing in a list of names.

When you create a schedule for the production of your program, be sure to schedule time for a systematic test of your finished product. Be aware that if you don't thoroughly test your software, the users of your program will discover the bugs for you.

Isolating the bug

The second step can sometimes be the most difficult part of the debugging process. Isolating the bug involves narrowing down your code to the routine that contains the programming error.

Sometimes you'll be able to determine the general location of the error as soon as you see the problem. Other times, the error might show up in one place, and then in another. If you can *reproduce the bug* (find a consistent series of steps that lead to the bug), you can usually identify the routine that contains the problem.

If you can't reproduce the bug, you'll need to break your program up into individual routines and debug and verify each routine separately. Turbo Debugger is the perfect tool for this because you can check your program's data values before you run a routine, and then recheck them after the routine runs. If a routine's output is correct, then you can move on to the next routine in your program. If the output doesn't seem correct, then it's time to delve deeper into the workings of the routine.

Finding the bug

Uncovering the cause of programming errors is the true test of software engineers. Sometimes, just discovering the problem leads you to the error. For example, if you find your name list is missing everyone's middle initial, it's likely that the bug is in the line that prints the names.

Other bugs can spread themselves out through several routines, requiring that you rethink the entire design of your program. In these cases, you must trace through several functions, carefully scrutinizing the variables and data structures used in your program. This is where Turbo Debugger can help the most. By studying a routine's behavior while it runs, you can uncover the bugs that are hiding in your code.

Fixing the bug

The final step is fixing the error. Even though Turbo Debugger can help with finding the bug, you cannot use the debugger to fix your program. Once you've found the bug, you must exit Turbo Debugger to fix the source code, and then recompile your program for the fix to take effect. However, you can use Turbo Debugger to test your theory of why things went wrong; you don't need to recompile your program just to test a simple fix.

What Turbo Debugger can do for you

Turbo Debugger helps with the two hardest parts of the debugging process: isolating the bug and finding the bug. By controlling your program's execution, you can use Turbo Debugger to examine the state of your program at any given spot. You can even test your "bug hypothesis" by changing the values of variables to see how they affect your program.

With Turbo Debugger, you can perform the following debugging functions:

Table 3.1   Turbo Debugger's debugging functions

Function	Description
Tracing	Executes your program one line at a time (*single-stepping*).
Stepping	Executes your program one line at a time, but steps over any routine calls. If you're sure that a routine is error-free, stepping over it speeds up debugging.
Viewing	Opens special Turbo Debugger windows to see the state of your program from various perspectives: variables and their values, breakpoints, the contents of the stack, a data file, a source file, CPU code, memory, registers, numeric coprocessor information, object or class hierarchies, execution history, or program output.
Inspecting	Delves deeper into the workings of your program by examining the contents of complex data structures (such as arrays).
Watching	Isolates program variables and keeps track of their changing values as the program runs.
Changing	Replaces the current value of a variable, either globally or locally, with a value you specify.
Back tracing	Traces backward through code that has already been executed.

Turbo Debugger's user interface

The Turbo Debugger environment consists of a series of menus, dialog boxes, and special debugger windows. In addition, the debugger has many special features that remain hidden to the casual user. To get the most out of Turbo Debugger, you should become familiar with the features listed here and in the section "Turbo Debugger's special features" on page 41.

Working with menus

Turbo Debugger's global menu system (called the *menu bar*), runs along the top of the screen and lets you access the debugger's commands via menus. The menu bar is always available, except when a dialog box is active. To open Turbo Debugger's menus, use one of these methods:

- Press *F10*, then use *Right* arrow or → or *Left* arrow or ← to go to the desired menu and press *Enter*.

- Press *F10*, then press the highlighted letter of any menu (press *Spacebar* for the ≡ (System) menu).

- Press *Alt* plus the highlighted letter of any menu. The ≡ (System) menu opens with *Alt+Spacebar*.

- Click the menu bar command with the mouse.

Once you access a menu, you can choose a command by pressing the highlighted letter of the command.

Working with windows

Turbo Debugger uses a number of windows that provide information about the program you're debugging. To make debugging easier, Turbo Debugger provides many window-management commands that let you arrange and move through the windows you open. The window-management commands are located on the Window menu and on the ≡ (System) menu.

Selecting a window

Each window that you open is numbered in the upper right corner to allow quick access to that window. Usually, the Module window is window 1 and the Watches window is window 2. The window you open next will be window 3, and so on.

You can activate any of the first nine open windows by pressing *Alt* in combination with the window number. For example, if you press *Alt+2* to make the Watches window active, any commands you choose will affect that window and the items in it.

The bottom half of the Window menu lists the open windows. To activate a specific window, open the Window menu and press the window number. If you have more than nine windows open, the window list will include a Window Pick command; choose it to open a menu of all the windows open onscreen.

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

Using window panes

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

As you move from pane to pane, you'll notice that a blinking cursor appears in some panes and a highlight bar appears in others. If a cursor appears, you can move around the text using standard keypad commands.

Moving and resizing windows

When you open a new window in Turbo Debugger, it appears near the current cursor location. If the size or the location of the window is inconvenient, you can use the Window | Size/Move command to adjust it. Once you give this command, use the arrow keys to move the window, or use *Shift* and the arrow keys to resize the window.

If you want to enlarge or reduce a window quickly, choose Window | Zoom (*F5*), or click the mouse on the minimize or maximize box in the upper right corner of the window.

Closing and recovering windows

When you're through working with a window, you can close it by pressing *Alt+F3*, by choosing Window | Close, or by clicking the close button in the upper left corner of the window.

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

If your program has overwritten your environment screen with output (because you turned off screen swapping), you can clean it up again with ≡ (System) | Repaint Desktop. To restore Turbo Debugger's screen layout to its opening setup, choose the ≡ (System) | Restore Standard.

SpeedMenus

Each Turbo Debugger window has a special *SpeedMenu* that contains commands specific to that window. In addition, individual panes within a window can have unique SpeedMenus. To access a SpeedMenu in the currently active window (or window pane), do one of the following:

- Press the *right* mouse button inside the active window (or window pane).
- Press *Alt+F10* to open the currently active window SpeedMenu.
- Press *Ctrl* and the highlighted letter of the SpeedMenu command to choose that command (shortcut keys must be enabled for this to be effective).

Turbo Debugger's windows

Turbo Debugger uses windows (or views) to display information relating to the program you're debugging. The many different windows in Turbo Debugger each display a different type of information.

Although most of Turbo Debugger's windows are opened from the View menu, several windows are opened by other means. For example, the Inspector window can be opened by choosing the Data | Inspect command, or by pressing *Ctrl+I* from the Module window.

The View menu's windows

The View menu provides the entry point to the majority of Turbo Debugger's windows. A brief outline of each of the View menu's windows is given in the following sections.

Breakpoints window

You use the Breakpoint window to set, modify, or delete breakpoints. A breakpoint defines a location in your program where the debugger can pause the execution of your program so you can examine its status.

The Breakpoint window contains two panes: the left pane lists all set breakpoints and the right pane describes the conditions and actions of the breakpoint highlighted in the left pane. See Chapter 5 for a complete description of the Breakpoint window.

Stack window

The Stack window displays the current state of the program stack. The first function called is listed on the bottom of window, with each subsequently called function layered on top.

You can bring up and examine the source code of any function listed in the Stack window by highlighting it and pressing *Ctrl+I*. In addition, you can open a Variables window that displays all local variables and function arguments by highlighting a function in the Stack window and pressing *Ctrl+L*. Chapter 6 provides detailed information on the Stack window.

Log window

The Log window displays the contents of the message log, which contains a scrolling list of messages and information generated as you work in Turbo Debugger. It tells you such things as why your program stopped, the results of breakpoints, and the contents of windows you saved to the log.

You can also use the log window to obtain information about memory usage, modules, and window messages for your Windows application. For more information on the Log window, see Chapter 5.

Watches window

The Watches window displays the values of variables and expressions. By entering expressions into the Watches window, you can track their values as they change during the program execution. Watches can be easily added to the Watches window by pressing *Ctrl+W* when the cursor is on a variable in the Module window. See Chapter 6 for more about the Watches window.

Variables window

The Variables window displays all the variables within a given scope of your program. The upper pane of the window lists global variables and the lower pane shows any variables local to the current function.

This Variables window is helpful when you want to find a routine or symbol whose name you can't fully remember. By looking in the global Symbol pane, you can quickly find what you want. Chapter 6 describes the Variables window in more detail.

Module window

The Module window is perhaps the most important window in the debugger, because it displays the source code for the program module you're currently debugging (this includes any DLLs your program might call). However, for the source code of a module to be displayed, the module must be compiled with debug information. Chapter 8 describes the Module window and its commands.

File window

The File window displays the contents of any disk file; not just program modules as with the Module window. You can view the file either as raw hex bytes or as ASCII text, and you can search for specific text or byte sequences. Chapter 8 contains more information about the File window.

CPU window

The CPU window (described in Chapter 9) displays the current state of the central processing unit (CPU). This window has six panes showing: the program's disassembled machine instructions, the contents of the Windows selectors (in TDW.EXE only), data as hex bytes, the stack as hex words, the CPU registers, and 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. This view is also used when you want to debug Assembler programs.

Dump window

The Dump window displays the raw hexadecimal contents of any area of memory. (This window is the same as the Dump pane of a CPU window.)

Using the Dump window, you can view memory as characters, hex bytes, words, doublewords, or any floating-point format. In addition, the SpeedMenu has commands to let you modify the displayed data and manipulate blocks of memory. See Chapter 9 for more on the Dump window.

Registers window

The Registers window displays the contents of the CPU's registers and flags. This window has two panes, a registers pane and a flags pane. You can change the value of any of the registers or flags through this window SpeedMenu commands. Chapter 9 provides more information on the Registers window.

Numeric Processor window

The current state of the numeric coprocessor is displayed in the Numeric Processor window. This window has three panes: one shows the contents of the floating-point registers, one shows the values of the status flag values, and one shows the values of the control flag.

This window can help you diagnose problems in routines that use the numeric coprocessor. To reap the benefits of this window, you must have a good understanding of how the numeric coprocessor works. See the online file TD_ASM.TXT for more information about the Numeric Processor window.

Execution History window

The Execution History window (described in Chapter 2) displays machine instructions or program source lines up to the last line executed. You use this view when you want to execute code in reverse order. The window shows the following information:

- Whether you are tracing or stepping.
- The line of source code for the instruction about to be executed.
- The line number of the source code.

Hierarchy window

The Hierarchy window displays a hierarchy tree of all classes used by the current module. The window has two panes: one for the class list, the other for the class hierarchy tree. This window shows you the relationship of the classes used by the current module. By using this window SpeedMenu commands, you can examine any class' data members and member functions. See Chapter 11 for more information about using the Hierarchy window.

Windows Messages window

The Windows Messages window (described in Chapter 10) displays a list of messages sent to the windows in your Windows program. The panes in this window show the windows that you've set up for message tracking, the type of messages you're tracking, and the messages being tracked.

Clipboard window

Turbo Debugger's Clipboard is used for clipping and pasting items from one debugger window to another. The Clipboard window shows the items you have clipped and their item types. See page 43 for more information on Turbo Debugger's Clipboard.

Duplicating windows

Use the View | Another command on the Views menu to duplicate the following three windows: the Dump window, the File window, and the Module window.

Using the Another command lets you keep track of different areas of assembly code, different program files, or different areas of memory.

Other windows

In addition to the windows listed on the Views menu, Turbo Debugger also lets you access Inspector windows and the user screen.

Inspector windows

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

You can open an Inspector window to look at an array of items or to examine the contents of a variable or expression. The number of panes in the window depends on the nature of the data you are inspecting; Inspector windows adapt to the type of data being displayed.

Inspectors display simple scalars (**int, float,** and so on), pointers, arrays, structures, unions, classes, and objects. Each type of data item is displayed in a way that closely mimics the way you're used to seeing it in your program's source code.

Note   You can create additional Inspector windows by choosing the Inspect command from within an Inspector window.

User screen

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

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

Alt+F5 is the hot key that toggles between the environment and the user screen.

Turbo Debugger's special features

Turbo Debugger has many special features that make debugging easier. To get the most out of your Turbo Debugger sessions, take the time to become familiar with the following features:

- Automatic name completion
- Select by typing
- Incremental matching
- Keyboard macro capability
- The Clipboard
- The Get Info text box
- The Attach command (TD32 only)
- The OS Shell command (TD and TD32 only)
- Comprehensive help

Automatic name completion

Whenever an input box prompts you for a symbol name, you can type in just part of the symbol name and then press *Ctrl+N* to have Turbo Debugger's *automatic name completion* fill in the rest of the name for you.

The following rules apply to automatic name completion:

Ctrl N
- If you have typed enough of a name to uniquely identify it, Turbo Debugger fills in the rest of it.

- If the name you have typed so far is not the beginning of any known symbol name, nothing happens.

- If you type something that matches the beginning of more than a single symbol, a list of matching names is presented so you can choose the one you need.

If READY... appears in the upper right corner of the screen, it means the symbol table is being sorted. *Ctrl+N* won't work until the ellipsis disappears, indicating that the symbol table is available for name completion.

Select by typing

A number of windows let you start typing a new value or search string without first choosing a SpeedMenu command. *Select by typing* usually applies to the most frequently used SpeedMenu commands, like Goto in a Module window, Search in a File window, or Change in a Registers window.

Incremental matching

Turbo Debugger's *incremental matching* feature helps you find entries in alphabetical lists. 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 an item is selected (highlighted) from a list, you can press *Alt+F10* or click the right mouse button to display the SpeedMenu and choose a command relevant to the highlighted item. In many lists, you can also just press *Enter* once you have selected an item. This acts as a hot key to one of the commonly used local-menu commands.

Keyboard macros

Macros are simply hot keys that you define. You can assign any series of commands and keystrokes to a single key, and use them whenever you want.

The Macros menu

The Macros command (located on the Options menu) displays a pop-up menu that provides commands for defining new keystroke macros and deleting ones that you no longer need. It has the following commands: Create, Stop Recording, Remove, and Delete All.

Create

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

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

Stop Recording

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

Do *not* use the Options | Macro | Stop Recording menu selection to stop recording your macro, because these keystrokes will then be added to your macro.

Remove

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

Delete All

Removes all keystroke macro definitions and restores all keys to their original meaning.

The Clipboard

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

The items copied to the Clipboard are dynamic; if an item has an associated value, the Clipboard updates the value as it changes during your program's execution.

To copy an item into the Clipboard, position the cursor on the item (or highlight it with the *Ins* and arrow keys), then press *Shift+F3*. To paste something into a window or dialog box from the Clipboard, press *Shift+F4* (or use the Clip button in the dialog box) to bring up the Clipboard's Pick dialog box.

The Pick dialog box

Pressing *Shift+F4* (or a dialog box's Clip button) brings up the Pick dialog box.

Figure 3.1   The Pick dialog box
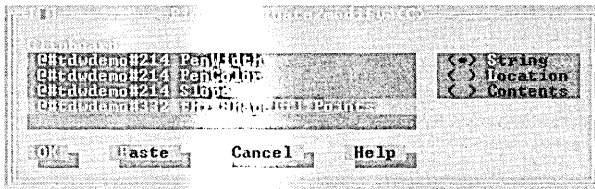
The Pick dialog box contains a list of the items in the Clipboard and a set of radio buttons that lets you paste the items in different ways:

String String pastes the Clipboard item.

Location Location pastes the address of the Clipboard item.

Contents Contents pastes the contents located at the address of the Clipboard item.

To paste an item, highlight it, select how you want to paste it, and click either OK or Paste, depending on whether you want to edit the entry:

- If you want to edit the entry, choose OK to copy the Clipboard item to the input box. Once the item is copied, you can edit the entry before pressing *Enter*.

- If you don't need to edit the entry, click Paste to copy the Clipboard item to the input box *and* to cause the dialog box to immediately perform its function.

The Clipboard window

The Clipboard window (opened with the View | Clipboard command) displays the entire contents of the Clipboard.

Figure 3.2   The Clipboard window
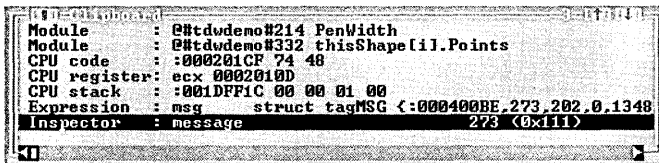
Each listing in the Clipboard window begins with the Clipboard item type. The item type is followed with the Clipboard item, and (if the item is an expression) the item's value. The following table shows Turbo Debugger's Clipboard item types:

Table 3.2   Clipboard item types

Type	Description
Address	An address without data or code attached
Control flag	An 80×87 control flag value
Coprocessor	An 80×87 numeric coprocessor register
CPU code	An address and byte list of executable instructions from the Code pane of the CPU window
CPU data	An address and byte list of data in memory from the Dump pane of the CPU window or the Dump window
CPU flag	A CPU flag value from the Flags pane of the CPU window
CPU register	A register name and value from the Register pane of the CPU window or the Registers window
CPU stack	A source position and stack frame from the Stack pane of the CPU window
Expression	An expression from the Watches window
File	A position in a file (in the File window) that isn't a module in the program
Inspector	One of the following: • A variable name from an Inspector window • A constant value from an Inspector or Watches window • A register-based variable from an Inspector window • A bit field from an Inspector window
Module	A module context, including a source code position, like a variable from the Module window
Status flag	An 80×87 status flag value
String	A text string, like a marked block from the File window

When pasting items, be careful to match the Clipboard item type with the type that the input field is expecting.

The Clipboard window SpeedMenu

The Clipboard window SpeedMenu contains the commands Inspect, Remove, Delete All, and Freeze.

The Inspect command positions the cursor in the window from which the item was clipped.

Remove deletes the highlighted Clipboard item or items. *Del* is a shortcut for the Remove command.

The Delete All command erases the contents of the Clipboard.

Freeze stops the Clipboard item's value from being dynamically updated. When you freeze an item's value, an asterisk (*) is displayed next to the entry in the Clipboard window.

Dynamic updating

The Clipboard dynamically updates the values of any items that can be evaluated, such as expressions from the Watches window. However, the Freeze command on the Clipboard window SpeedMenu lets you turn off the dynamic updating for specific Clipboard items. This lets you use the Clipboard as a large Watches window, where you can freeze and unfreeze items as you like.

The Get Info text box

The File | Get Info command opens the System Information text box, which displays general system information. Once you've finished examining the system information, close the text box by pressing *Enter, Spacebar,* or *Esc*.

The System Information text boxes display different sets of information, depending on the operating system in use. The title bar of the System Information text box lists the operating system: Windows, Win32s, or Windows NT. Figure 3.3 shows the Get Info text box used with Windows NT.

Figure 3.3   The Get Info text box

```
 Program:  ...S\TD\TDWDEM32\TDWDEMO.EXE
 Status :  Trace

 WindowsNT version  :  3.10
 Memory load factor :  4%

 Memory        Available        Total

 Physical        4345056      16449536
 Page file      16211968      37822464
 Virtual      2134241280    2147352576

 8-24-1993   5:18pm

       OK              Help
```
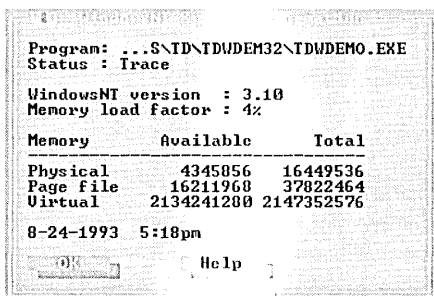
All System Information text boxes display the following general information:

- The name of the program you're debugging.

- A status line that describes how Turbo Debugger gained control. (A complete listing of Status line messages is given on page 187.)

- The DOS or Windows version number.

- The current date and time.

In addition to the general information previously listed, TDW's System Information text box provides the following global memory information:

Table 3.3   TDW's System Information text box

Field	Description
Mode	Memory modes can be large-frame EMS, small-frame EMS, and non-EMS (extended memory).
Banked	The amount in kilobytes of memory above the EMS bank line (eligible to be swapped to expanded memory if the system is using it).
Not banked	The amount in kilobytes of memory below the EMS bank line (not eligible to be swapped to expanded memory).
Largest	The largest contiguous free block of memory, in kilobytes.
Symbols	The amount of RAM used to load you program's symbol table.

TDW's System Information text box contains an additional field located under the Global Memory information. The Hardware field displays either Hardware or Software, depending on whether or not the TDDEBUG.386 device driver has been installed. For information on hardware debugging, see page 74.

The System Information text box for Win32s provides the same information as TDW's, with the exception of the two fields Symbols and Hardware.

In addition to the general information previously listed, the Windows NT System Information text box displays the following memory statistics:

Table 3.4   Windows NT System Information text box

Field	Description
Memory Load Factor	Displays the percentage of used RAM.
Physical	Displays the available and total amounts of your system's RAM.
Page file	Displays the size of the current page file, and the file's maximum size.
Virtual	Displays the available and total amounts of virtual memory.

The Attach command

The File | Attach command lets you connect TD32 to a process that's already running under Windows NT. This command is useful when you know where a program encounters problems, but are having difficulties reproducing the problem when the program runs under the debugger. By running your program up to the point of difficulty, and then attaching to it, you can start your debugging session at the point where things begin to go wrong.

When you issue the File | Attach command, the Attach to and Debug a Running Process dialog box opens.

Figure 3.4 The Attach to and Debug a Running Process dialog box
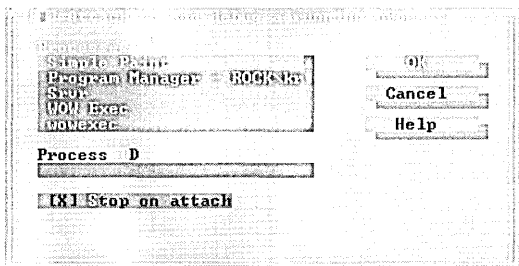
To attach to a running process,

1 Run the process you want to debug.

2 Start TD32.

3 Choose File | Change Dir to change to the directory of the running process.

4 Choose File | Attach to open the Attach to and Debug a Running Process dialog box.

5 Check or clear the Stop on Attach check box according to the following criteria:

• Check the Stop on Attach check box if you want Turbo Debugger to pause the process' execution when you attach to it.

• Clear the Stop on Attach check box if you don't want to pause the process when you attach to it.

6 Choose a process from the Processes list box (or enter a process identification number into the Process ID input box), and choose OK.

If the process contains debug information, and Turbo Debugger can find the source code, then the Module window opens with the cursor positioned at the instruction pointer, otherwise the CPU window opens. However, if the process is executing Windows code when you attach to it, then the cursor is positioned at the beginning of the program.

Once you attach to a running process, you can access Turbo Debugger and debug the process as you normally would.

If you disconnect Turbo Debugger from the running process while it's running (by either resetting the program (*Ctrl+F2*), exiting Turbo Debugger, or loading a new program), the process terminates.

The OS Shell command

The File | OS Shell command, found in TD and TD32, works with DOS and the Windows NT operating systems. When you issue this command, Turbo Debugger opens a command prompt. To return to the debugger from the command prompt shell, type *Exit*.

Getting help

Turbo Debugger offers several ways to obtain help while you're in the middle of a debugging session:

F1

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

- An activity indicator in the upper right corner always displays the current activity. For example, if your cursor is in a window, the activity indicator reads READY; if there's a menu visible, it reads MENU; if you're in a dialog box, it reads PROMPT. Other activity indicator modes are SIZE/MOVE, MOVE, ERROR, RECORDING, REMOTE, WAIT, RUNNING, HELP, STATUS, and PLAYBACK.

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

Online help

Turbo Debugger offers context-sensitive help at the touch of a key. Help is available anytime you're in a menu or window, or when an error message or prompt is displayed.

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

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

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

The status line

Whenever you're in Turbo Debugger, a quick-reference help line appears at the bottom of the screen. This status line always provides help for the current context.
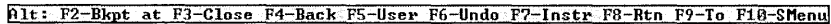
When you're in a window, the status line shows the commands performed by the function keys:

Figure 3.5   The normal status line

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

If you hold down the *Alt* key, the commands performed by the *Alt*-key combinations are displayed:

Figure 3.6   The status line with *Alt* pressed

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

If you hold down the *Ctrl* key, the commands performed by the *Ctrl*-key combinations are displayed on the status line. Because this status line shows the keystroke equivalents of the current SpeedMenu commands, it changes to reflect the current window and pane. If there are more SpeedMenu commands than can be described on the status line, only the first keys are shown.

Figure 3.7   The status line with *Ctrl* pressed

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

Whenever you're in a menu or dialog box, the status line displays an explanation of the current item. For example, if you have highlighted View | Registers, the status line says Open a CPU registers window.

Debugging a simple example

This chapter introduces you to Turbo Debugger's basic features by guiding you through a simple debugging session. After completing the example debugging session, you'll be able to:

- Start Turbo Debugger and load your program into the debugger.
- Run the program.
- Single-step through source code.
- Set breakpoints to pause your program's execution.
- Set conditional breakpoints.
- · Set watches on program variables.
- Examine the values of arrays and data structures.
- Change the value of variables.
- Test bug fixes.

The Simple Paint program

The example program, titled "Simple Paint," is a Windows program that lets you draw rectangles, lines, and ellipses. To install the example program files,

1 Run the Borland C++ 4.5 installation program.

2 Click the Customize BC 4.5 Installation button to open the Customize BC 4.5 Installation dialog box.

3 Click the Example Files button to open the Borland C++ Example Options dialog box.

4 Check the Misc. Windows Examples check box.

5 Click the Install button to install the example program files into the EXAMPLES\WINDOWS\TDW subdirectory under your Borland C++ directory.

The Simple Paint program consists of the source-code file S_PAINT.C and the executable program S_PAINT.EXE. In addition to these two files, the example program

also contains all the files you'll need to create a buggy version of the program for the sample debugging session: MAKEFILE, TDWDEMO.H, TDWDEMO.RC, TDWDEMO.ICO, TDWDEMO.IDE, and TDWDEMO.BUG.

Although the example program is not particularly useful, it effectively demonstrates how to debug a program using Turbo Debugger. In addition, this chapter shows you how to overcome some of the obstacles you might encounter while debugging Windows programs.

Running Simple Paint

Before you begin debugging the example program, run the working version of the program (S_PAINT.EXE) to get an idea of what the program does:

1 Choose File | Run from the Windows Program Manager.

2 Type \<*YourBC++Directory*>\EXAMPLES\WINDOWS\TDW\S_PAINT into the Command Line input box.

3 Choose OK.

When you run the program, it displays a blank window titled "Simple Paint." Experiment with the program by dragging the mouse to draw lines, rectangles, and ellipses using the three available colors and three different pen sizes. Notice that when you resize or minimize the Simple Paint window, the figures you've drawn are remembered and correctly redrawn by the program. When you've finished experimenting with the program, click the Quit menu item to exit the program.

Once you have an idea of what the program does, examine S_PAINT.C in Borland's C++ Integrated Development Environment (or your favorite program editor) to see how the program works.

The beginning of the program contains the prototypes of several user-defined functions. The functions that do most of the work in Simple Paint are **DoLButtonDown** (which does the preparation work when you press the left mouse button to begin drawing), **DoMouseMove** (which draws the shapes as you move the mouse), and **DoLButtonUp** (which cleans up when you've finished drawing a shape).

Compiling TDWDEMO

Now that you're familiar with the working version of the program, you can compile the buggy version of the program and begin the example debugging session. First, make a working copy of the buggy program by copying TDWDEMO.BUG to TDWDEMO.C using the following DOS commands:

```
cd \<YourBC++Directory>\examples\windows\tdw
copy tdwdemo.bug tdwdemo.c
```

Now compile the example program using either DOS commands or Borland's C++ Integrated Development Environment (IDE).

Compiling TDWDEMO using DOS commands

To compile the example program from a DOS command line, issue the following DOS command from the \<*YourBC++Directory*>\EXAMPLES\WINDOWS\TDW directory:

```
make -DWIN16 -DDEBUG
```

Note   To make a 32-bit version of the example program, substitute the make switch -DWIN32 for -DWIN16.

Compiling TDWDEMO using the IDE

Follow these steps to compile the example program using Borland's C++ IDE:

1 Load the IDE by double-clicking the Borland C++ icon in the Borland C++ 4.5 program group.

2 Choose Project | Open Project to access the Open Project File dialog box.

3 In the Directories list box, navigate to the directory \<*YourBC++Directory*>\EXAMPLES\WINDOWS\TDW.

4 Double-click the file TDWDEMO.IDE in the File Name list box.

The Project window opens at the bottom of the IDE.

5 Choose Project | Build All to compile the example program.

6 Choose OK when the project finishes building to close the Compile Status dialog box.

Debugging TDWDEMO

When you compile the example program for the first time, the compilation ends with a couple of warnings and a string of error messages. Although the compiler will generate an executable file if your program has warnings, it will not create an executable file if your program has errors.

The first error message reported is

```
Error TDWDEMO.C 171: Statement missing ;
```

You have now stumbled across the first bug, a program *syntax error*—the program source code doesn't conform to the syntax of your language compiler.

Syntax errors are the easiest program bugs to find because your language compiler flags each syntax error with an error message. Each error message displays the source file containing the error (in this case, TDWDEMO.C), the line number on which the error occurs (in this case, line 171), and the reason why the compilation failed (in this case, a semicolon is missing). Although the compiler might not find the exact cause or location of the program error, the error message usually gives enough information to help you find the bug in the program's source code. Because of this, you don't need Turbo Debugger to find or fix syntax errors.

To fix the first syntax error,

1 Load TDWDEMO.C into your program editor and navigate to line 171. (If you're using the IDE, double-click on the first error message in the Message window.)

When you arrive at line 171, you'll find that it cannot contain the error; line 171 contains only a closing brace. However, because the compiler could not get past this point during compilation, you can deduce that the error must be located in one of the preceding lines of code.

If you examine line 170, you'll see a call to the function **LineTo** that is missing the statement-closing semicolon.

2 Insert a semicolon after the **LineTo** function call to fix the first program bug.

3 Save the change to the TDWDEMO.C file.

4 If you're not using the IDE, exit your program editor.

5 Recompile the program using the steps given in the "Compiling TDWDEMO" section on page 52.

The program now compiles without any errors and the executable file TDWDEMO.EXE is created. This bug fix demonstrates two things:

• A single syntax error can generate several compiler errors.

• Syntax errors usually cause the compiler to fail on the line immediately following the one that contains the error.

Running the buggy program

Now that you've corrected the syntax error, you have an executable program that can be run under Windows. You can run TDWDEMO through either the Program Manager or by creating a program icon. Be careful when you do this, however, because it might hang your system.

To run the program through Windows' Program Manager:

1 Choose File | Run from the Program Manager.

2 Type \<*YourBC++Directory*>\EXAMPLES\WINDOWS\TDW\TDWDEMO into the Command Line input box.

3 Choose OK.

To create a program icon for TDWDEMO:

1 Open the Borland C++ 4.5 program group in Windows.

2 Choose File | New from the Windows Program Manager.

3 Choose the Program Item radio button and choose OK.

4 Enter Simple Paint into the Description input box.

5 Press *Tab*.

6 Type \<*YourBC++Directory*>\EXAMPLES\WINDOWS\TDW\TDWDEMO into the Command Line input box.

7 Choose OK.

The Simple Paint icon is added to the Borland C++ program group.

8 Double-click the Simple Paint icon to run the program.

When you run this version of the program, the Simple Paint window comes up but nothing else happens. You can't draw any shapes, you can't access any program menus, and you can't close the window. In fact, you'll have to reboot your system to terminate the program.

This is a bug that Turbo Debugger can help you resolve. After restarting your system, run Turbo Debugger (as described in the "Starting Turbo Debugger" section on page 18) and load TDWDEMO.EXE (using the method described in "Loading your program into the debugger" on page 20).

When you load TDWDEMO.EXE into Turbo Debugger, the debugger activates the Module windows and places the cursor at the **WinMain** function, as shown in Figure 4.1.

Figure 4.1 TDWDEMO loaded into Turbo Debugger
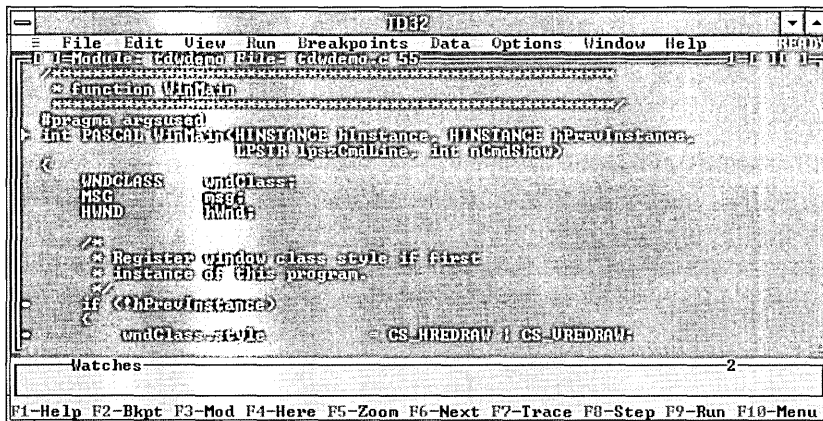
Stepping through program code

Now you're ready to begin debugging the program. However, you don't want to start by running the program because you know that this would hang your system. Instead, you must try to find this error by stepping through the program code:

1 Press *F7* (or choose the Run | Trace Into command) to execute the first line of code.

This command positions the instruction pointer at the next line of executable code (marked with a bullet in the left column of the Module window), which contains the statement if (!hPrevInstance). This line begins the block of code that initializes and registers the window class of the application's parent window.

2 Press *F7* twelve times to step through the program to the line of code containing the **CreateWindow** call. (Make sure the program finishes each step before you press *F7* to execute the next step.)

Executing this series of steps confirms that the window is registered correctly.

3 Press *F7* three more times to create the window, display the window onscreen, and validate the window.

The cursor now rests on the line that contains the `while` statement that marks the beginning of the message loop (the lines of code where the program receives and dispatches window messages).

4 Press *F7* twice to execute the calls to **GetMessage** and **TranslateMessage**.

At this point, the instruction pointer returns to the `while` statement containing the **GetMessage** call.

Careful examination reveals the problem: the previous received message was never dispatched. Although the program receives messages from the message queue, the call to **DispatchMessage** is outside the message loop block. This bug causes the program to hang because it never gets a chance to process the messages that are sent to it.

Fixing a bug

When you find a bug as serious as this one, you must fix it before you can continue debugging. To fix this program bug,

1 Exit Turbo Debugger.

2 Load TDWDEMO.C into your program editor and navigate to line 93.

If you're using the IDE, open the TDWDEMO project and navigate to line 93.

3 Restructure the message loop so it reads as follows:

```
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return 0;
```

4 Save the new file.

5 If you're not using the IDE, exit the editor.

6 Recompile the program using the methods described in "Compiling TDWDEMO" on page 52.

Note The "Compiling TDWDEMO using the IDE" section on page 53 explains how to open and compile a project.

Fixing warnings

Compile the program again. This time, you don't receive any error messages, but you still receive the warnings. In spite of the warnings, the compiler generates the executable file TDWDEMO.EXE.

Now when you run the program (using the steps described in "Running the buggy program" on page 54), it runs without hanging the system. You can draw different colored lines, rectangles, and ellipses. However, you'll soon discover a number of serious problems. In fact, there are so many problems that it might be difficult to figure out which one to fix first.

In this case, the best place to start is with the compiler warning:

```
Warning tdwdemo.c 164: Possibly incorrect assignment
```

Although you can sometimes ignore compiler warnings, you must first know the cause of the warnings so you can determine if will affect the behavior of your program. For example, the linker warning can be ignored because it only notifies you of the absence of a .DEF file.

To find the cause of the compiler warning, open TDWDEMO.C in your program editor and navigate to line 164, where you'll find the following code:

```
If (Slope = 1)
```

This code reveals a common C programming error: the assignment operator (a single equal sign) was used where an equality operator (a double equal sign) was needed. This programming error causes the condition in the **if** statement to always evaluate true (the value 1 can always be assigned to the variable *Slope*) and the **else** block of the if-then-else statement never gets executed.

This warning cannot be ignored; you must fix the problem before continuing:

1 Open TDWDEMO.C in your program editor and navigate to line 164 (if you're using the IDE, double-click on the warning in the Message window).

2 Modify the source code so the **if** statement reads as follows:

```
If (Slope == 1)
```

3 Save the program change.

4 Recompile the program.

  The program now compiles without errors or warnings.

5 Run the program to see the effect of your latest fix.

Stepping into the message loop

Test the program again: you'll find that things work better, but the program is still not perfect. The shapes that appear onscreen aren't the same shapes you're drawing, but when you resize the window, the correct shapes appear.

Again, you can use Turbo Debugger to find the cause of these problems:

1 Reload the program into Turbo Debugger (if you're in the IDE, you can choose the Tool | TDW command to transfer to Turbo Debugger).

The following message appears: `Restart info is old, use anyhow?`

2 Click *No* to discard the old session-state information (for more information on session-state files, see "Restarting a debugging session" on page 22).

The screen shown in Figure 4.1 is displayed.

3 Step to the message loop on line 93 by pressing *F7* sixteen times.

Now you're ready to see how the program handles the messages passed to it.

4 Press *F7* three times to execute the message loop functions **TranslateMessage** and **DispatchMessage**.

The instruction pointer returns to the beginning of the message loop.

5 Press *F7* three more times.

Again the instruction pointer is placed at the beginning of the message loop.

At this point, you're trapped in the message loop. To get out of the loop, you must run your program so that it will generate window messages that can be processed by **WndProc**.

Note   If you step into the message loop using *F7* or *F8*, you'll be trapped in the loop until you press *F9* to run out of the loop. However, before pressing *F9* to run the program, you must set *execution controls* so that Turbo Debugger can regain control from the program after it starts running. Execution controls consist of items such as breakpoints and messagepoints.

Setting breakpoints

When you set a breakpoint in your code, it's best to position the breakpoint just before the area where you think there's a bug. If you have an idea of where your program runs into trouble, you can use a breakpoint to pause your program's execution before it hits the trouble spot. When your program pauses, control is given to Turbo Debugger and you can use its features to monitor your program's behavior.

To figure out where to place a breakpoint in TDWDEMO.C, you must focus on a single bug. With TDWDEMO, the most obvious bug shows itself as soon as you move the mouse to draw a shape. By reviewing the source code, you can see that the function **DoMouseMove** draws the shapes. Knowing this, you can place a breakpoint on the **DoMouseMove** function so that Turbo Debugger will gain control when this function is called:

1 With Turbo Debugger's Module window active, press *Ctrl+S* (or choose the Search command from the Module window SpeedMenu) to bring up the Enter Search String input box.

2 Type `DoMouseMove`, and press *Enter* to search for the **DoMouseMove** function definition.

The cursor moves to the **DoMouseMove** function call.

3 Press *Ctrl+N* (or choose the Next command from the Module window SpeedMenu) to repeat the search for the function definition.

The cursor moves to the comment section for the **DoMouseMove** function.

4 Press *Ctrl+N* again to bring the cursor to the definition of **DoMouseMove** (line 375 in TDWDEMO.C).

Use one of the following methods to place a breakpoint at the beginning of the **DoMouseMove** function:

- Press *F2*.
- Choose Breakpoints | Toggle.
- Click in either of the two leftmost columns of the Module window.

When you set the breakpoint, the line containing the breakpoint changes colors.

After setting the breakpoint, run the program by pressing *F9*. The Simple Paint window is displayed.

Now as soon as you move the mouse, the breakpoint activates and the program's execution pauses at the beginning of the **DoMouseMove** function. However, this isn't exactly what you intended because you need the breakpoint to pause the program just before the program gets into trouble. In this case, you need Turbo Debugger to gain control just as you begin to draw a shape.

To get the proper result from the breakpoint, you must modify it so that it activates only when a shape is being drawn. You can do this by setting a conditional breakpoint.

Creating a conditional breakpoint
In the Simple Paint program, the static variable *mouseDown* is used to indicate whether or not a shape is being drawn; when you're drawing a shape, *mouseDown* equals 1.

With this information, you can create a conditional breakpoint that pauses the program on **DoMouseMove** only when you're drawing a shape:

1 Open Turbo Debugger's Breakpoints window (View | Breakpoints).

2 If it's not already highlighted, highlight the breakpoint listed as _DoMouseMove.

3 Press the right mouse button inside the left pane of the Breakpoints window to open the Breakpoint window SpeedMenu.

4 Choose Set Options to open the Breakpoint Options dialog box.

5 Click the Change button to open the Conditions and Actions dialog box.

6 Click the Expression True radio button and enter the following expression into the Condition Expression list box:

```
mouseDown == 1
```

7 Choose OK to exit the Conditions and Actions dialog box.

8 Choose OK to exit the Breakpoint Options dialog box.

The Breakpoints window displays the newly created conditional breakpoint, as shown in Figure 4.2.

Figure 4.2   Breakpoints window with a conditional breakpoint
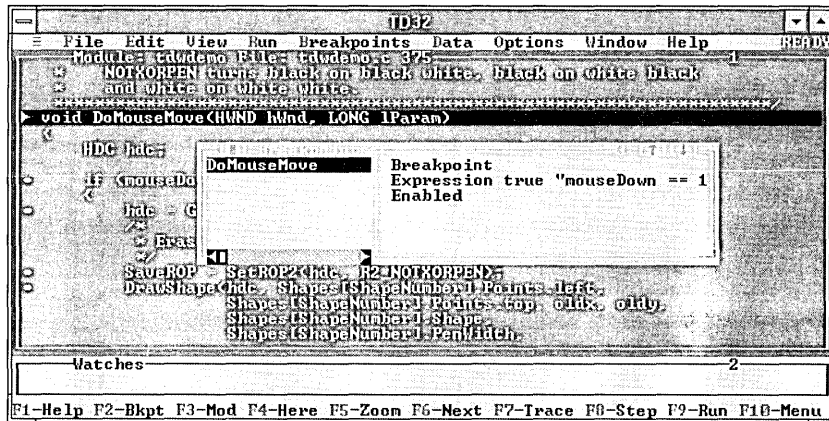
9 Close the Breakpoints window by clicking the close button in the window's upper left corner.

Once you've set the conditional breakpoint, press *F9* to run the program.

When you run the program this time, you can move the mouse freely without pausing the program's execution. However, as soon as you press the mouse button to draw a shape, the breakpoint activates and causes Turbo Debugger to display the following message:

```
Breakpoint at _DoMouseMove "mouseDown == 1" true
```

Choose OK to dismiss the message.

Turbo Debugger now gains control and you can begin stepping through the **DoMouseMove** function:

1 Press *F7* four times to bring the instruction pointer to the **DrawShape** function call.

2 Press *F8* to "step over" this function call.

Because you're not interested in examining the function **DrawShape**, you can run this function without stepping into it by pressing *F8*. However, if step into a function by accident (using *F7*), you can run that function to its end by choosing the Run | Until Return command. From there, you can continue single-stepping.

After stepping over the call to **DrawShape**, the instruction pointer is placed at the following line of code:

```
oldy = LOWORD(lParam);
```

This line of code places the low-order word of the *lParam* argument into the variable *oldy* to set the y starting coordinate for the shape. This line is interesting because the program seems to be drawing the shape from the wrong starting position.

By tracing the *lParam* argument back to its origin, you can find that it was passed to the program as part of the WM_MOUSEMOVE window message. If you look up the definition of WM_MOUSEMOVE, you'll see that its *lParam* argument represents x and y coordinates, divided into a low-order word and a high-order word. However, the low-order word represents the x coordinate, not the y coordinate as the program implies. In the program, the variables *oldy* and *oldx* are incorrectly assigned each other's values.

This program bug causes the program to draw incorrect shapes every time you move the mouse. To fix the bug, exit Turbo Debugger and use your editor to modify lines 395 and 396 of TDWDEMO.C to read as follows:

```
oldx = LOWORD(lParam);
oldy = HIWORD(lParam);
```

After saving your program changes, recompile TDWDEMO.C.

Setting watches and inspecting data structures

The new bug fix lets you draw shapes, but there are still some problems that prevent the program from working smoothly. When you experiment with the program now, you'll find that extra lines appear when you release the left mouse button after drawing a line. However, the extra lines disappear when you resize the window.

Inspecting the source code reveals that the user-defined function **DoLButtonUp** processes the program's WM_LBUTTONUP window messages (these messages are sent to your program whenever you release the left-mouse button after drawing a shape). Because of this, the **DoLButtonUp** function might be a good place to find the next bug.

Run Turbo Debugger and load TDWDEMO.EXE to begin searching for this bug. When you load the program, the following message is again displayed by Turbo Debugger:

```
Restart info is old, use anyhow?
```

This message pertains to the session-state file that contains the settings from your last debugging session. In your last session, you created a conditional breakpoint in the **DoMouseMove** function. Because you no longer need this breakpoint, click the No button to close the message box without loading this breakpoint setting.

Next, navigate to the **DoLButtonUp** function:

1 With the Module window active, press *Ctrl+S* (or choose the Search command from the Module window SpeedMenu) to bring up the Enter Search String input box.

2 Type DoLButtonUp and choose OK.

The cursor moves to the **DoLButtonUp** function call.

3 Press *Ctrl+N* (or choose the Next command from the Module window SpeedMenu) two times to move the cursor to the **DoLButtonUp** function definition on line 316 of TDWDEMO.C.

Setting watches

You can now set watches to monitor the *oldx* and *oldy* function variables:

1 Press *Ctrl+F7* (or choose Data | Add Watch) to open the Enter Expression to Watch input box.

2 Type `oldx` and choose OK to set the first watch.

3 Use the down-arrow key to move the cursor to line 361 of TDWDEMO.C.

This line contains the code `oldy = -1;`.

4 Use the mouse to highlight the variable *oldy*.

5 Press *Ctrl+W* (or choose Watch from the Module window SpeedMenu) to add the variable *oldy* to the Watches window.

The Watches window at the bottom of your display now shows the watches *oldx* and *oldy*, which both have a value of –1.

Running to the cursor location

You're now ready to run the program and examine the workings of the **DoLButtonUp** function. To pause the program's execution inside this function, run the program to the cursor location:

1 Move the cursor to the **SetRect** function call on line 328 of TDWDEMO.C.

2 Run the program to this location by choosing Run | Go To Cursor (or by pressing *F4*).

When you run the program, its screen appears. To reproduce the bug you're working on, draw a negatively-sloped line that starts in the lower right corner of the screen and ends in the upper left corner of the screen. As soon as you release the left-mouse button, Turbo Debugger regains control at the **SetRect** call, which is in the **DoLButtonUp** function.

Inspecting compound data structures

Because the **DoLButtonUp** function is called only after you've finished drawing a shape, you can inspect the *Shapes* data structure to see how the program stores the line you've just drawn:

1 Use the → key to place the cursor under the `S` in `Shapes` on the line containing the cursor.

2 Press *Ctrl+I* (or choose Inspect from the Modules window SpeedMenu) to open an Inspector window on the *Shapes* array.

The Inspector window opens and displays the contents of the array *Shapes*. Each element in the *Shapes* array is a structure of the *SSHAPE* data type.

3 Use the ↓ key to highlight the first element in the array (array element `[0]` holds the data pertaining to the line you've just drawn).

4 Press *Ctrl+D* (or choose Descend from the Module window SpeedMenu) to replace the current Inspector window with an Inspector window that displays the data structure of the first *Shape* array element.

If the new Inspector window is covering the Watches window, uncover the Watches window by moving the Inspector window to a different screen location.

The Watches window shows the variables *oldx* and *oldy*, which correspond to the x and y starting coordinates of your line. The ending point of your line is contained in the *Points* data member of the *SSHAPES* data structure, which can be seen by examining the Inspector window.

Figure 4.3 Inspector and Watches windows
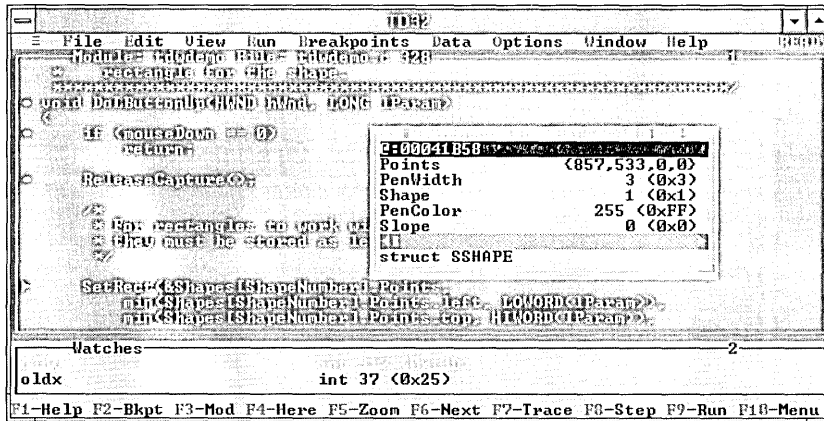
Now, you can watch the program's behavior by monitoring the Inspector window:

1 Press *F7* to step past the **SetRect** function call.

The *Points* data member in the *SSHAPES* data structure has been updated to contain the starting and ending points of the line you've just drawn.

2 Press *F7* two more times to execute the **if** statement and the statement that follows it.

These two statements set the value of the *Slope* variable if the shape drawn was a line. The second statement uses the ternary operator to set the *Slope* variable to 1 if the slope of the line is positive, and to 0 if the slope of the line is negative.

You can see the result of the operation by examining the *Slope* variable in the Inspector window—the slope of the line has been set to 1. This setting indicates a positively-sloped line. However, the line you drew was negatively-sloped. You have just uncovered another bug in the program.

By examining the condition in the ternary expression, you can see that the condition is correct, but the assignments at the end of the statement are switched. The last line in the statement should read as follows:

```
HIWORD(lParam)) ? 1 : 0;
```

Fix this bug by exiting Turbo Debugger and editing the source code on line 343 in TDWDEMO.C so that it matches the preceding line of code. After saving your change, recompile the program.

Now when you run the program, things seem to work better. You can draw lines that start from any corner of the screen and you can change colors and line thicknesses. You can also draw ellipses and rectangles. However, overlapping rectangles aren't drawn

correctly and if you draw three shapes and resize the screen, only two shapes get redrawn. Indeed, there's still one more bug left in the code.

Producing the bug in Turbo Debugger

By closely watching the behavior of the program, you can find a place to begin the next bug search. One part of the bug shows itself whenever the program redraws the screen. In the program, the function **DoPaint** processes the **WM_PAINT** messages, which in turn causes the program to redraw the screen. When you inspect this function's code, you'll discover that the **for** loop on line 242 does the actual drawing of the shapes.

However, the condition in the **for** loop looks a bit suspicious; it appears to terminate the loop before the last shape is drawn. Because you can modify the **for** loop during the execution of your program, you can use Turbo Debugger to test your theory about why things are going wrong. To do this, run the program once to demonstrate the bug and then run it a second time to see if you can fix the problem:

1 Run Turbo Debugger and load TDWDEMO.C.

2 Click No to dismiss the `Restart info is old, use anyhow?` message.

3 Run the program by pressing *F9*.

4 Draw three shapes that overlap each other: a line, an ellipse, and a rectangle.

5 Resize the window.

The bug should be now be apparent—only two of the original three shapes are redrawn.

6 Click Quit to exit the program.

Turbo Debugger displays the message: `Terminated, exit code 0`.

7 Choose OK.

Resetting the program

You can now test your fix by modifying the value of *ShapeNumber* to see if that changes the behavior of the program.

1 Press *F9* to run the program.

Turbo Debugger issues the following message:

```
Program already terminated, reload?
```

2 Click Yes to reload the program from disk.

The Simple Paint window now opens, but everything seems to be broken; you can't draw any shapes. However, this behavior is a result of the way that Windows interacts with Turbo Debugger. To fix this, press the *Alt* key.

Note When you reset a program (either after it terminates or after using the Run | Program Reset command) and then run it, you must press the *Alt* key before Windows resumes passing mouse messages to the program.

Click Quit to exit the program, then choose OK to dismiss the `Terminated, exit code 0` message.

Changing the values of variables

You're now ready to resume testing. Press *Ctrl+F2* to reset the program. Next, set a breakpoint that activates when the program processes a **WM_PAINT** window message:

1 Navigate to the *for* in the **DoPaint** function (line 242 of TDWDEMO.C).

2 Press *F2* to set a breakpoint on that line.

You now need to modify the breakpoint so that it activates after **DoPaint** is called for the fourth time. This will pause the program just before you resize the program's screen.

3 Choose View | Breakpoints to open the Breakpoints window.

4 If it's not already highlighted, highlight the breakpoint listed as `#TDWDEMO#242`.

5 Choose Set Options on the Breakpoints window SpeedMenu to open the Breakpoint Options dialog box.

6 Press the Change button to open the Conditions and Actions dialog box.

7 Type 4 into the Pass Count input box, and choose OK to return you to the Breakpoint Options dialog box.

This sets the Pass Count to 4, which causes the breakpoint to activate the fourth time it's encountered during the program's execution.

8 Press *Enter* two times to return you to the Module window.

Now run the program to test your theory of the bug.

9 Press *F9* to run the program.

10 Press the *Alt* key to tell Windows to resume passing mouse messages to your program.

11 Draw three overlapping shapes, a line, an ellipse, and a rectangle.

12 Resize the screen.

The breakpoint you set has now been encountered for the fourth time and Turbo Debugger regains control. The Module window opens with the cursor positioned on the **for** loop, and you're ready to modify the value of *ShapeNumber*:

13 Use the mouse to highlight `ShapeNumber` in the **for** loop.

14 Choose Data | Evaluate/Modify to open the Evaluate/Modify dialog box.

`ShapeNumber is inserted onto the Expression input box.`

15 Press the Eval button to evaluate the expression.

When you press the Eval button, the Result list box displays the current value of the expression being evaluated. In this case, the Result list box shows that the current value of ShapeNumber is 2.

To force the loop to draw all shapes, change the value of ShapeNumber to 3 using the Evaluate/Modify dialog box:

1 Enter 3 into the New Value input box in the Evaluate/Modify dialog box.

2 Press the Modify button to have the change take effect.

The Result list box now shows the new value of ShapeNumber.

3 Close the Evaluate/Modify dialog box by clicking its close button in the upper left corner of the window.

4 Press *F9* to resume running the program.

When the program's execution resumes, all three shapes are correctly redrawn. Your bug fix worked perfectly. However, the patch you made to the program works only for this one **WM_PAINT** message. The next time you draw a shape or resize the window, the bug will show up again. To correct this final program bug, you'll need to exit Turbo Debugger and fix the source code.

This last bug is actually a common C programming mistake: the condition in the **for** loop is off by one. You can fix this last bug by changing the **for** loop on line 242 in TDWDEMO.C to read as follows:

```
for (i = 0; i <= ShapeNumber; ++i)
```

After you compile the program this time, you'll find it runs without errors.

You've now complete the sample debugging session and should be able to use Turbo Debugger's basic features. However, this chapter introduces only some of the more powerful features of the debugger. To get the most out of Turbo Debugger, skim the rest of the book to get a general idea of the other features that you can use while debugging your programs.

Setting and using breakpoints

Breakpoints are tools that you use to control the execution of your program. By setting breakpoints in the areas of your program that you want to examine, you can run your program at full speed, knowing that its execution will pause when the breakpoints are encountered. Once your program's execution is paused, you can use Turbo Debugger's features to examine the state of your program.

In this chapter, you'll learn how to set the following types of breakpoints:

- Simple breakpoints
- Expression-true breakpoints
- Changed-memory breakpoints
- Global breakpoints
- Hardware breakpoints

This chapter also describes the Log window (see page 81), which lets you "take notes" during your debugging session.

Breakpoints defined

Turbo Debugger defines a breakpoint in three ways:

- The *location* in the program where the breakpoint is set.
- The *condition* that allows the breakpoint to activate.
- The *action* that takes place when the breakpoint activates.

Breakpoint locations

A breakpoint is usually set on a specific source line or machine instruction in your program. When set at a specific *location*, Turbo Debugger evaluates the breakpoint when your program's execution encounters the code containing the breakpoint.

However, a breakpoint can also be *global* in context. Turbo Debugger evaluates global breakpoints after the execution of *each* line of source code or machine instruction. Global breakpoints let you pinpoint where in your program a variable or pointer gets modified.

Breakpoint conditions

When your program's execution encounters a breakpoint, Turbo Debugger checks the breakpoint's condition to see if the breakpoint should activate. If the condition evaluates to true, the breakpoint activates, and its actions are carried out.

The *condition* of a breakpoint can be any of the following:

- Always activate when the breakpoint is encountered.
- Activate when an expression evaluates to true.
- Activate when a data object changes value.

In addition to the breakpoint condition, a *pass count* can be specified, requiring that a breakpoint be encountered a designated number of times before it activates.

When you're debugging programs written for Windows NT, you can also set breakpoints that relate to specific program threads. For more on setting breakpoints on program threads, see the section "Setting breakpoints on threads" on page 80.
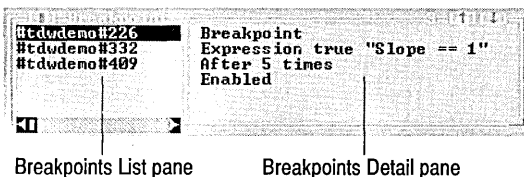
Breakpoint actions

When a breakpoint activates, it performs a specified action. A breakpoint's *action* can be any of the following:

- Pause the program's execution.
- Log the value of an expression.
- Execute an expression.
- Enable a group of breakpoints.
- Disable a group of breakpoints.

The Breakpoints window

The Breakpoints window, opened with the View | Breakpoints command, lists all currently set breakpoints.

Figure 5.1   The Breakpoints window

Breakpoints List pane   Breakpoints Detail pane

The Breakpoints window has two panes. The List pane (on the left) lists the addresses of all currently set breakpoints. The Detail pane (on the right) displays the condition and

action settings of the breakpoint that's highlighted in the List pane. (Although a breakpoint can have several sets of conditions and actions, the Detail pane displays only its first set of details.)

The Breakpoints window SpeedMenu

You access the SpeedMenu of the Breakpoints window through the List pane. The commands in this menu let you add new breakpoints, delete existing ones, and change a breakpoint's settings.

Breakpoint types

In Turbo Debugger, you can create the following types of breakpoints:

Table 5.1   Breakpoint types

Breakpoint type	Definition
Simple breakpoints	Always pause your program's execution when they're encountered.
Expression-true breakpoints	Pause your program when an expression you enter evaluates to true (nonzero).
Changed-memory breakpoints	Pause your program when a specific location in memory changes value.
Global breakpoints	Expression-true or changed-memory breakpoints that are evaluated after the execution of each source line or machine instruction is executed.
Hardware breakpoints	Global changed-memory breakpoints that are hardware assisted.

Note   You can also set breakpoints on window messages. For a complete description of message breakpoints, refer to Chapter 10, page 133.

Setting simple breakpoints

When you first set a breakpoint, Turbo Debugger creates a *simple breakpoint* by default. Simple breakpoints are set on specific lines of code and contain a condition of "Always" and an action of "Break."

When you begin a debugging session, you can quickly reach the sections of code you want to examine by setting simple breakpoints in the code. After setting the breakpoints, run your program using *F9*; the program's execution will pause when it encounters the breakpoints.

Although there are several ways to set simple breakpoints, the Module window and the Code pane of the CPU window offer the easiest methods:

• If you're using the keyboard, place the cursor on any executable line of source code (or on any machine instruction in the Code pane of the CPU window) and press *F2*. (In the Module window, executable lines of source code are marked with a " • " in the leftmost column.) The Breakpoint | Toggle command provides the same functionality.

Whenever you set a breakpoint, the line containing the breakpoint turns red. Pressing *F2* again removes the breakpoint.

- Alternately, if you're using a mouse, click either of the two leftmost columns of the line where you want the breakpoint set. (When you're in the correct column, an asterisk (*) appears inside the mouse pointer.) Clicking the line again removes the breakpoint.

- The Breakpoint | At command also sets a simple breakpoint on the current line in the Module window or Code pane of the CPU window. However, in addition to setting the breakpoint, the At command opens the Breakpoint Options dialog box, giving you quick access to the commands that let you customize the breakpoint. The hot key for At is *Alt+F2*.

In addition to setting breakpoints from the Module and CPU windows, Turbo Debugger offers the following commands for setting simple breakpoints:

- You can set breakpoints on the entry points of all the functions in the currently loaded module, or on all member functions in a class, using the Breakpoints window SpeedMenu Group command. For more information on this command, see page 77.

- You can use the Add command on the Breakpoint window SpeedMenu to set breakpoints. This command opens the Breakpoint Options dialog box and positions the cursor on an empty Address input box. Enter an address or line number expression for which you'd like a breakpoint to be set.

  For example, if you'd like to set a breakpoint at line number 3201 in your source code, type #3201 in the input box. If the line of code is in a module other than the one displayed in the Module window, type a pound sign (#) followed by the module name, followed by another pound sign and the line number. For example: #OTHERMOD#3201.

  You can also access the Add command by typing an address directly into the Breakpoints window. After typing the first character of the address, the Breakpoint Options dialog box opens with the Address input box active.

Once you set a breakpoint, you can modify the action that it will take when it activates. The default action is "Break"—Turbo Debugger pauses the program's execution when the breakpoint is activated. For a list of possible breakpoint actions, see page 74.

Setting expression-true breakpoints

Expression-true breakpoints, like simple breakpoints, are set at specific program locations. However, unlike simple breakpoints, expression-true breakpoints have special conditions and actions added to their definitions.

Sometimes, you will not want a breakpoint to activate every time it's encountered, especially if the line containing the breakpoint is executed many times before the actual occurrence you're interested in. Likewise, you might not always want a breakpoint to pause the program's execution. With Turbo Debugger, you can specify when a breakpoint should activate and the actions it should take when it does activate.

Expression-true breakpoints are essentially simple breakpoints that have been customized. The following steps explain how to create an expression-true breakpoint:

1 Set a simple breakpoint (as described in the previous section).

2 Open the Conditions and Actions dialog box:

  1 Open the breakpoints window, and highlight the desired breakpoint in the List pane.

 **2** Choose Set Options from the SpeedMenu to open the Breakpoint Options dialog box.

Figure 5.2   The Breakpoint Options dialog box
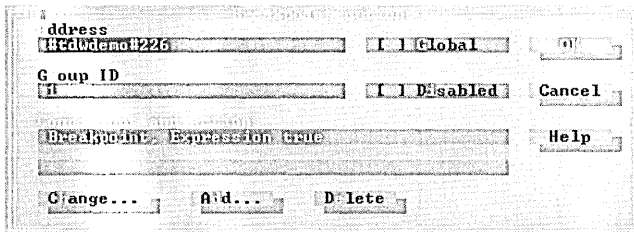
The Breakpoint Options dialog box contains commands that let you modify breakpoint settings. The Conditions and Actions list box displays the current settings of the selected breakpoint.

3 To modify a breakpoint's condition and action settings, click the Change button to open the Conditions and Actions dialog box.

Figure 5.3   The Conditions and Actions dialog box
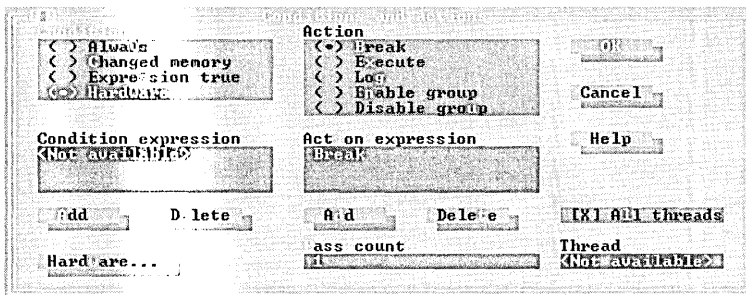
The Conditions and Actions dialog box lets you customize the conditions under which a breakpoint is activated, and the actions that take place once the conditions are met.

3 Select the Expression True radio button.

By default, the breakpoint's condition is set to *Always*—the breakpoint will activate each time it is encountered by the program's execution. Clicking the Expression True radio button specifies that the breakpoint should activate when an expression you supply becomes true (nonzero).

4 Enter the expression you want evaluated each time the breakpoint is encountered into the Condition Expression input box.

5 If needed, specify a *pass count* with the breakpoint settings.

The Pass Count input box lets you set the number of times the breakpoint condition set must be met before the breakpoint is activated. The default number is 1. The pass count is decremented only when the entire condition set attached to the breakpoint is true; if you set a pass count to *n*, the breakpoint is activated the *n*th time the entire condition set evaluates to true.

6 If you want to change the breakpoint's default action, click the desired Action radio button and enter any pertinent action expression into the Action Expression input box. Page 74 lists the different actions that you can associate with a breakpoint. For a list of possible breakpoint actions, see page 74.

See page 75 for details on entering breakpoint condition and action sets.

7 Choose OK or press *Esc* to exit the Conditions and Actions dialog box.

Setting changed-memory breakpoints

Changed-memory breakpoints (sometimes known as *watchpoints*) monitor expressions that evaluate to a specific data object or memory location. Set on specific lines of code, changed-memory breakpoints activate if a data object or memory pointer has changed value.

To set a changed-memory breakpoint, follow the same instructions for setting an expression-true breakpoint (described in the preceding section), with two exceptions:

1 In the Conditions and Actions dialog box, click the Changed Memory radio button instead of the Expression True radio button.

2 In the Condition Expression input box, enter an expression that evaluates to a memory location (a data object or memory pointer).

Note When your program's execution encounters a line that contains a changed-memory breakpoint, the condition expression is evaluated *before* the line of code gets executed. Because of this, carefully consider the placement of changed-memory breakpoints.

When entering an expression, you can also enter a count of the number of objects you want monitored. The total number of bytes watched in memory is the size of the object that the expression references times the object count.

For example, suppose you have declared the following C array:

```
int string[81];
```

You can watch for a change in the first ten elements of this array by entering the following item into the Condition Expression input box:

```
&string[0], 10
```

The area monitored is thus 20 bytes long—an **int** is 2 bytes and you instructed Turbo Debugger to monitor ten of them.

Setting global breakpoints

Global breakpoints are essentially expression-true or changed-memory breakpoints with the added characteristic that the breakpoint is monitored continuously during your program's execution. Because Turbo Debugger checks the breakpoint conditions after the execution of every line of source code or machine instruction, global breakpoints are excellent tools for pinpointing code that's corrupting data.

To create a global breakpoint, first set either a changed-memory or expression-true breakpoint, as described in the previous sections. Then, after you exit the Conditions and Actions dialog box, check the Global check box in the Breakpoint Options dialog box to specify that the breakpoint should be global.

When you create a global breakpoint, the Address input box in the Breakpoint Options dialog box reads `<not available>`; global breakpoints are not associated with specific program locations.

Normally, Turbo Debugger checks a global breakpoint after the execution of every line of source code. However, if you want Turbo Debugger to check the breakpoint after every machine instruction, press *F9* while the CPU window is active.

Because Turbo Debugger evaluates global breakpoints after the execution of every line of source code or machine instruction, these breakpoints greatly slow the execution of your program. Be moderate with your use of global breakpoints; use them only when you need to closely monitor the behavior of your program.

Although it's possible to create a global breakpoint with a condition of "Always," it's not recommended. Because the breakpoint condition is evaluated after the execution of each source line, a condition of "Always" will cause the breakpoint to activate after the execution of each line of code.

Global breakpoint shortcuts

The Breakpoints menu contains two commands that provide fast ways to set global breakpoints: Changed Memory Global and Expression True Global. When you set a breakpoint with either of these two commands, the breakpoint action is set to "Break" by default.

Changed Memory Global sets a global breakpoint that's activated when an area of memory changes value. When you issue this command, you're prompted for an area of memory to watch with the Enter Memory Address, Count input box. For information on valid expression types, see the preceding "Setting changed-memory breakpoints" section.

Expression True Global sets a global breakpoint that is activated when the value of a supplied expression becomes true (nonzero). When you select this command, you are prompted for the expression to evaluate with the Enter Expression for Conditional Breakpoint input box.

Setting hardware breakpoints

Hardware breakpoints, available with TDW and with TD32 when you debug Windows NT programs, take advantage of the special debugging registers of Intel 80386 (or higher) processors and certain hardware debugging boards. Hardware breakpoints let your hardware monitor the global breakpoints, so you don't have to use CPU-expensive software for that task.

Before you can set a hardware breakpoint in TDW, the TDDEBUG.386 device driver must be copied to your hard disk and loaded by your CONFIG.SYS file. If you want, Turbo Debugger's installation program can complete the installation process for you, or you can install it yourself by following the directions in the online file TD_HDWBP.TXT. When TDDEBUG.386 is properly installed, the Breakpoints field in TDW's File | Get Info dialog box reads *Hardware*; otherwise it reads *Software*.

To set a hardware breakpoint, choose the Hardware Breakpoint command from the Breakpoints menu. This command automatically checks the Global check box in the Breakpoint Options dialog box, chooses the Hardware radio button in the Conditions and Actions dialog box, and opens the Hardware Breakpoint Options dialog box. This dialog box contains all the hardware breakpoint settings, and is fully described in the online text file TD_HDWBP.TXT.

You can also create a hardware breakpoint by modifying an existing breakpoint:

1 Check the Global check box in the Breakpoint Options dialog box.

2 Open the Conditions and Actions dialog box and choose the Hardware radio button.

3 Click the Hardware button in the Conditions and Actions dialog box to access the Hardware Breakpoint Options dialog box.

4 Specify the hardware breakpoint settings and choose OK.

5 If needed, specify the action settings in the Conditions and Actions dialog box.

When you set a hardware breakpoint, its listing in the Breakpoint window's List pane will have an asterisk (*) displayed next to it.

Breakpoint actions

The Action radio buttons in the Conditions and Actions dialog box (Figure 5.3) specify the actions that you want a breakpoint to perform when it activates. Each of the following actions can be applied to any of the breakpoints you set.

Break

The Break button (default) pauses your program when the breakpoint is activated. When your program pauses, Turbo Debugger becomes active, and you can use its windows and commands to view your program's state.

Execute

The Execute button executes an expression that you enter into the Action Expression input box. For best results, use an expression that changes the value of a variable or data object.

By "splicing in" a piece of code before a given source line, you can effectively test a simple bug fix; you don't have to go through the trouble of compiling and linking your program just to test a minor change to a routine. Keep in mind, however, that you cannot use this technique to directly modify your compiled program.

Log

The Log button writes the value of an expression to the Log window. Enter the expression you want evaluated into the Action Expression input box. (For more information on the Log window, see page 81.)

This command is handy when you want to output a value each time you reach a specific place in your program (this technique is known as *instrumentation*). By creating a breakpoint with a Log action, you can log values each time the breakpoint activates.

For example, you can place a breakpoint at the beginning of a routine and set it to log the values of the routine arguments. Then, after running the program, you can determine where the routine was called from, and if it was called with erroneous arguments.

Note When you log expressions, be careful of expressions that unexpectedly change the values of variables or data objects.

Enable group

The Enable Group button causes a breakpoint to reactivate a group of breakpoints that have been previously disabled. Supply the group integer number to enable in the Action Expression input box. See page 77 for information on breakpoint groups.

Disable group

The Disable Group button lets you disable a group of breakpoints. When a group of breakpoints is disabled, the breakpoints are not erased, they are simply hidden from the debugging session. Supply the group integer number to disable in the Action Expression input box.

Setting breakpoint conditions and actions

You use the Conditions and Actions dialog box, shown in Figure 5.3, to specify when a breakpoint should activate, and what it should do when it does activate. Usually, you will enter a single condition or action expression for any given breakpoint. However, Turbo Debugger lets you create condition and action sets that contain multiple

expressions. In addition, a single breakpoint can have several condition and actions sets associated with it.

The following sections describe how to create complex breakpoint condition and action sets.

Creating breakpoint condition sets

When you create an expression-true or changed-memory breakpoint, you must provide a *condition set* so the debugger knows when to activate the breakpoint. A condition set consists of one or more expressions. For the breakpoint to activate, *every* expression in the condition set must evaluate to true. To create a condition set,

1 Choose either the Changed Memory or Expression True radio button.

2 Enter the condition expression into the Condition Expression input box.

3 Choose the Add button under the Condition Expression input box.

To enter more than one condition expression to a breakpoint's definition, repeat steps 2 and 3 until all your expressions have been added to the Condition Expression input box.

The Delete button located below the Condition Expression input box lets you remove the currently highlighted expression from the Condition Expression input box.

Creating breakpoint action sets

When you select either an Execute, Log, Enable Group, or Disable Group Action radio button, you must provide an *action set* so Turbo Debugger knows what to do when the breakpoint activates. An action set is composed of one or more expressions. To create an action set,

1 Choose either the Execute, Log, Enable Group, or Disable Group radio button.

2 Enter the action into the Action Expression input box.

3 Choose the Add button under the Action Expression input box.

To execute more than one expression when the breakpoint activates, repeat steps 1, 2, and 3, until all expressions have been added to the Action Expression input box.

Note If the Enable Group or Disable Group radio button is chosen, type the breakpoint group number into the Action Expression input box to reference the group of breakpoints you want enabled or disabled.

The Delete button located below the Action Expression input box lets you remove the currently highlighted expression from the action set.

When you have finished entering actions, choose OK on the Conditions and Actions dialog box.

Multiple condition and action sets

A single breakpoint can have several condition and action sets associated with it. To assign multiple condition and action sets to a single breakpoint, choose OK on the Conditions and Actions dialog box after you have entered the first series of conditions and actions. This closes the Conditions and Actions dialog box and returns you to the Breakpoint Options dialog box. From here, choose the Add button to enter a new set of conditions and actions.

Each condition and action set is evaluated in the order in which it was entered. If any condition set evaluates to true, then the actions associated with those conditions are performed.

To delete a condition and action set from a breakpoint's definition, select the Delete button on the Breakpoint Options dialog box.

The scope of breakpoint expressions

Both the conditions and actions of a breakpoint are controlled by the expressions you supply. Turbo Debugger evaluates breakpoint expressions with regards to the scope of the breakpoint location, not the scope of the location where you happen to be when you're entering the expressions.

Using scope-override syntax, you can access the values of any data objects that are defined when the breakpoint is encountered. However, breakpoints that reference data objects that are out of scope execute much slower than breakpoints that use only local or global variables. For a complete discussion of scopes and scope overrides, see "Accessing symbols outside the current scope" on page 101.

To modify a breakpoint that's set in a module that isn't currently loaded, you must use scope-overriding syntax to identify the module. However, when setting and modifying breakpoints, it's easiest to access the desired module using the View | Another | Module command.

Breakpoint groups

Turbo Debugger lets you group breakpoints together, allowing you to enable, disable, or remove breakpoints with a single action. In addition, you can set a group of break-points on all functions in a module or all member functions in a class with a single command.

The Group command on the Breakpoints window SpeedMenu activates the Edit Breakpoint Groups dialog box. Using this dialog box, you can create and modify breakpoint groups.

Figure 5.4 The Edit Breakpoint Groups dialog box
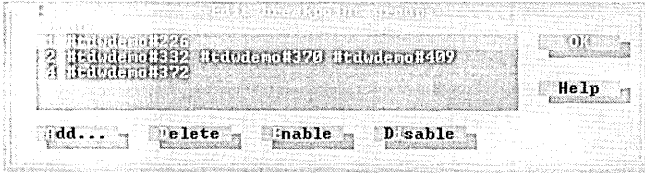
A breakpoint *group* is identified by a positive integer, generated automatically by Turbo Debugger or assigned by you. The debugger automatically assigns a new group number to each breakpoint as it's created. The group number generated is the lowest number not already in use. Thus, if the numbers 1, 2, and 5 are already used by groups, the next breakpoint created is automatically given the group number 3.

Once a breakpoint is created, you can modify its group status with the commands in the Edit Breakpoint Groups dialog box. You can also assign a breakpoint to a new or existing breakpoint group with the Group ID input box on the Breakpoints window's Breakpoint Options dialog box.

Creating breakpoint groups

The Add button on the Edit Breakpoint Groups dialog box activates the Add Group dialog box. The Add Group dialog box contains one list box and a set of radio buttons that let you add all functions in a single module, or all member functions in a class, to a breakpoint group.

The Module/Class list box displays a list of the modules or classes contained in the currently loaded program. Highlight the desired module or class and press OK to set breakpoints on all routines in that module or class. All breakpoints set in this manner are collected into a single breakpoint group.

Using the two radio buttons in the Add Group dialog box, you can select the type of routines that are displayed in the Module/Class list box:

- The Modules radio button selects all modules contained in the current program.
- The Classes radio button selects all the Borland C++ classes contained in the current program.

Deleting breakpoint groups

The Delete button on the Edit Breakpoint Groups dialog box removes the group currently highlighted in the Groups list box. Use this command with caution; all breakpoints in the selected group, along with their settings, are permanently erased by this command.

Enabling and disabling breakpoint groups

The Edit Breakpoint Groups dialog box contains two commands for enabling and disabling breakpoint groups. The Enable button activates a breakpoint group that has been previously disabled.

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

In addition to the two commands on the Edit Breakpoint Groups dialog box, you can enable and disable breakpoint groups through the action settings of breakpoints. For information on this feature, see page 75.

Navigating to a breakpoint location

The Inspect command on the breakpoint window SpeedMenu opens the Module or CPU window, and positions the display at the location of the breakpoint that's highlighted in the List pane.

Note   You can also invoke this command by pressing *Enter* once you have highlighted the desired breakpoint in the List pane.

Enabling and disabling breakpoints

Checking the Disabled check box in the Breakpoint Options dialog box masks the current breakpoint, hiding it until you want to reenable it by unchecking this box. When the breakpoint is reenabled, all settings previously made to the breakpoint become effective.

Disabling a breakpoint is useful when you have defined a complex breakpoint that you don't need just now, but will need later. It saves you from having to delete the breakpoint, and then reenter it along with its complex conditions and actions.

Removing breakpoints

You can erase existing breakpoints from either the Breakpoints window SpeedMenu, or the Breakpoint menu.

The Remove command on the Breakpoint window SpeedMenu erases the breakpoint currently highlighted in the List pane. *Del* is the hot key for this command.

The Delete All command, found on both the Breakpoint menu and the Breakpoints window SpeedMenu, removes all the currently set breakpoints, including global breakpoints and those set at specific addresses. Use this command with caution; its effects cannot be reversed.

Setting breakpoints on C++ templates

Turbo Debugger supports the placement of breakpoints on C++ templates, function templates, and template class instances and objects.

The method you use to set template breakpoints affects the way the breakpoints are set:

* If you set a breakpoint on a template by pressing *F2* while in the Module window, breakpoints are set on all class instances of the template. This lets you debug the overall template behavior.
* If you press *Alt+F2* to set a template breakpoint, the Breakpoint Options dialog box activates, and you can enter the address of a template into the Address input box. A dialog box opens that lets you choose a specific class instance for the breakpoint.
* You can set a breakpoint on a specific class instance of a template through the CPU window. Position the cursor on a line of template code in a single class instance and press *F2* to set a breakpoint on that class instance only.

You remove template breakpoints just as you remove other breakpoints; position the cursor on the breakpoint in the Module window and press *F2*. All breakpoints on associated class instances are deleted.

You can remove specific template breakpoints by deleting them from the CPU window. Position the cursor on the desired breakpoint in the CPU window and press *F2* to it.

Setting breakpoints on threads

Programs written for the Windows NT operating system consist of one or more executable "threads." When debugging a Windows NT program, you can set a breakpoint on a specific thread, even though the code at the breakpoint location is shared by multiple threads.

When you set a breakpoint in a Windows NT program, by default, the breakpoint is set for all program threads. To specify that the breakpoint should be checked for a single thread only,

1 Highlight the desired breakpoint in the Breakpoints window's List pane.

2 Choose the List pane Set Options SpeedMenu command.

3 Click the Change button in the Breakpoint Options dialog box to open the Conditions and Actions dialog box. Set the breakpoint's conditions and actions as needed.

By default, the All Threads check box is checked, indicating that the breakpoint is set for all active threads.

4 Clear the All Threads check box; the Threads input box becomes available.

5 Type the Windows NT thread number you want to monitor into the Threads input box.

To obtain a Windows NT thread number, open the Threads window with the View | Threads command. The Threads List pane displays all currently active threads, listing them by the Windows NT thread number and their given name.

6 Choose OK to confirm your breakpoint settings.

For more information on debugging threads, see "Debugging multi-threaded programs" on page 139.

The Log window

The Log window keeps track of the significant events that occur during your debugging session. To open the Log window, choose View | Log.

Figure 5.5   The Log window

```
Hwnd:2a00e0 wParam:0001 lParam:008f005b (0201) WM_LBUTTONDOWN
Breakpoint at #tdwdemo#171
Breakpoint at #tdwdemo#226 "Slope == 1" true
Breakpoint at #tdwdemo#226 "Slope == 1" true
Hwnd:2a00e0 wParam:0001 lParam:016a011b (0201) WM_LBUTTONDOWN
Breakpoint at #tdwdemo#171
```
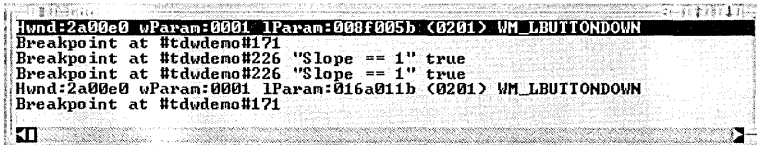
By default, the Log window can list 50 lines of text. However, you can change the default using TDWINST.EXE or TDINST32.EXE.

The following debugging actions are tracked by the Log window:

* When your program pauses, the program location is recorded in the Log window.

* When you use the Log window's Add Comment command, your comment gets added to the Log window.

* When a breakpoint activates that logs an expression, the value of the expression is written to the Log window.

* When you choose the Edit | Dump Pane to Log command, the contents of a pane or window are recorded in the Log window.

* When you use the Display Windows Info command on the Log window SpeedMenu, the global or local heap information, or the list of program modules is written to the Log window.

* When you set Send to Log Window to *Yes* from the Windows Messages window, all window messages sent to that window are copied to the Log window.

The Log window SpeedMenu

The commands in the Log window SpeedMenu let you write the log to a disk file, stop and start logging, add a comment to the log, clear the log, and write information about a Windows program to the log.

```
Open log file...
Close log file
Logging      Yes
Add comment...
Erase log
Display Windows info...
```
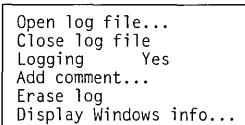
Open Log File

The Open Log File command causes all lines written to the Log window to also be written to a disk file. When you choose this command, a dialog box prompts you for the name of the disk file. By default, the log file's name is the name of your program, followed by a .LOG extension.

When you open a log file, all the lines already displayed in the Log window 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 doesn't contain the lines already displayed in the Log window, choose Erase Log before choosing Open Log File.

Close Log File

The Close Log File command closes the log file that you opened with the Open Log File command.

Logging

The Logging command enables and disables the writing of events to the Log window. Use this command to control when events are logged. When logging is turned off, the Log window's title bar displays *Paused*.

Add Comment

Add Comment lets you insert comments into the Log window. When you choose this command, a dialog box opens, prompting you for a comment.

Erase Log

Erase Log clears the Log window. This command affects only what's in memory; the log disk file is not erased by this command.

Display Windows Info

The Display Windows Info command, available only with TDW, displays the Windows Information dialog box. This dialog box lets you list global heap information, local heap information, or the list of modules making up your application. See page 144 in Chapter 10 for more information on this feature.

Examining and modifying data

The data in your program consists of global variables, local variables, and defined constants. Turbo Debugger provides the following ways to view and modify the data that your program processes:

- The Watches window displays the current values of variables and expressions.

- The Variables window displays your program's local and global variables.

- The Inspector windows display the values of program data items, including compound data objects.

- The Stack window displays the current functions and procedures located on the stack, including their argument values.

- The Evaluate/Modify command evaluates expressions and lets you assign new values to variables.

- The Function Return command displays the value that the currently executing function is about to return.

The Watches window

The Watches window provides the easiest way to keep track of your program's data items. In the Watches window, you list the program variables and expressions whose values you want to track. Each time your program's execution pauses, Turbo Debugger evaluates all the items listed in the window and updates their displayed values.

With the Watches window, you can watch the value of both simple variables (such as integers) and compound data objects (such as arrays). In addition, you can watch the values of calculated expressions that do not refer directly to memory locations. For example, you could watch the expression x * y + 4.

Figure 6.1   The Watches window

```
 lParam                  int 8847843 (0x8701E3)
 oldy                    int 135 (0x87)
 oldx                    int 483 (0x1E3)
 CurrentShape            int 1 (0x1)
 thisShape[CurrentPoint].Points  struct tagRECT (690,305,0,0)
 thisShape[CurrentPoint].Slope int 1 (0x1)
```

Expressions that you enter as watches are listed on the left side of the Watches window, and their corresponding data types and values appear on the right. The values of items in compound data objects (such as arrays and structures) appear with their values between braces ({ }). The Watches window truncates any expressions or values that do not fit into the window.

Creating watches

To create a watch, choose one of the following commands:

- The Data | Add Watch command
- The Module window SpeedMenu Watch command
- The Variable window SpeedMenu Watch command
- The Watches window SpeedMenu Watch command

When you choose a command to create a watch, Turbo Debugger opens the Enter Expression to Watch dialog box. Enter a variable name or expression, and press *Enter* to add it to the Watches window.

If the cursor is on a variable in the Module window, that variable is automatically added to the Watches window when you choose the SpeedMenu Watch command. The same is true for expressions selected using *Ins* and the arrow keys.

Unless you use scope override syntax (as described in Chapter 7), Turbo Debugger evaluates watch expressions with regards to the current instruction pointer. If a watch expression contains a symbol that isn't accessible from the currently active scope, the value of the expression is undefined, and is displayed as four question marks (????).

When you enter expressions into the Watches window, you can use variable names that aren't yet defined; Turbo Debugger lets you set up a watch expression before its scope becomes active. This is the only situation in Turbo Debugger where you can enter an expression that can't be immediately evaluated.

Be careful when you enter expressions into the Watches window. If you mistype the name of a variable, Turbo Debugger won't detect the mistake because it assumes the variable will become available at a later time during program execution.

When you're tracing inside a member function, you can add the **this** pointer to the Watches window. Turbo Debugger knows about the scope and presence of the **this** pointer. You can evaluate **this** and follow it with format specifiers and quantifiers.

The Watches window SpeedMenu

The Watches window SpeedMenu contains all the commands needed to manage the items in the window:

```
Watch...
Edit...
Remove
Delete all

Inspect
Change...
```
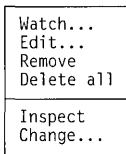
Watch
The Watch command prompts you for a variable name or expression to add to the Watches window. Unless you explicitly enter a scope, Turbo Debugger evaluates the expression with regards to the current cursor location.

Edit
Edit opens the Edit Watch Expression dialog box, letting you modify the expression currently highlighted in the Watches window. When you've finished editing the expression, press *Enter* or click the OK button.

You can also invoke this command by pressing *Enter* after you've highlighted the watch expression you want to change.

Remove
The Remove command removes the currently selected item from the Watches window.

Delete All
Delete All removes all expressions from the Watches window. This command is useful if you move from one area of your program to another, and the variables you were watching are no longer relevant.

Inspect
The Inspect command opens an Inspector window that shows the details of the currently highlighted watch. This command is useful when the watch expression is a compound data object, or if the expression is too long to be fully displayed in the Watches window.

Change
Use the Change command to modify the value of the currently highlighted variable in the Watches window. When you enter a new value into the Enter New Value dialog box, Turbo Debugger performs any necessary type conversion, exactly as if the assignment operator had been used to change the variable.

The Variables window

The Variables window displays the names and values of all the local and global variables accessible from the current program location. You can use this view to examine and change the values of variables, and to view the variables local to any function that has been called. To access this window, choose View | Variables.

Figure 6.2   The Variables window
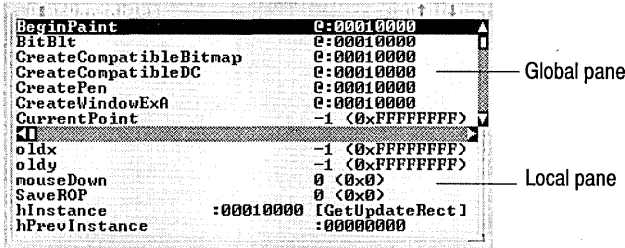
The Variables window has two panes:

*   The Global pane shows all the global symbols in your program.

*   The Local pane shows all the static symbols in the module and all the symbols local to the current function or procedure.

Both panes display the variable names on the left and their data types and values on the right. If Turbo Debugger can't resolve a symbol's data type, it displays four question marks (????).

The Variable window SpeedMenus

Each pane of the Variables window has its own SpeedMenu. Both menus contain Inspect, Change, and Watch commands; the Local pane also has the Show command.
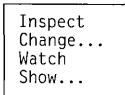
```
Inspect
Change...
Watch
Show...
```

Inspect
The Inspect command opens an Inspector window that displays the contents of the currently highlighted global, local, or static symbol.

If you inspect a global variable whose name matches a local variable's name, Turbo Debugger displays the value of the global variable, not the local variable. This behavior is slightly different from the usual behavior of Inspector windows, which normally display values from the point of view of your current program location. This difference gives you a convenient way to look at global variables whose names are also used as local variables.

If you issue the Inspect command on an entry that's a routine name (in the Global pane), Turbo Debugger activates the Module window and places the cursor on the routine's

source code. If Turbo Debugger can't find the source code, or if the file wasn't compiled with debug information, a CPU window opens, showing the disassembled instructions.

Change

The Change command opens the Change dialog box so you can modify the value of the currently highlighted symbol. Turbo Debugger performs any necessary data type conversion exactly as if the assignment operator for your current language had been used to change the variable.

You can also access the Change dialog box by choosing the SpeedMenu Inspect command and typing the new value into the Inspect window.

Watch

The Watch command opens a Watches window and adds the currently highlighted symbol to that window.

The Watches window doesn't keep track of whether the variable is local or global. If you insert a global variable using the Watch SpeedMenu command, and later encounter a local variable by the same name, the local variable takes precedence whenever you're in the scope of the local variable. The Watches window always displays the value of a variable from the point of view of your current program location.

Show

The Local pane's Show command brings up the Local Display dialog box. The radio buttons in this dialog box enable you to change the scope of the variables displayed in the Local pane and the module from which these variables are selected:

Static   Show only static variables.

Auto   Show only variables local to the current block.

Both   Show both static and local variables (default).

Module   Change the current module. This command brings up a dialog box showing the list of modules for the program, from which you can select a new program module.

Viewing variables from the Stack window

Using the Stack window, you can examine the variables of any routine that's located on the stack, including the different version of a recursive routine. To do so, open the Stack window and highlight the routine you want to examine. Next, press *Alt+F10*, and choose Locals. The Static pane of the Variables window opens, showing the argument values of the selected routine.

Inspector windows

Inspector windows are the best way to view data items because Turbo Debugger automatically formats Inspector windows according to the type of data being displayed.

Inspector windows display data differently for scalars (for example, **char** or **int**), pointers (**char ***), structures, arrays (**long** ×[4]), and functions. In addition, there are special Inspector windows for C++ classes (for a description of class Inspector windows, see Chapter 11). In the sections that follow, Inspector windows are described as they appear when you inspect scalar, pointer, structure and union, array, and function data types.

Inspector windows are especially useful when you want to examine compound data objects, such as arrays and linked lists. Because you can inspect individual items displayed in an Inspector window, you can "walk" through compound data objects by opening an Inspector window on a component of the compound object.

Inspector windows also offer a quick way to view the raw bytes of a data item. To do so, choose View | Dump when an Inspector window is active. The Dump window opens with the cursor positioned on the data displayed in the Inspector window.

Opening Inspector windows

Although you cannot open Inspector windows from the View menu, you can open them from the following debugger locations:

* The Data | Inspect command
* The Module window SpeedMenu
* Watches window SpeedMenu
* Variables window SpeedMenu
* Inspector window SpeedMenu

When you open an Inspector window, the Enter Variable to Inspect dialog box prompts you for an expression to inspect. After entering a variable name or expression, an Inspector window opens, displaying the value of the expression entered.

If the cursor is on a program symbol when you issue the Inspect command, or if you select an expression using *Ins* and the arrow keys, Turbo Debugger automatically places the selected symbol in the input box.

When you open an Inspector window, the title of the window displays the expression that's being inspected. The first item listed in an Inspector window is always the memory address of the data item that's detailed in the rest of the window, unless the data item is a constant or is a variable that has been optimized to a register.

Scalar Inspector windows

Scalar Inspector windows show the values of simple data items, such as **char, int, long,** and so on.

Scalar Inspector windows have two lines of information. The first line contains the address of the variable. The second line displays the type of the scalar on the left and the current value of the variable on the right. The value can be displayed as decimal, hexadecimal, or both. Normally, however, the value is displayed first in decimal, followed by the hexadecimal value enclosed in parentheses.

Figure 6.3   A C scalar Inspector window


```
⊡:00040074
int                       569  <0x239>
```

If the variable being inspected is of type **char**, the equivalent character is displayed to the left of the numeric values. If the present value doesn't have a printing character equivalent, Turbo Debugger displays a backslash (\) followed by the hexadecimal value that represents the character value.

Pointer Inspector windows

Pointer Inspector windows show the values of variables that point to other data items. Pointer Inspector windows have a top line that contains the address of the variable, followed by detailed information regarding the data pointed to. Pointer Inspector windows also have a lower pane indicating the data type to which the pointer points.

Figure 6.4   A C pointer Inspector window


```
Register : :00770104
bKillThrd                      29753468  <0x1C6007C>
hWnd                          :01C60098
vects ((65536,-2147483648),-2147483648,-2147483648,-214
ClientRect                    (7864324,132250,1990215942,168)
hThread                       :000000B4
threadno                      144  <0x90>
struct _Params *
```
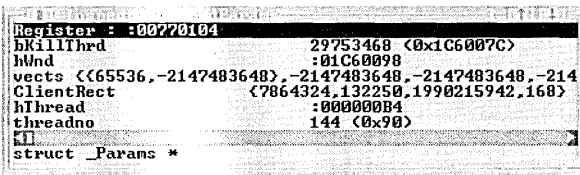
If the value pointed to is a compound data object (such as a structure or an array), Turbo Debugger enclosed the values in braces ({ }) and displays as much of the data as possible.

If the pointer appears to be pointing to a null-terminated character string, Turbo Debugger displays the value of each item in the character array. The left of each line displays the array index ([0], [1], [2], and so on), and the values are displayed on the right. When you're inspecting character strings, the entire string is displayed on the top line, along with the address of the pointer variable and the address of the string that it points to.

In addition, you can use the Range command to cause the Inspector window to display multiple lines of information. This is helpful for Borland C++ programmers who use pointers to point to arrays of data structures as well as to single items. For example, suppose you have the following code:

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

To see what *arrayp* points to, use the Range local command on *arrayp*, and specify a starting index of 0 and a range of 10. If you had not done this, you would have seen only the first item in the array.

Structure and Union Inspector windows

Structure and Union Inspector windows show the values of members contained in compound data objects.

Figure 6.5   A C Structure and Union Inspector window

```
@:00041454
Points                          <253,144,0,0>
PenWidth                        3 <0x3>
theShape                        1 <0x1>
PenColor                        255 <0xFF>
Slope                           0 <0x0>
struct SSHAPE
```
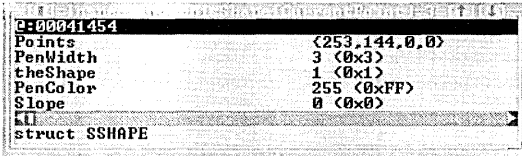
Structure and Union Inspector windows have two panes:

* The top pane displays the address of the data object, followed by lines listing the names and values of the data members contained in the object. This pane contains as many lines as are necessary to show the entire data object.

* The lower pane consists of one line. If you highlight the address of the data object in the top pane, the lower pane displays the type of the data object (either structure or union) along with its name. Otherwise, the lower pane displays the data type of the object member highlighted in the top pane.

The Structure and Union Inspector window shown in Figure 6.5 was taken from a program containing the following code:

```
struct linfo {
    unsigned int count;
    unsigned int firstletter;
} letterinfo [26];
```

Array Inspector windows

Array Inspector windows show the values of the elements contained in arrays. These windows contain a line for each element in the array. The left side of each line shows the index of the array element, and the right side shows the element's value. If the value is a compound data object, Turbo Debugger displays as much of the object as possible.

Figure 6.6   A C array Inspector window

```
@:00041454
[0]      <<253,144,0,0>,3,1,255,-1>
[1]      <<127,67,277,166>,3,3,255,0>
[2]   <<323,142,417,224>,3,3,255,0>
[3]      <<358,56,0,0>,3,2,255,0>
[4]      <<275,54,0,0>,5,1,0,0>
[5]             <<0,0,0,0>,0,0,0,0>
struct SSHAPE
```
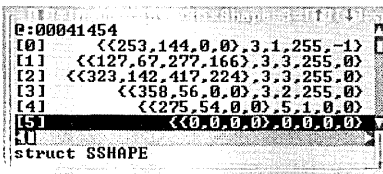
As an example of using the Array Inspector window, suppose your program contains the following statement:

```
MyCounter[TheGrade]++;
```

Pressing *Ctrl+I* when the cursor is at *MyCounter* in the Module window opens an Inspector window that displays the contents of the entire array. However, if you press *Ctrl+I* after selecting the entire array name and index (using *Ins* and the arrow keys), Turbo Debugger opens an Inspector window that displays only the single element of the array.

You can also use the Range SpeedMenu command to show any portion of an array.

Function Inspector windows

Function Inspector windows show the memory address of the function, followed by the arguments with which a function is called. To inspect a function, use the function's name without parenthesis or arguments.

Figure 6.7   A C function Inspector window
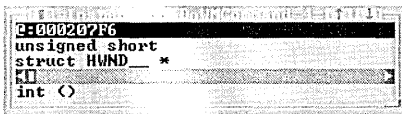
Function Inspector windows also give you information about the return type and calling conventions of the function you're inspecting. The return type is displayed in the lower pane.

The Inspector window SpeedMenu

The Inspector window SpeedMenu offers a variety of commands:

```
Range...
Change...

Inspect
Descend
New expression...
Type cast...
```
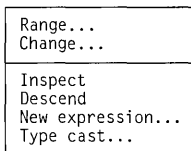
Range
The Range command sets the starting element and number of elements that you want to view in an array. Use this command when you have a large array and you need to examine only a subset of its elements.

Change
The Change command lets you change the value of the currently highlighted item to the value you enter in the Enter New Value dialog box. Turbo Debugger performs any necessary casting exactly as if an assignment operator had been used to change the variable.

Inspect
Inspect opens a new Inspector window listing the highlighted item in the current Inspector window. Use this command if you're inspecting a compound data object

(such as a linked list), and you want to open a new Inspector window on one of the items in the object. If the current Inspector window is displaying a function, issuing the Inspect command activates the Module window, and shows you the source code for that function.

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

To return to the previous Inspector window, press *Esc*. If you are through inspecting a data structure and want to remove all the Inspector windows, use the Window | Close command or its hot key, *Alt+F3*.

Descend
The Descend command works like the Inspect SpeedMenu command, except that it *replaces* the current Inspector window with the new item you want to examine. Using this command reduces the number of Inspector windows onscreen.

Note   When you use Descend to expand a data structure, you can't return to previous views of the data like you can when you use the Inspect command. Use Descend when you want to work your way through a complicated data structure, and don't need to return to a previous view of the data.

New Expression
You can inspect a different expression by selecting the New Expression command. The data in the current Inspector window is replaced with the data relating to the new expression you enter.

Type Cast
The Type Cast command lets you specify a different data type (for example **int, char ***, **gh2fp, lh2fp**, and so on) for the item being inspected. Typecasting is useful if the Inspector window contains a symbol for which there is no type information, and when you want to explicitly set the type for untyped pointers. Page 146 explains how to use the **gh2fp** and **lh2fp** data types.

The Stack window

The Stack window deciphers the call stack and lists all active functions and their argument values in a readable format. The most recently called function is displayed at the top of the list, followed by its caller, then by that caller's caller, and so on. This display of called functions continues down to the first function in the calling sequence, which is displayed at the bottom of the list. Functions that have been called from DLLs and Windows kernel code are also listed in the Stack window, even though they might not have symbolic names associated with them.

The View | Stack command opens the Stack window:

Figure 6.8   The Stack window
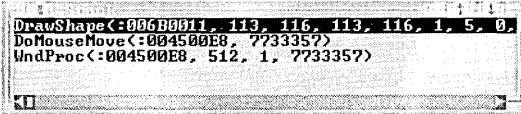
The Stack window also displays the names of member functions. Each member function is prefixed with the name of the class that defines the function; for example,

```
shapes::acircle(174, 360, 75.0)
```

The Stack window SpeedMenu

The Stack window SpeedMenu contains the following commands:

```
Inspect...
Locals
```
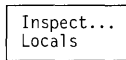
Inspect
The Inspect command opens a Module window and positions the cursor at the active line in the currently highlighted function. If the highlighted function is at the top of the call stack (the most recently called function), the Module window shows the location of the current instruction pointer. If the highlighted function is not at the top of the call stack, the cursor is positioned on the line following the related function call.

You can also invoke this command by pressing *Enter* when the highlight bar is positioned over the desired function.

Locals
The Local command opens a Variables window that shows the symbols that are local to the current module and to the currently highlighted function.

When a function calls itself recursively, the Stack windows shows multiple instances of the function. By positioning the highlight bar on an instance of that function, you can use the Locals command to look at the local variables of a particular function call.

The Evaluate/Modify command

The Evaluate/Modify command on the Data menu opens the dialog box shown in Figure 6.9. The Expression input box automatically contains the text located at the cursor position, or the expression that you have selected using *Ins* and the arrow keys. When you choose the Eval button, the expression in the Expression input box is evaluated, and the result is placed in the Result field.

Figure 6.9   The Evaluate/Modify dialog box
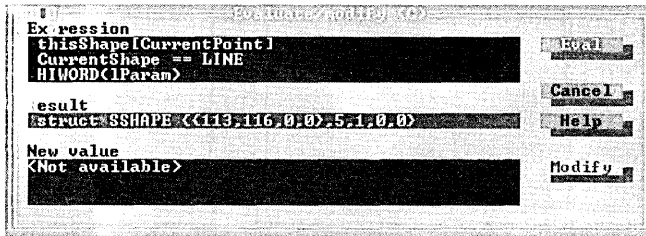
The Evaluate/Modify dialog box contains the following three fields:

Table 6.1   Evaluate/Modify dialog box fields

Field	Description
Expression	You enter expressions to evaluate into the Expression input box. This input box contains a history list of all the expressions you enter.
Result	The Result field displays the result of the expression evaluation.
	Data strings that are longer than the width of the Result input box are terminated by an arrow (▸). You can see more of the string by scrolling to the right.
New Value	The New Value input box is where you enter a new value for the expression highlighted in the Evaluate input box. This entry takes effect when you choose the Modify button.
	If the expression can't be modified, this box reads <Not available>, and you can't move your cursor into it.

When you evaluate expressions, be careful of Borland C++ expressions that cause side effects. See "Expressions with side effects" on page 100 for more information on side effects.

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

To call member functions from the Evaluate/Modify dialog box, type the instance name followed by a dot, followed by the member function name, followed by the actual parameters (or empty parentheses if there are no parameters). You cannot, however, execute constructors or destructors from the Evaluate window.

For example, suppose your program contains the following code:

```
class point {
   public:
      int x, y, visible;
      point ();
      ~point();
      int Show();
      int Hide();
      void MoveTo(int NewX, int NewY);
};

point APoint;
```

You could then enter any of the following expressions in the Evaluate window:

Expression	Possible result
APoint.x	int 2 (0×2)
APoint	class point {1,2,27489}
APoint.MoveTo	void () @6B61:0299
APoint.Show	int () @6B61:0285
APoint.Show()	int 1 (0×1)

Function Return command

The Function Return command, located on the Data menu, displays the value that the currently executing function is about to return. You should use this command only when the current function is about to return to its caller.

The return value is displayed in an Inspector window, so you can easily examine return values that are pointers to compound data objects. This command saves you from having to use the CPU window to examine return values that are placed in registers.

Evaluating expressions

An *expression* is a sequence of program symbols, constants, and language operators that can be evaluated to produce a value. To be valid, an expression must conform to the rules and syntax of the selected language. Turbo Debugger's *expression evaluator* ensures that the expressions you enter are valid, and it evaluates them to produce a value.

In this chapter, you'll learn how to select an expression evaluator, how to formulate different types of expressions, and how to use scope override syntax to explicitly reference a program symbol.

Turbo Debugger's expression evaluator

When you enter an expression into one of Turbo Debugger's input boxes, the expression is passed to the selected expression evaluator. The evaluator checks the expression's syntax and resolves the values of any symbols used in the expression. If all the symbols can be resolved and the syntax of the expression conforms to the syntax of the expression evaluator, then Turbo Debugger evaluates the expression and returns its calculated value.

Selecting an evaluator

To select an expression evaluator, choose Options | Language to open the Expression Language dialog box. The four radio buttons in this dialog box let you choose an expression evaluator for your debugging session:

- Source
- C
- Pascal
- Assembler

By default, Turbo Debugger selects the Source radio button, which automatically determines which expression evaluator to use (either C, Pascal, or Assembler) according to the source language of the current module being debugged. If Turbo Debugger can't determine the module's language, it uses the expression rules for inline assembler.

Usually, you can let Turbo Debugger choose the expression evaluator. Sometimes, however, you'll find it useful to explicitly set the evaluator. For example, if you're debugging an assembler module that's called from another language, you might want to override the default evaluator.

Also, by manually setting the expression evaluator, you can enter expressions in the language of your choice. Turbo Debugger can successfully resolve expressions that are not in your program's language; the debugger retains information about the original source language and handles the conversions appropriately.

Expression limitations

For the most part, Turbo Debugger supports the full language syntax for C, C++, Object Pascal, and assembler expressions. However, there are certain language statements and expressions that are out of context while debugging. For example, control structures such as if/then/else statements cannot be entered into the debugger. In addition, data and function declarations, and expressions that attempt to assign values to more than a single variable will be flagged as errors. Also be aware that the debugger cannot call object constructors or destructors. For complete details on language syntax, refer to the manuals accompanying your Borland language product.

Types of expressions

Although you'll usually use expressions to access the values of program symbols, calculate values, and change the values of data items, you can also use expressions to:

- Specify hexadecimal values
- Specify memory addresses
- Enter program line numbers
- Enter byte lists
- Call functions and routines

Specifying hexadecimal values

While debugging, you might need to supply a hexadecimal value to Turbo Debugger. For example, you'll need to use a hexadecimal address to specify a memory location.

The notation used to specify hexadecimal values depends upon the expression evaluator you've selected, as shown in the following table:

Table 7.1 Hexadecimal notation

Language	16-bit	32-bit
Assembler	0*nnnn*h	*nnnnnnnn*h
C	0×*nnnn*	0×*nnnnnnnn*
Pascal	$*nnnn*	$*nnnnnnnn*

In assembler, hexadecimal numbers starting with A to F must be prefixed with a zero.

Specifying memory addresses

To specify a 16-bit offset or a 32-bit address, preface the hexadecimal address location with the formats described in Table 7.1.

If you're debugging 16-bit code, you can use segment:offset notation to specify an exact memory location. When doing so, use the hexadecimal format of the expression evaluator you've selected. The following table gives examples:

Table 7.2 Segment:Offset address notation

Language	Format	Example
Assembler	*nnnn*h	1234h:0B010h
C	0×*nnnn*	0×1234:0×0010
Pascal	$*nnnn*	$1234:$0010

In assembler, hexadecimal numbers starting with A to F must be prefixed with a zero.

Entering line numbers

If you're using the C or Assembler expression evaluator, you can use an expression to specify a program line number. To do so, precede the decimal line number with a cross hatch (#). For more information on this notation, see "Overriding scope in C, C++, and assembler programs" on page 102.

Entering byte lists

In Turbo Debugger, several commands require that you enter a list of bytes. For example, the Search command in the File window requires a byte list as the search criteria when it's displaying a file in hexadecimal format.

A *byte list* can be any mixture of scalar (non-floating-point) numbers and strings in the syntax of the current expression evaluator. Scalars are converted into a corresponding byte sequence. For example, the C **long** value *123456* becomes a 4-byte hex quantity *40 E2 01 00*.

The following table gives an example of a byte list for each of the expression evaluators:

Table 7.3 Byte lists

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

Calling routines

You can call routines from expressions exactly as you do in your source code. Turbo Debugger executes your program code with the routine arguments that you supply. This can be a useful way to quickly test the behavior of a routine; simply call the routine with different arguments and check the return values after each call.

If you make specific calls to routines while debugging, be aware that certain routines can have the side effect of changing program data values. After calling such a routine, you cannot count on your program behaving normally during the rest of your debugging session. For more information on side effects, see the following section.

Note If you call a Pascal routine that doesn't use parameters, you must follow the routine name with empty parentheses to indicate to the debugger that you are indeed making a call to the routine. Without the parentheses, Turbo Debugger will return the address of the routine.

Expressions with side effects

An expression is said to have a *side effect* when the evaluation of the expression changes the value of a data item. Using expressions to change the values of data items can be a powerful debugging technique. However, there are times when you should avoid such expressions. For example, the expressions you enter for breakpoint conditions must not contain side effects.

Expressions that generate side effects are:

• Expressions that use assignment operators (=, +=, and so on).
• Expressions that use the C increment (++) and decrement (– –) operators.

A more subtle type of side effect occurs when you call a function or procedure that changes the value of a data item. Because you can't always tell which functions change the values of program variables, all functions are considered to generate side effects.

Format specifiers

When Turbo Debugger displays the value of an expression, it displays the value in a format based on the value's data type. To change the default display format of an

expression, follow the expression with a comma and with one of the following format specifiers:

Table 7.4 Expression format specifiers

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 extended character set.
d	Displays an integer as a decimal number.
f[#]	Displays the number in decimal notation. An integer following the specifier indicates the number of digits to the right of the decimal point. If you don't supply this number, as many digits as necessary are used to represent the number.
m	Displays a memory-referencing expression as hex bytes.
md	Displays a memory-referencing expression as decimal bytes.
p	Displays a raw pointer value, showing segment as a register name if applicable. Also shows the object pointed to. This is the default if no format control is specified.
s	Displays an array or a pointer to an array of characters as a quoted character string.
x or h	Displays a value as a hexadecimal number.

Turbo Debugger ignores any format specifier that cannot be applied to the expression's data type.

Note In addition to a format specifier, you can supply a *repeat count* to indicate that the expression relates to repeating data item such as an array or pointer. To specify a repeat count, follow the expression with a comma, the repeat count, another comma, and the format specifier.

Accessing symbols outside the current scope

The *scope* of a symbol is the area in your program in which the symbol can be referenced. The *current scope* is the area in your program in which defined symbols can be referenced. Usually, the current scope is defined with regards to the location of the instruction pointer. This section describes:

- How Turbo Debugger searches for symbols
- The implied scope for expression evaluation
- Scope override syntax
- Scope and DLLs

How Turbo Debugger searches for symbols

When you enter an expression that contains symbols, Turbo Debugger tries to resolve the symbols by searching the following locations in the order shown:

1 The symbols located in the current function's stack.
2 The symbols local to the module or unit containing the current function.
3 The global symbols for the entire program.

4 The global symbols of any loaded DLLs, starting with the earliest loaded DLL.

However, using *scope override syntax*, you can access any program symbol that has a defined value within the currently loaded executable module, including symbols that are private to a function and symbols that have conflicting names. By specifying an object module, a file within a module, a routine name, or a line number, you can give explicit directions to where a symbol can be found.

Implied scope for expression evaluation

Whenever you enter an expression into Turbo Debugger, the expression is evaluated according to the current scope. However, instead of using the instruction pointer to define the current scope, Turbo Debugger uses the *current cursor position* to determine the scope of an expression. Thus, you can set the scope in which an expression will be evaluated by moving the cursor to a specific line in the Module window. You can also change the scope of evaluation by either moving through the Code pane of a CPU window, moving the cursor to a routine in the Stack window, or moving the cursor to a routine name in a Variables window.

If you change the scope from where Turbo Debugger paused your program, you might get unexpected results when you evaluate expressions. To ensure that expressions are evaluated relative to the current position of your program, use the Origin command in the Module window to return the text to the location of the instruction pointer.

Scope override syntax

Turbo Debugger uses different syntax to override the scope of a symbol, depending on the language evaluator specified in the Options | Language dialog box:

- With the C, C++, and Assembler evaluators, use a cross hatch (#) to override scope. (The following section provides more information.)

- With the Pascal evaluator, use a period (.) to override scope. (See page 104 for more information.)

Overriding scope in C, C++, and assembler programs

You can use either of the following two types of scope overriding syntax with C, C++, and assembler expressions (items enclosed in brackets ([]) are optional):

```
[#module[#filename.ext]]#linenumber[#symbolname]
[#module[#filename.ext]#][functionname#]symbolname
```

The following rules also apply to the scope overrides:

- If you don't specify an object module, the currently loaded object module is assumed.

- If you use a file name in a scope override statement, it must be preceded by an object module name.

- If a file name has an extension (such as .ASM, .C, or .CPP), you must specify it; Turbo Debugger doesn't determine extensions.

- If a function name is the first item in a scope override statement, it must not have a # in front of it. If there's a #, Turbo Debugger interprets the function name as a module name.

- Any variable you access through scope override syntax must be initialized. Although an automatic variable doesn't have to be in scope, it must be located on the stack and in the currently loaded executable module.

- If you're trying to access an automatic variable that's no longer in scope, you must use its function name as part of the scope override statement.

- You can't use scope override syntax to access the value of a register variable because once the scope changes, the register no longer holds the value of the variable.

- The scope of a template depends on the current location in the program. The value of a template expression depends on the object that is currently instantiated.

Usually, you'll enter expressions that can be evaluated from the current scope. However, scope overrides are useful when you want to specifically reference a program symbol. For example, you could set up two watches for the variable *nlines*. By setting the watches at different program locations, you can monitor how the variable changes value. The following expressions could be used to set watches on *nlines* for both lines 51 and 72:

```
#51#nlines
#72#nlines
```

Scope override examples using C

Here are some examples of C and C++ expressions that use scope overrides:

`#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#file1.cpp#123`	Line 123 in source file *file1.cpp*, which is part of the object module *mymodule*.
`#mymodule#file1.cpp#123#myvar1`	Symbol *myvar1* accessible from line 123 in source file *file1.cpp*, which is part of *mymodule*.
`#myvar2`	Symbol *myvar* in the current scope.
`#mymodule#myfunc#myvar2`	Symbol *myvar2* accessible from routine *myfunc* in module *mymodule*.
`#mymodule#file2.c#myvar2`	Symbol *myvar2* accessible from *file2.c*, which is defined in *mymodule*.
`AnObject#AMemberVar`	Data member *AMemberVar* accessible in object *AnObject* accessible in the current scope.
`AnObject#AMemberF`	Member function *AMemberF* accessible in object *AnObject* accessible in the current scope.

`#AModule#AnObject#AMemberVar`	Data member *AMemberVar* accessible in object *AnObject* accessible in module *AModule*.
`#AModule#AnObject#AClass::AMemberVar`	Data member *AMemberVar* of class *AClass* accessible in object *AnObject* accessible in module *AModule*.

Note   To examine or call an overloaded member function, enter the name of the function in the appropriate input box. Turbo Debugger opens the Pick a Symbol Name dialog box, which shows a list box of all the functions of that name with their arguments, enabling you to choose the specific function you want.

Overriding scope in Pascal programs

You can use either of the following two types of scope overriding syntax with the Pascal expression evaluator (items enclosed in brackets ([]) are optional):

```
[unit.][procedurename.]symbolname
[unit.][objecttype.|objectinstance.][method.]fieldname
```

The following additional rules apply to the Pascal scope override syntax:

- If you don't specify a unit, the current unit is assumed.

- If you're trying to access a local variable that's no longer in scope, you must use its procedure or function name as part of the scope override statement.

- You can't use a line number or a file name as part of a Pascal scope override statement. If you want to use line number syntax, change the expression evaluator to C with the Options | Language command.

Scope override examples using Pascal

Here are some examples of Pascal expressions that use scope overrides:

`MyVar`	Variable *MyVar* in the current scope.
`MyProc.MyVar`	Variable *MyVar* accessible from the routine *MyProc*.
`MyUnit.MyVar`	Variable *MyVar* accessible from the unit *MyUnit*.
`MyUnit.MyProc.MyVar`	Variable *MyVar* accessible from procedure *MyProc* in unit *MyUnit*.
`AnInstance`	Instance *AnInstance* accessible in the current scope.
`AnInstance.AField`	Field *AField* accessible in instance *AnInstance* accessible in the current scope.
`AnObjectType.AMethod`	Method *AMethod* accessible in object type *AnObjectType* accessible in the current scope.
`AnInstance.AMethod`	Method *AMethod* accessible in instance *AnInstance* accessible in the current scope.

`AUnit.AnInstance.AField`	Field *AField* accessible in instance *AnInstance* accessible in the unit *AUnit*.
`AUnit.AnObject.AMethod`	Method *AMethod* accessible in object type *AnObjectType* accessible in unit *AUnit*.

Scope and DLLs

When you step into a function that's located in a .DLL, Turbo Debugger loads the symbol table for the .DLL, if it exists, over the currently loaded symbol table. Because a DLL's symbol table will be overwritten when your program makes a call to another executable file, you won't have immediate access to variables that are located in an executable file that isn't currently loaded.

If a variable has the same name in multiple .EXE or .DLL files, you can access the desired symbol by loading the executable file in which the symbol is located (press *F3*, and use the Load Modules and DLLs dialog box to load the executable file containing the symbol). For more information on symbol tables and .DLL files, see page 136.

Note   When debugging, all .EXE and .DLL files must be located in the same directory.

Examining disk files
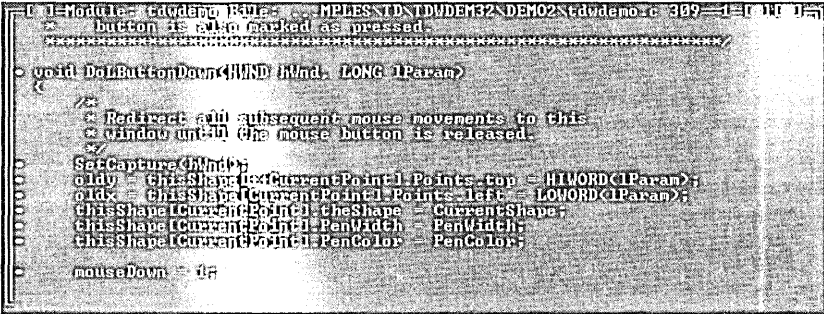
Turbo Debugger provides two ways to view source files, data files, and other files that you have stored on disk:

* The Module window displays the source code relating to executable modules that were compiled with debug information.
* The File window lets you view any disk file as either ASCII text or as hexadecimal data.

Examining program source files

The Module window is the most frequently used window in Turbo Debugger. You can use this window to examine the executable source code of any module that was compiled and linked with debug information.

Figure 8.1   The Module window

When you open the Module window, the title bar displays the name of the currently loaded module, the name of the current source file, and the cursor's line number.

In the Module window, executable lines of code are marked with a bullet (•) in the left column of the window. You can set breakpoints or step to any of these lines of code. An arrow (▸) in the first column of the window indicates the location of the instruction pointer. This always points to the next statement to be executed.

As you step through your program, the Module window automatically displays the source code relating to the current location of the instruction pointer. By navigating to different source-code locations, you can set breakpoints and watches, and inspect the values of different program variables.

If the abbreviation opt appears after the file name in the title bar, the program has been optimized by the compiler. If you compiled your program with optimizations, you might have trouble finding variables that have been optimized away. In addition, compiler optimizations can place variables in registers, meaning that they cannot be linked to memory addresses. Because of this, it is recommended that you do not optimize your program while you are in the debugging stage.

If the word modified appears after the file name in the title bar, the file has been changed since it was last compiled. In this case, the line numbers in the source file might not correspond to the line numbers in the executable's debug information. If these line numbers don't match, the debugger will not be able to show the correct program locations when you step through your code. To correct this problem, recompile your program with symbol debug information.

Loading source files

When you load a program into Turbo Debugger, the file containing the entry point to the program automatically loads into the Module view.

If you want to change the source file that's currently displayed in the Module window, choose one of the following two commands from the Module window SpeedMenu:

- The File command lets you change to another source file contained in the current program module.
- The Module command lets you change the currently loaded program module.

The Module window SpeedMenu

The Module window SpeedMenu provides commands that let you navigate through the displayed file, inspect and watch data items, and load new source code files. The SpeedMenu in TD32 also has the Thread and Edit commands:

```
Inspect
Watch

Thread
Module...
File...

Previous
Line...
Search...
Next
Origin
Goto...
Edit
Exceptions...
```

Inspect
The Inspect command opens an Inspector window that shows the details of the program variable at the current cursor position. If the cursor isn't on a variable, you're prompted to enter an expression to inspect.

You can also use the arrow keys or your mouse to quickly select an expression or string of text in the Module window. To use the keyboard, press *Ins*, and use the left or right arrow keys to mark your selection. To use the mouse, click and drag the mouse pointer over the section of text you want to select. After selecting an expression, press *Ctrl+I* to activate the Inspector window.

Watch
Watch adds the variable at the current cursor position to the Watches window. Putting a variable in the Watches window lets you monitor the value of that variable as your program executes.

If you have selected an expression in the Module window, press *Ctrl+W* to add the expression to the Watches window.

Thread
The Thread command, found only in TD32, opens the Pick a Thread dialog box, from which you can pick a specific program thread to monitor. For more information on threads, see page 139.

Module
The Module command lets you load a different module into the debugger by picking the module you want from the Load Module Source or DLL Symbols dialog box.

The Load Module Source or DLL Symbols dialog box is fully described on page 136.

File

File lets you examine another source file that's compiled into the module you're currently viewing. This command opens the Pick a Source File dialog box, which lists the source files that contain executable code. When you choose the source file you want to examine, that file replaces the current file in the Module window.

To view different files simultaneously, use the View | Another | Module command to open multiple Module windows.

Files that are included in your program with the **#include** directive are also program source files. If an include file contains executable lines of code, you can use the File command to load the file into the Module window. However, if the include file doesn't contain executable code (such as many C header files), you must use the File window to examine the file.

Previous

The Previous command returns you to the source location you were viewing before you changed your position. For example, if you use the Goto command to view the source code at a different address, the Previous command returns you to your original position.

Line

Line positions you at a new line number in the file. The Enter New Line Number dialog box prompts you for a decimal line number. If you enter a line number after the last line in the file, you will be positioned at the end of the file.

Search

The Search command searches for a character string, starting at the current cursor position. When you choose this command, the Enter Search String dialog box prompts you for a search string. If the cursor is positioned over something that looks like a variable name, the dialog box opens initialized to that name.

If you mark a block in the file using *Ins* and the arrow keys, that block will be used to initialize the Search String dialog box.

You can also search using simple wildcards: a question mark (?) indicates a match on any single character and an asterisk (*) matches zero or more characters.

The search does not wrap around from the end of the file to the beginning. To search the entire file, first go to the beginning of the file by pressing *Ctrl+PgUp*.

Next

Next searches for the next instance of the character string you specified with the Search command.

Origin

The Origin command positions the cursor at the module and line number containing the current instruction pointer. If the module you are currently viewing is not the module that contains the instruction pointer, the Module window will change to show that module.

This command is useful when you have been examining various places in your code, and you want to return to the location of the instruction pointer.

Goto

Goto opens the Enter Address to Position To dialog box, which enables you to view any address location within your program. Enter the address you want to examine as either a procedure name or a hexadecimal address. If the address you enter doesn't have a corresponding source line, the CPU window opens. See "Types of expressions" on page 98 for a description of entering addresses.

Note   You can also invoke this command by typing into the Module window. This brings up the Enter Address to Position To dialog box, exactly as if you had chosen the Goto command.

Edit

When you're debugging a Win32s program with TD32, you can invoke the editor of your choice using the Edit command. This command is useful if you've found the program bug, and you want to fix the source code before leaving Turbo Debugger.

Before you can use this command, you must configure TD32 so it knows where to find your editor:

1   Load the TDINST32.EXE installation program.

2   Choose Options | Directories to access the Directories dialog box.

3   Enter the absolute path and name of your editor into the Editor Program Name input field.

4   Save the settings.
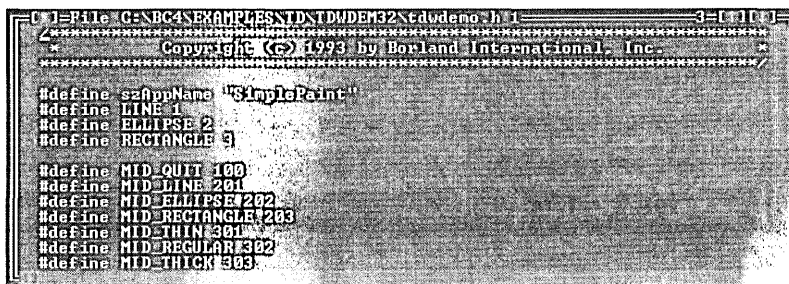
Exceptions

If you have implemented C or C++ exception handling in your program, the Exception command becomes active. For complete details on this command, see page 152.

Examining other disk files

You can use the File window to examine any disk file, including binary and text files.

Figure 8.2   The File window

When you choose View | File from the menu bar, Turbo Debugger displays the Enter Name of File to View dialog box. You can type a specific file name to load, or you can enter a file mask using wildcards characters to get a list of files to choose from.

After you select a file name, the File window opens and displays the file name and contents.

Figure 8.3   The File window showing hex data
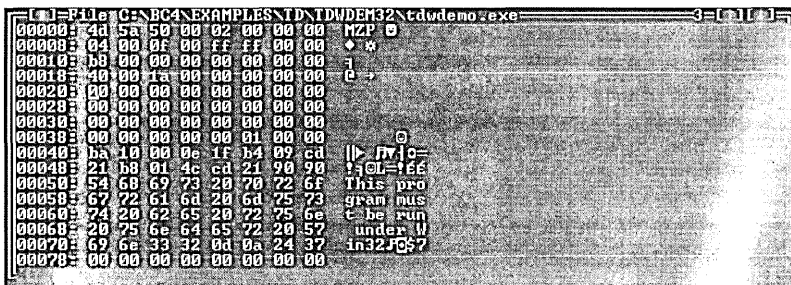
The File window displays files as either ASCII text or as hexadecimal bytes, depending on the contents of the file. If Turbo Debugger determines that the file contains text, it displays the file as ASCII; otherwise, the file is displayed as hexadecimal. You can switch between an ASCII or hexadecimal display using the Display As SpeedMenu command. If you're viewing the file as ASCII, the current line number is also displayed in the title bar.

The File window SpeedMenu

The File window SpeedMenu has commands for navigating through a disk file and for changing the file's display format.

```
Goto...
Search...
Next

Display as    Ascii
File...
```
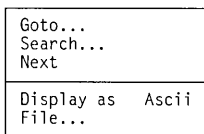
Goto

The Goto command positions the display at a new line number or offset in the file. If you are viewing the file as ASCII text, enter the new line number to go to. If you are viewing the file as hexadecimal bytes, enter the offset that you want to move to. If you enter a line number greater than the last line in the file (or an offset beyond the end of the file), Turbo Debugger displays the end of the file.

Search

The Search command searches for a character string, starting at the current cursor position. When you choose this command, the Enter Search String dialog box prompts you for a search string. If the cursor is positioned over something that looks like a variable name, the dialog box opens initialized to that name.

If you mark a block in the file using *Ins* and the arrow keys, that block will be used to initialize the Search String dialog box.

The search does not wrap around from the end of the file to the beginning. To search the entire file, first go to the beginning of the file by pressing *Ctrl+PgUp*.

If the file is displayed in ASCII, you can use DOS wildcards in your search string: a question mark (?) indicates a match on any single character and an asterisk (*) matches zero or more characters.

If the file is displayed as hexadecimal bytes, enter a byte list consisting of a series of byte values or quoted character strings, using the syntax of the selected expression evaluator. For example, if the language is C++, a byte list consisting of the hex numbers 0408 would be entered as 0x0804. If the language is Pascal, the same byte list is entered as $0804.

You can also invoke this command by typing the string that you want to search for. This brings up the Search dialog box exactly as if you had specified the Search command.

Next

The Next command searches for the next instance of the character string you specified with the Search command.

Display As

Display As toggles the display between the following two formats:

- ASCII displays the file using the printable ASCII character set.

- Hex displays the file in hexadecimal format. With this display, each line starts with the offset from the beginning of the file (shown as a hexadecimal number), followed by the hexadecimal representation of the bytes in the file.The ASCII character for each byte in the file appears on the right side of the display. The File window displays the entire 256 IBM extended-character set.

File

The File command lets you change the file that's displayed in the File window. This command lets you view different files without opening duplicate File windows. If you want to view two different files (or two parts of the same file) simultaneously, choose View | Another | File to open another File window.

Edit

The Edit command is the same as the Module window SpeedMenu Edit command. For more information, refer to page 111.

Assembly-level debugging

When you're debugging a program, the high-level view of your source code is often all you need. Sometimes, however, you might need to take a closer look at your program. Viewing the assembly-level aspects of your program can reveal details such as the machine code generated by your compiler, the contents of the CPU registers and flags, and the items contained on the call stack.

Turbo Debugger provides the following windows for examining the assembly-level state of your program:

- The CPU window
- The Dump window
- The Registers window
- The Numeric Processor window

This chapter describes how to use these windows to view the assembly-level aspects of your program. The online file TD_ASM.TXT contains additional information on assembly-level debugging, including a section describing the Numeric Processor window.

The CPU window

The CPU window uses various panes to describe the low-level state of your program. A SpeedMenu in each pane provides commands specific to the contents of that pane.

Among other things, you can use the CPU window to:

- Examine the machine code and disassembled assembly instructions produced from your program's source code.
- Examine and modify the bytes that make up your program's data structures.
- Use the built-in assembler in the Code pane to test bug fixes.

The CPU window is shown in Figure 9.1. Table 9.1 gives brief a description of each pane in the CPU window.

Figure 9.1   The CPU window

Code pane   Registers pane   Flags pane

Dump pane   Stack pane
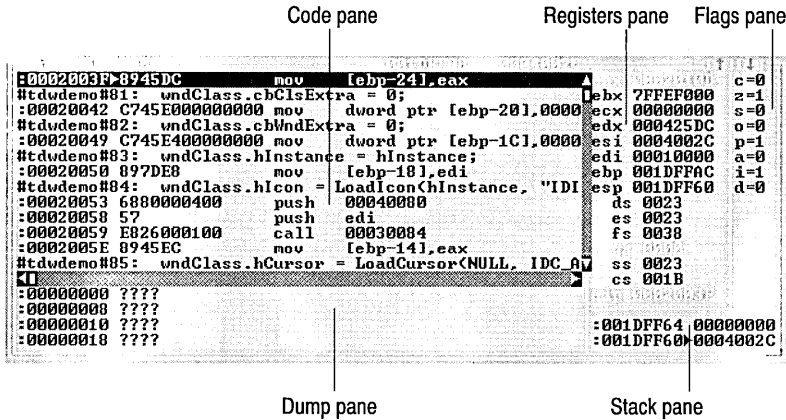
Table 9.1   CPU window panes

Pane	Description
Code pane	Shows the machine code and disassembled assembly instructions of your executable program. Source code lines can also be displayed.
Registers pane	Shows the contents of the CPU registers.
Flags pane	Shows the state of the eight CPU flags.
Dump pane	Shows a hexadecimal dump of any memory area accessible by your program. A variety of display formats is available.
Stack pane	Shows the hexadecimal contents of the program stack.
Selector pane	Available in TDW only, this pane shows and describes all Windows 3.x selectors.

From within the Code, Dump, and Stack pane, it's possible to scroll outside the current protected-mode segment, even though the operating system marks these as invalid addresses for your program. Because of this, the CPU window displays question marks for any addresses referenced outside the current-mode segment.

Note   In the Code, Dump, and Stack panes, press *Ctrl+Left* and *Ctrl+Right* to shift the starting display address of the pane by 1 byte up or down. Using these keystrokes is often faster than using the Goto command to make small adjustments to the display.

Opening the CPU window

To open the CPU window, choose View | CPU from the menu bar. Turbo Debugger opens the CPU window automatically in the following cases:

* If it gains control when Windows code is being executed.
* If you enter a module that doesn't contain debug information.
* If your program stops on an instruction within a line of source code.
* If you trace through instructions using *Alt+F7*.

When you open the CPU window, Turbo Debugger positions the display at the appropriate Code, Dump, or Stack pane, depending on the window that was active

when you opened the CPU window. The following table describes where the cursor is positioned when you open the CPU window:

Table 9.2   CPU window positioning

Current window	CPU pane	Position
Module window	Code	Address of item
Breakpoint (nonglobal)	Code	Breakpoint address
Variable window	Dump/Code	Address of item
Watches window	Dump/Code	Address of item
Inspector window	Dump/Code	Address of item
Stack window	Stack	Top of stack frame for highlighted item
Other area	Code	Current instruction pointer location

Once opened, the title bar of the CPU window displays your system's processor type (8086, 80286, 80386, or 80486). In addition, if the highlighted instruction in the Code pane references a memory location, the memory address and its current contents are displayed in the title bar of the CPU window. This lets you see both where an instruction operand points in memory and the value that is about to be accessed.

The Code pane

The left side of the Code pane lists the address of each disassembled instruction. If you're viewing 16-bit code, the addresses are shown in segment:offset notation. Otherwise, addresses are displayed as 32-bit addresses. An arrow (▸) to the right of the memory address indicates the location of the current *instruction pointer*. The instruction pointer always points to the next instruction to be executed. To the right of this, the CPU window displays the hexadecimal machine code, followed by its disassembled assembly instruction.

When an assembly instruction contains an immediate operand, you can infer its size from the number of digits in the operand: a 1-byte immediate has two digits, a 16-bit immediate has four digits, and a 32-bit immediate has eight digits.

Displaying source code

If you set the Mixed SpeedMenu command to *Yes*, the Code pane displays the source code that relates to the displayed assembly instructions. If an address corresponds to either a global symbol, static symbol, or line number, the CPU window displays the original source code above the first disassembled instruction relating to the source code. 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 cross hatch (#), followed by the static symbol name. Line numbers appear as the module name, followed by a cross hatch (#), followed by the decimal line number.

Setting breakpoints

You can set or remove breakpoints in the Code pane by highlighting the desired assembly instruction, and pressing *F2*. Also, clicking a line sets and removes breakpoints on that line. Once a breakpoint is set, the line containing the breakpoint turns red (the default color).

The Code pane SpeedMenu

The SpeedMenu contains commands that let you navigate through the Code pane, alter the pane's display, and assemble instructions that you supply.

For the most part, the SpeedMenus for TDW and TD32 contain the same commands. However, TDW has the extra command I/O, and TD32 contains the extra commands Thread and OS Exceptions.

Goto

When you choose the Goto command, the Enter Address to Position To dialog box prompts you for an address to go to. You can examine any address that your program can access, including addresses in the ROM BIOS, inside DOS, and in the Windows program.

Origin

The Origin command positions you at the location of the instruction pointer. This command is useful when you have navigated through the Code pane, and you want to return to the next instruction to be executed.

Follow

The Follow command positions the Code pane at the destination address of the currently highlighted instruction. Use this command in conjunction with instructions that cause a transfer of control (such as **CALL, JMP, INT**), and with conditional jump instructions (**JZ, JNE, LOOP**, and so forth). For conditional jumps, the address is shown as if the jump had occurred. Use the Previous command to return to the origin of the jump.

Caller

Caller positions you at the instruction that called the current interrupt or subroutine. Be aware that if the current interrupt routine has pushed data items onto the stack, Turbo Debugger might not be able to determine where the routine was called from.

Previous

The Previous command restores the Code pane display to the position it had before the last command that explicitly changed the display (such as Previous, Caller, Origin, and Follow). The keys do not affect this command.

Search

The Search command searches forward in the code for an expression or byte list that you supply (see Chapter 7 for information on byte lists).

Note   When you search for an expression in the Code pane, Turbo Debugger assembles the expression that you're searching for, and searches for a match in the resulting machine code. Because of this, care must be taken when you specify the search expression; you should search only for expressions that don't change the bytes they assemble to. For example, you will not encounter problems if you search for the following expressions:

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

However, searching for these instructions can cause unpredictable results:

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

View Source
The View Source command activates the Module window, showing you the source code that corresponds to the current disassembled instruction. If there is no corresponding source code (for example, if you're examining Windows kernel code), this command has no effect.

Mixed
Mixed toggles between the three ways of displaying disassembled instructions and related source code:

Table 9.3   Mixed command options

Command	Description
No	Disassembled instructions are displayed without source code.
Yes	Source code lines are listed before the first disassembled instruction relating to that source line. This is the default mode for C and Pascal programs.
Both	Source code lines replace disassembled lines for the lines that have corresponding source code. If there is no source code, the disassembled instruction appears. This is the default mode for assembly modules.
	Use this mode when you're debugging an assembler module and you want to see the original source code instead of the corresponding disassembled instructions.

Thread
The Thread command, found only in TD32, lets you choose the thread of execution you want to debug. When selected, this command opens the Pick a Thread dialog box, from which you can pick a specific program thread. For more information on threads, see page 139.

OS Exceptions
The OS Exceptions command, found only in TD32, lets you choose the operating-system exceptions you want to handle. For more information on operating-system exceptions, see page 142.

New EIP

The New EIP command changes the location of the instruction pointer to the currently highlighted line in the Code pane (in TDW, this command is called New CS:IP). When you resume program execution, execution starts at this address. This command is useful when you want to skip certain machine instructions.

Use this command with extreme care; it is easy to place your system in an unstable state when you skip over program instructions.

Assemble

The Assemble command assembles an instruction, replacing the instruction at the currently highlighted location. Use this command when you want to test bug fixes by making minor changes to assembly instructions.

When you choose Assemble, the Enter Instruction to Assemble dialog box opens, prompting you for an expression to assemble. For more information on assembling instructions, refer to "The Assembler" section in the online file TD_ASM.TXT.

This command is invoked if you type into the Code pane.

I/O

The I/O command, found only in TDW, reads or writes a value in the CPU's I/O space, and lets you examine and write to the contents of special I/O registers. This command gives you access to the I/O space of peripheral device controllers such as serial cards, disk controllers, and video adapters.

When you choose this command, a menu opens with the following commands:

Table 9.4   I/O commands

Command	Description
In Byte	Reads a byte from an I/O port. You are prompted for the I/O port whose value you want to examine.
Out Byte	Writes a byte to an I/O port. You are prompted for the I/O port to write to and the value you want to write.
Read Word	Reads a word from an I/O port.
Write Word	Writes a word to an I/O port.
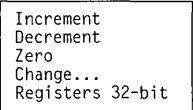
Some I/O devices perform an action (such as resetting a status bit or loading a new data byte into the port) when their ports are read. Because of this, you might disrupt the normal operation of the device with the use of these commands.

The Registers pane

The Registers pane displays the contents of the CPU registers. The display varies, depending on whether you're using TDW or TD32. By default, TDW displays the thirteen 16-bit registers. TD32 always displays the fifteen registers found in the 80386 (and higher) processors.

The Registers pane SpeedMenu

Using the commands on the Register pane SpeedMenu, you can modify and clear the register values.

```
Increment
Decrement
Zero
Change...
Registers 32-bit
```

Increment

Increment adds 1 to the value in the currently highlighted register. This lets you test "off-by-one" bugs by making small adjustments to the register values.

Decrement

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

Zero

The Zero command sets the value of the currently highlighted register to 0.

Change

Change lets you change the value of the currently highlighted register. When you chose this command, the Enter New Value dialog box prompts you for a new value. You can make full use of the expression evaluator to enter new values.

You can also invoke this command by typing the new register value into the Registers pane.

Registers 32-bit

The Registers 32-bit command, used only by TDW, toggles the register display between 16-bit values and (on systems with 32-bit processors) 32-bit values.

TDW usually displays 16-bit registers, unless you use this command to set the display to 32-bit registers. Toggle this command to *Yes* if you're debugging a module that uses 32-bit addressing. Notice that all segment registers will remain as 16-bit values, even when you toggle on the 32-bit display.

The Flags pane

The Flags pane shows the state of the eight CPU flags. The following table lists the different flags and how they are shown in the Flags pane:

Table 9.5   The CPU Flags

Letter in pane	Flag name
c	Carry
z	Zero
s	Sign
o	Overflow

Table 9.5 The CPU Flags (continued)

Letter in pane	Flag name
p	Parity
a	Auxiliary carry
i	Interrupt enable
d	Direction

The Flags pane SpeedMenu

The Flags pane contains the Toggle command, which changes the value of the currently highlighted flag between 0 and 1. You can also press *Enter* or the Spacebar to toggle the value of a flag.

The Dump pane

This pane shows a raw hexadecimal display of an area in memory. The leftmost part of each line shows the starting address of that line, using either 16-bit segment:offset notation or 32-bit flat addresses. With 16-bit code, the address is displayed as either a hex segment and offset, or with the segment value replaced with one of the register names if the segment value is the same as that register. The Dump pane matches registers in the following order: DS, ES, SS, CS.

To the right of the address, the value of one or more data items is displayed. The format of this area depends on the display format selected with the Display As SpeedMenu command. If you choose one of the floating-point display formats (Comp, Float, Real, Double, or Extended), a single floating-point number is displayed on each line. Byte format displays 8 bytes per line, Word format displays 4 words per line, and Long format displays 2 long words per line.

When the data is displayed as bytes, the rightmost part of each line shows the ASCII characters that correspond to the data byte values. Turbo Debugger displays all byte values as their display equivalents, including "nonprintable" characters and the characters from the IBM extended-character set.

If you use the Goto command in the Dump 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 of the program being debugged, not the actual values that are in memory while Turbo Debugger is running. Turbo Debugger detects when you're accessing areas of memory that it is using, and displays the correct program values from where it stores them in memory.

The Dump pane SpeedMenu

The Dump pane SpeedMenu contains commands that let you navigate through the pane, modify memory contents, follow near or far pointers, format the display, and manipulate blocks of memory.

Goto

Goto prompts you for a new area of memory to display with the Enter Address to Position To dialog box. Enter any expression that evaluates to a memory location that your program can access.

Search

The Search command searches for a character string or byte list, starting from the memory address indicated by the cursor.

Next

Next searches for the next instance of the item you previously specified in the Search command.

Change

The Change command lets you modify the bytes located at the current cursor location. If the display is ASCII or if the hexadecimal format is Byte, you're prompted for a byte list. Otherwise, you're prompted for an item of the current display type.

You can invoke this command by typing into the Dump pane.

Follow

The Follow command opens a menu containing commands that let you examine the data at near and far pointer addresses. The TD32 menu contains only the commands that relate to 32-bit addressing.

Table 9.6   Follow command options

Command	Description
Near Code	Interprets the word under the cursor in the Dump pane as an offset into the segment specified by the CS register. This command activates the Code pane, and positions it to the near address.
Far Code	Interprets the doubleword under the cursor in the Dump pane as a far address (segment:offset). This command activates the Code pane, and positions it to the far address.
Offset to Data	Lets you follow word-pointer chains (near and offset only). The Dump pane is set to the offset specified by the word at the current cursor location.
Segment:Offset to Data	Lets you follow long pointer chains (far, segment, and offset). The Dump pane is set to the offset specified by the two words at the current cursor location.
Base Segment: to Data	Interprets the word under the cursor as a segment address and positions the Dump pane to the start of that segment.

Previous

Previous restores the Dump pane position to the address before the last command that explicitly changed the display address. The arrow keys do not affect this command.

Turbo Debugger maintains a stack of the last five addresses accessed in the Dump pane, so you can backtrack through multiple uses of the Follow menu or Goto commands.

Display As

Use the Display As command to format the data that's listed in the Dump pane. You can choose any of the following data formats:

Table 9.7   Display As command options

Command	Description
Byte	Hexadecimal bytes.
Word	2-byte hexadecimal numbers.
Long	4-byte hexadecimal numbers.
Comp	8-byte decimal integers.
Float	4-byte floating-point numbers in scientific notation.
Real	6-byte floating-point numbers in scientific notation.
Double	8-byte floating-point numbers in scientific notation.
Extended	10-byte floating-point numbers in scientific notation.

Block

This command brings up a menu that lets you move, clear, and set blocks of memory. In addition, you can read and write memory blocks to and from files. Use *Ins* and the arrow keys to quickly select the block of bytes that you want to work with.

Table 9.8   Block command options

Command	Description
Clear	Sets a contiguous block of memory to zero (0). You are prompted for the address and the number of bytes to clear.
Move	Copies a block of memory from one address to another. You are prompted for the source address, the destination address, and how many bytes to copy.
Set	Sets a contiguous block of memory to a specific byte value. You are prompted for the address of the block, how many bytes to set, and the value to set them to.
Read	Reads all or a portion of a file into a block of memory. You are prompted for the file name to read from, for the address to read it into, and for how many bytes to read.
Write	Writes a block of memory to a file. You are prompted for the file name to write to, for the address of the block to write, and for how many bytes to write.

The Stack pane

The Stack pane shows the hexadecimal contents of the program stack. An arrow (▸) shows the location of the current stack pointer.

Although you might need to review the hexadecimal bytes that make up the program stack, Turbo Debugger uses the Stack window to show the contents of the stack in a more readable format. See page 92 for a discussion on the Stack window.

The Stack pane SpeedMenu

The SpeedMenu of the Stack pane contains the following commands:

```
Goto
Origin
Follow
Previous
Change...
```

Goto

Goto prompts you for an address to view with the Enter Address to Position To dialog box. If you want, you can enter addresses outside your program's stack, although it's usually easier to use the Dump pane to examine arbitrary memory locations.

Origin

Origin positions you at the current stack location as indicated by the SS:SP register pair.

Follow

The Follow command positions you at the location in the stack pointed to by the currently highlighted word. This is useful for following stack-frame threads back to the calling procedure.

Previous

The Previous command restores the Stack pane position to the address before the last command that explicitly changed the display address (such as Goto, Origin, and Follow). The arrow keys do not affect this command.

Change

Change lets you enter a new word value for the currently highlighted stack word with the Enter New Value for Unsigned Int dialog box.

You can invoke this command by typing the new value for the highlighted stack item.

The Selector pane

The Selector pane, found only in TDW, lists the Windows 3.x protected-mode selectors. A selector can be either valid or invalid. If valid, the selector points to a location in the protected-mode descriptor table corresponding to a memory address. If invalid, the selector is unused.

If a selector is valid, the pane shows the following information:

- The contents of the selector segment (Data or Code).
- The status of the selector memory area: Loaded (present in memory) or Unloaded (swapped out to disk).
- The length of the referenced memory segment in bytes.

If the selector references a data segment, the pane displays additional information on the access rights (Read/Write or Read Only), and the direction in which the segment expands in memory (Up or Down).

The Selector pane SpeedMenu

You use the SpeedMenu of the Selector pane to go to a new selector or see the contents of the currently highlighted selector. Turbo Debugger displays selector contents in either the Code pane or the Dump pane, depending on the nature of the data being displayed.

Selector

The Selector command opens the Enter New Selector dialog box, which prompts you for a selector to display in the pane. You can use full expression syntax to enter the selector. If you enter a numeric value, Turbo Debugger assumes it is decimal, unless you use the syntax of the current language to indicate that the value is hexadecimal.

For example, if the current language were C, you could type the hexadecimal selector value 7F as 0x7F. For Pascal, you'd type it as $7F. You can also type the decimal value 127 to go to selector 7F.

Another method of entering the selector value is to display the CPU window and check the segment register values. If a register holds the selector you're interested in, you can enter the name of the register preceded by an underscore (_). For example, you could type the data segment register as _DS.

Examine

Examine displays the contents of the memory area referenced by the currently highlighted selector. When this command is invoked, either the Code pane or the Dump pane gains focus. If the selector points to a code segment, the contents are displayed in the Code pane. If the selector contents are data, they're displayed in the Dump pane.

The Dump window

The Dump window, opened with the View | Dump command, displays the raw data that's located in any area of memory that can be accessed by your program. The Dump window is identical in behavior to the Dump pane in the CPU window, including all SpeedMenu commands (see page 122 for a description of this pane). The advantage of using the Dump window, however, is that it can be resized.

Figure 9.2   The Dump window
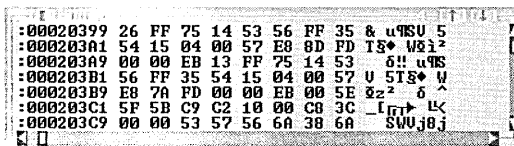
The Dump window is useful when you're in an Inspector window and you want to look at the raw bytes that make up the object you're inspecting. Choosing the View | Dump command when an Inspector window is active opens a Dump window that's positioned at the address of the data in the Inspector window.

You can open several Dump windows simultaneously by choosing View | Another |
Dump.

The Registers window

The Registers window is a combination of the Registers and Flags panes in the CPU
window (see page 120).

Figure 9.3   The Registers window

Registers pane ———
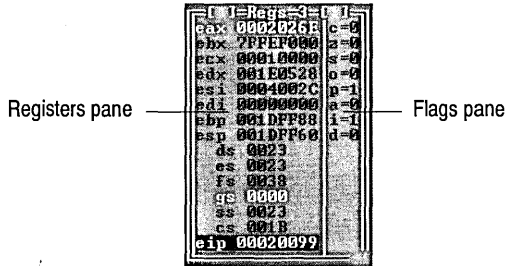
——— Flags pane

You can perform the same functions from the SpeedMenu of the Registers window as
you can from the SpeedMenus of the Registers and the Flags panes in the CPU window.

10

Windows debugging features

Programs written for the Windows operating system can be robust and powerful. However, the added complexity of programming for Windows opens up a new category of software bugs. Turbo Debugger provides the following features to help you find the bugs in your Windows code:

- Windows message tracking and message breakpoints
- Dynamic-link library debugging
- Thread support (for Windows NT only)
- Operating-system exception support for Windows NT and Windows 32s
- Listings of your program's local heap, global heap, and program modules (TDW only)
- Expression typecasting from memory handles to near and far pointers (TDW only)
- A Selector pane in the CPU window of TDW lets you examine any Windows 3.x protected-mode selector (see "The Selector pane" on page 125 for a description of this feature)

Monitoring window messages

The Windows Messages window provides commands for tracking and examining the window messages received by your program. Using this window, you can create *message breakpoints* (breakpoints that pause your program's execution when a specific window message is received), and you can log the messages that a particular window processes.

You open the Windows Messages window, shown in Figure 10.1, with the View | Windows Messages command. Table 10.1 defines the three panes of the Windows Messages window.

Figure 10.1 The Windows Messages window

Window Selection pane Message Class pane

Message Log pane
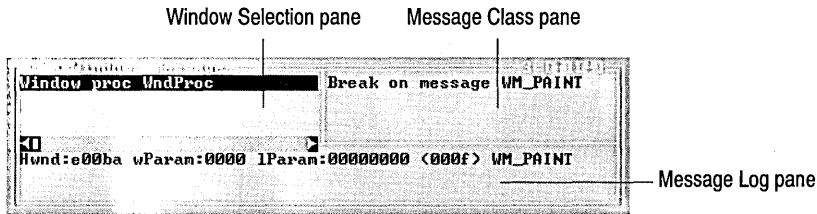
Table 10.1 Windows Messages window panes

Pane	Description
Window Selector pane	Lists the windows that you've selected for messages tracking.
Message Class pane	Lists the messages and message classes that you're tracking for the highlighted window in the Window Selection pane.
Message Log pane	Displays the window messages received by your program.

To track messages for a specific window, follow these steps:

1 Specify a window to monitor.

2 Specify the messages you want to track.

3 Specify the action that Turbo Debugger should take when the window messages are received: Break or Log.

Specifying a window to monitor

The first step in tracking window messages is to specify the window you want to monitor. Although the procedure for specifying windows is similar in both TD32 and TDW, there are some differences.

To specify a window in TD32, use the name of the window procedure that processes the window's messages:

1 Open the Add Window Procedure to Watch dialog box by choosing Add from the Window Selector pane SpeedMenu, or typing directly into the pane.

2 Type the name of the window procedure into the Window Identifier input box, and press *Enter*.

You can repeat this procedure for each window whose messages you want to monitor.

In TDW, you can specify a window by either its window handle or by the window procedure that processes the window's messages. In either case, you use the Add Window or Handle to Watch dialog box to select a window. To access this dialog box, choose Add from the Window Selector pane SpeedMenu, or type directly into the pane.

In TDW's Add Window or Handle to Watch dialog box, the Identify By radio buttons let you choose how you're going to specify the window whose messages you're going to track:

Window Proc   Choose this when you supply the name of the routine that processes the window messages (for example *WndProc*).

Handle   Choose this when you supply the name of the window's handle.

Specifying a window procedure
If you select the Window Proc radio button, enter the name of the window procedure that processes the window's messages in the Window Identifier input box. This is usually the best way to specify a window because you can enter the procedure name any time after you've loaded your program.

Specifying a window handle
If you prefer to use the window's handle name, follow these steps to specify the window's handle:

1 Run your program past the line where the handle is initialized (Turbo Debugger issues an error message if you try to specify a handle name before it's assigned a value).

2 Open the Windows Messages window and choose Add from the Window Selection pane SpeedMenu.

3 Click the Handle radio button.

4 Type the name of the window handle into the Window Identifier input box, and cast the handle to a **UINT** data type.

For example, the following entry would be used to specify the *hWnd* window handle:

```
(UINT)hWnd
```

5 Complete the entry by pressing *Enter*.

Note   If you enter a handle name but click the Window Proc radio button, Turbo Debugger will accept your input, falsely assuming that the "window procedure" will be defined later during your program's execution.

Deleting window selections
The Window Selection pane SpeedMenu contains two menu commands for deleting window selections: Remove and Delete All.

To delete a single window selection, highlight the desired window entry in the Window selection pane, and press *Ctrl+R* (or choose Remove from the pane SpeedMenu). The Delete All command (*Ctrl+D*) erases all window selections, which removes all existing window message tracking.

Specifying the messages to track

After you specify a window in the Window Selector pane, Turbo Debugger, by default, lists all the *WM_* messages sent to that window in the Message Log pane. Because a single window can process many messages, you'll probably want to narrow the focus by selecting the specific messages you're interested in.

To change a window's message-tracking settings, use the Set Message Filter dialog box, which is accessed with the Window Class pane SpeedMenu Add command. (You can also begin typing into the Window Class pane to access the dialog box.) This dialog box lets you select window messages by either message class or by individual message names.

Figure 10.2 The Set Message Filter dialog box
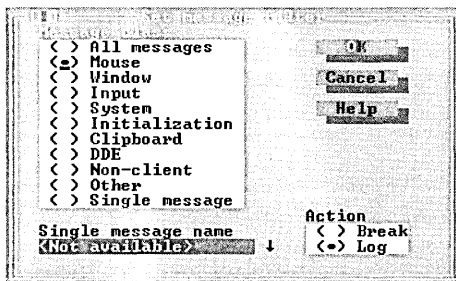
Note   Before you can access the Set Message Filter dialog box, you must first specify a window in the Window Selection pane.

Specifying a message class to track

To track a specific message class for the highlighted window in the Window Selection pane, open the Set Message Filter dialog box and choose one of the following message classes from the Message Class radio buttons.

All Messages	All window messages.
Mouse	Messages generated by a mouse event (for example, WM_LBUTTONDOWN and WM_MOUSEMOVE).
Window	Messages generated by the window manager (for example, WM_PAINT and WM_CREATE).
Input	Messages generated by a keyboard event or by the user accessing a System menu, scroll bar, or size box (for example, WM_KEYDOWN).
System	Messages generated by a system-wide change (for example, WM_FONTCHANGE and WM_SPOOLERSTATUS).
Initialization	Messages generated when an application creates a dialog box or a window (for example, WM_INITDIALOG and WM_INITMENU).

Clipboard	Messages generated when the user accesses the Clipboard (for example, WM_DRAWCLIPBOARD and WM_SIZECLIPBOARD).
DDE	Dynamic Data Exchange messages, generated by applications communicating with one another's windows (for example, WM_DDE_INITIATE and WM_DDE_ACK).
Non-client	Messages generated by Windows to maintain the non-client area of an application window (for example, WM_NCHITTEST and WM_NCCREATE).
Other	Any messages that don't fall into the other message categories, such as owner draw control messages and multiple document interface messages.
Single Message	Lets you specify a single message to track.

To track a single message, choose the Single Message radio button and enter the message name or message number (an integer) into the Single Message Name input box. Message names are case sensitive; be sure to match their names exactly.

Although you can set up a single window to track many different message classes and message names, you can add only one message class or message name at a time. If you want to track more than a single class or message with a particular window,

1 Specify a single message class or message name.

2 Choose Add from the Message Class pane SpeedMenu.

3 Append additional message classes or message names to the window's message-tracking definition.

Specifying the message action

After specifying a window and the messages to track, you must indicate the action that you want to perform when the window messages are received. Turbo Debugger provides the following two Action radio buttons in the Set Message Filter dialog box:

Break	Pause program execution when the window receives one of the specified messages.
Log	List all specified messages in the Message Log pane of the Windows Messages window (default).

Breaking on messages

If you want Turbo Debugger to gain control when a specific window message is received by your program, choose Break as the message action. This setting is known as a *message breakpoint*.

The following example shows how to set a message breakpoint on WM_PAINT, which pauses your program every time the message is sent to the window you've selected in the Window Selection pane:

1 Enter a window procedure name into the Window Selection pane.

2 Activate the Message Class pane (on the top right), and choose Add from its SpeedMenu. This opens the Set Message Filter dialog box.

3 Click Single Message from the Message Class radio buttons, and enter WM_PAINT in the Message Name input box.

4 Click the Break radio button.

5 Press *Enter*.

Figure 10.1 on page 130 shows how the Windows Messages window looks after you have made these selections and a WM_PAINT message has been received.

Logging messages

If you choose the Log radio button, Turbo Debugger lists the specified window messages in the Message Log pane of the Windows Messages window. This pane can list up to 200 messages.

If you're tracking many messages, you might want to write the messages to a file so you don't overwrite the messages already sent to the Message Log pane. To do so,

1 Set the Action radio button to Log.

2 Activate the Message Log pane, and set the Send to Log Window SpeedMenu command to Yes.

3 Open the Log window, using the View | Log command.

4 Choose Open Log File from the Log window SpeedMenu.

For details on logging messages to a file, see page 82.

To clear the Message Log pane, choose Erase Log from its SpeedMenu. Messages already written to the Log window are not affected by this command.

Deleting message class and action settings

To delete a window's message and action settings, highlight the desired item in the Message Class pane and choose Remove from the SpeedMenu. You can also remove window settings by pressing either *Delete* or *Ctrl+R*. To delete all window message and action settings, choose Delete All from the SpeedMenu, or press *Ctrl+D*.

If you delete all message and action settings, the default setting (Log All Messages) is automatically assigned to the window highlighted in the Window Selection pane.

Message tracking tips

The following tips can be helpful when you track window messages:

• If you're tracking messages for more than a single window, don't log all the messages. Instead, log specific messages or specific message classes for each window. If you log all messages, the large number of messages being transferred between Windows and Turbo Debugger might cause your system to crash.

- When setting a message breakpoint on the Mouse message class, be aware that a *WM_MOUSEDOWN* message must be followed by a *WM_MOUSEUP* message before the keyboard becomes active again. This restriction means that when you return to the application, you might have to press the mouse button several times to get Windows to receive a *WM_MOUSEUP* message. You'll know that Windows has received the message when you see it displayed in the Message Log pane.

Debugging dynamic-link libraries

A dynamic-link library (DLL) is a library of routines and resources that is linked to your Windows application at run time rather than at compile time. Windows links DLLs at run time to save memory by allowing multiple applications to share a single copy of routines, data, or device drivers. When an application needs to access a DLL, Windows checks to see if the DLL is already loaded into memory. If the DLL is loaded, then there is no need to load a second copy of the file.

DLLs can be loaded into memory by your program at two different times:

- When your program loads (DLLs are loaded at this time if you've statically linked them using the IMPLIB utility)

- When your program issues a *LoadLibrary* call

Stepping into DLL code

When you single-step into a DLL function, Turbo Debugger loads the DLL's symbol, loads the source code of the DLL into the Module window, and positions the cursor on the called routine.

However, before a DLL's source code can be loaded into the Module window, the following conditions must be met:

- The DLL must be compiled with symbolic debug information.

- The .DLL file must be located in the same directory as your program's .EXE file.

- The DLL's source code must be available.

    Turbo Debugger searches for DLL source code the same way it searches for the source code of your program's executable file, as described on page 22.

If a DLL doesn't contain debug information, or if Turbo Debugger can't find the DLL's source code, Turbo Debugger opens the CPU window and displays the DLL's disassembled machine instructions.

Returning from a DLL
If, when debugging a DLL function, you step past the **return** statement with *F7* or *F8*, your program might begin to run as though you had pressed *F9*. This behavior is typical when you're debugging a DLL that was called from a routine that doesn't contain symbolic debug information, or when the DLL function returns through a Windows function call.

If you're debugging DLL startup code, set a breakpoint on the first line of your program before you load the DLL to ensure that your program will pause when you step past the DLL's return statement.

Accessing DLLs and source-code modules

Although Turbo Debugger makes stepping into DLL functions transparent, you might need to access a DLL before your program makes a call to it. For example, you might need to access a DLL to set breakpoints or watches, or to examine a function's source code.

To access an executable module other than the one that's currently loaded, open the Load Module Source or DLL Symbols dialog box by choosing the View | Modules command or by pressing *F3*.

Figure 10.3 The Load Module Source or DLL Symbols dialog box
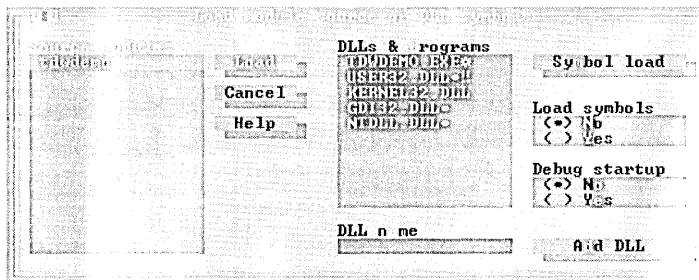
The Source Modules list box displays all the source modules contained in the currently loaded executable file. The DLLs & Programs list box displays all the .DLL and .EXE files that are currently loaded by Windows. (If you're running TDW, the list also displays all loaded .DRV and .FON files.)

A bullet (•) next to a DLL listing indicates that it can be loaded into Turbo Debugger (as long as the DLL contains symbolic debug information and the source code is available). An asterisk (*) next to a module indicates that the module has been successfully loaded by Turbo Debugger.

Note   Because your program might load DLL modules with the *LoadLibrary* call, the DLLs & Programs list box might not display all of the .DLL files your program uses.

Changing source modules
If you need to access a different source code module in the currently loaded executable file, highlight the desired module in the Source Modules list box, and press the Load button (you can also double click the desired module to load it). Turbo Debugger opens the Module window, which displays the selected source code module.

Changing executable files
To access an executable file that's not currently loaded:

1 Open the Load Module Source or DLL Symbols dialog box (press *F3* or choose View | Modules).

2 Highlight the desired file in the DLLs & Programs list box.

3 Choose the Symbol Load button.

Turbo Debugger opens the Module window, which displays the first source code module found in the executable module. If you need to switch source code modules, follow the directions in the preceding section.

Adding DLLs to the DLLs & Programs list

To access a DLL through the Load Module Source or DLL Symbols dialog box, the DLL must be listed in the DLLs & Programs list box. However, if a DLL is loaded with the *LoadLibrary* call, the DLL might not yet be listed (a DLL's name is listed only after it's been loaded).

To add a DLL to the DLLs & Programs list box:

1 Open the Load Module Source or DLL Symbols dialog box (press *F3* or choose View | Modules).

2 Activate the DLL Name input box, and enter the name of the desired DLL (enter the full path if necessary).

3 Press the Add DLL button to add the DLL to the list.

Stepping over DLLs

Whenever you step into a function contained in a DLL, Turbo Debugger automatically loads in the symbol table and source code for that DLL (providing that the source code is available and the DLL was compiled with symbolic debug information). This includes DLLs that your program loads with the *LoadLibrary* call.

Because it takes time to swap symbol tables and source code, you might want to disable the swapping operation for the DLLs you don't need to debug. To prevent Turbo Debugger from loading a DLL's symbol table and source code,

1 Open the Load Module Source or DLL Symbols dialog box (press *F3* or choose View | Modules).

2 Highlight the desired DLL in the DLLs & Programs list box.

3 Choose the No radio button, and choose OK.

   To re-enable the loading of a DLL's symbol table, choose the Yes radio button in the Load Symbols group.

When you disable the loading of a DLL's symbol table, the bullet next to the DLL listing in the DLLs & Programs list box disappears. Although Turbo Debugger will now automatically step over calls to the DLL, you can still access the DLL through the Symbol Load button, as described in the preceding section "Accessing DLLs and source-code modules."

Note When you reload a program, the Load Symbols radio button is set to Yes for all DLLs and modules, even for DLLs or modules that were previously set to No.

Debugging DLL startup code

When your application loads a DLL (when either the program is loaded or when your program makes a *LoadLibrary* call), the DLL's startup code is executed. By default, Turbo Debugger does not step through a DLL's startup code. However, if you need to verify that a DLL is loading correctly, then you'll need to debug the DLL's startup code.

Turbo Debugger lets you debug two types of DLL startup code:

• The initialization code immediately following *LibMain* (default mode).

• The assembly-language code linked to the DLL. This code initializes the startup procedures and contains the emulated math packages for the size model of the DLL. (Select this debug mode by starting Turbo Debugger with the –l command-line option.)

You set DLL startup code debugging with the Load Module Source or DLL Symbols dialog box. However, if you try to run your application after setting the startup debugging, Turbo Debugger might not behave as you expect because some or all of the DLLs might already have been loaded. Because of this, you must load your application, set the startup debugging for selected DLLs, and then restart your application using the Run | Program Reset command (*Ctrl+F2*).

With these preliminaries in mind, follow these steps to specify startup debugging for one or more DLLs:

1 Load your program into Turbo Debugger.

2 Bring up the Load Module Source or DLL Symbols dialog box (press *F3* or choose View | Modules).

3 Highlight the DLL whose startup code you want to debug in the DLLs & Programs list box.

4 Choose the Debug Startup Yes radio button.

If the needed DLL isn't on the list, add it using the method described in the section "Adding DLLs to the DLLs & Programs list" on page 137.

When you specify startup debugging for a DLL, the DLL's entry in the DLLs & Programs list box displays a double exclamation point (!!) next to it.

5 Repeat steps 3 and 4 until you've set startup debugging for all desired DLLs.

6 Choose Run | Program Reset or *Ctrl+F2* to reload your application.

After you've set up startup debugging for DLLs, you're ready to run your program. However, before you begin, keep the following in mind:

• *Be sure to run to the end of a DLL's startup code before reloading the current application or loading a new one.* If you don't, the partially executed DLL startup code might cause Windows to hang, forcing you to reboot.

- Setting breakpoints on the first line of your application, or the first line after a *LoadLibrary* call, guarantees that control returns to Turbo Debugger after the DLL's startup code executes.

- As your application loads each DLL, Turbo Debugger places you in either the Module window at the DLL's *LibMain* function (the default), or in the CPU window at the start of the assembly code for the startup library.

- When you've finished debugging the startup code for a DLL, press *F9* to run through the end of the startup code and return to the application. If you've specified any additional DLLs for startup code debugging, Turbo Debugger displays startup code for them when your application loads them.

Debugging multithreaded programs

The Threads window (opened with the View | Threads command) supports the multithreaded environment of Windows NT.

Figure 10.4 The Threads window

Threads Detail pane   Threads List pane
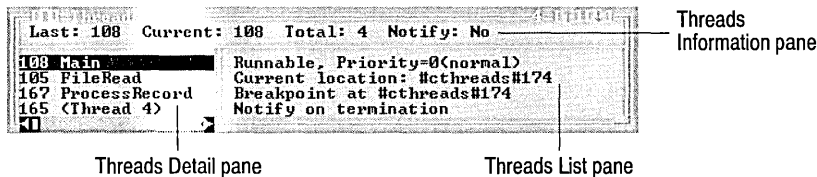
The Threads Information pane

The Threads Information pane, which lists general thread information, consists of these fields:

The *Last field* lists the last thread that was executing before Turbo Debugger regained control.

The *Current field* shows the thread whose values are displayed in Turbo Debugger's windows. You can change the thread you're debugging via the Make Current SpeedMenu command.

The *Total field* indicates the total number of active program threads.

The *Notify field* displays either Yes or No, the Notify on Termination status of all threads. Although you can set the Notify on Termination status for individual threads, the overall status is set through the All Threads SpeedMenu command. Newly created threads are also assigned this status.

The Threads List pane

The Threads List pane lists all your program's active threads. Threads are identified by a thread number (assigned by Windows NT) and a thread name. Turbo Debugger generates a thread name when your program creates a thread. The first thread created is

named Thread 1, followed by Thread 2, and so on. You can modify a thread's name using the Option command on the List pane SpeedMenu.

Threads List pane SpeedMenu

The Threads window contains a single SpeedMenu (which you activate through the Threads List pane) which contains the Options, Make Current, and All Threads commands.

Options

The Options SpeedMenu command opens the Thread Options dialog box. This dialog box lets you set options for individual program threads.

Figure 10.5   The Thread Options dialog box
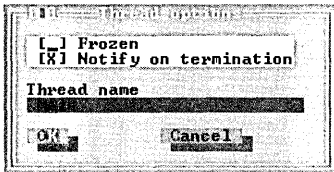
The *Freeze check box* lets you freeze and thaw individual threads. When you freeze a thread by checking this box, the thread will not run. To thaw the thread (which enables it to run), clear the check box. For your program to run, there must be at least one thread that isn't frozen.

Note   If you freeze the only thread in your program that processes window messages, your program and the debugger will hang when you run the program.

The *Notify on Termination check box* lets you specify whether Turbo Debugger should notify you when the currently highlighted thread terminates. When this box is checked, Turbo Debugger generates a message when the thread terminates, and activates a Module or CPU window that displays the current program location. If you clear the Notify on Termination check box, Turbo Debugger doesn't pause when the thread terminates. To set the Notify on Termination status for all threads, use the All Threads SpeedMenu command.

The *Thread Name input box* lets you modify the thread name that's generated by Turbo Debugger. If your program generates many threads, it can be easier to keep track of them if you specify your own thread names.

Make current

The Make Current command lets you change the thread currently being processed by Turbo Debugger. To change the current thread, highlight the thread that you want to examine in the Threads List pane, and press *Ctrl+M* (or choose the Make Current command). When you do so, the Thread Information pane displays the thread number whose data values are displayed in Turbo Debugger's windows, and all references to the CPU registers and stack data will now relate to this thread.

Inspect

The Inspect command opens a Module or CPU window that shows the current point of execution for the highlighted thread. Pressing *Enter* has the same effect as choosing Inspect from the SpeedMenu.

All threads

The All Threads command opens a menu whose commands relate to all program threads.

The *Thaw command* unfreezes any currently frozen threads. When you issue this command, all threads in your program are able to run.

The *Freeze command* disables all thread execution. When you issue this command, all threads in your program will be frozen and unable to run. For your program to run, you must thaw at least one thread using the Options SpeedMenu command (or use the Thaw command on the All Threads menu to unfreeze all the threads).

The *Enable Exit Notification command* sets the notify-on-exit status for all program threads, including threads that have yet to be created. Choosing this command causes Turbo Debugger to issue a message when any thread terminates. The status of notify-on-exit is displayed in the Notify field of the Threads Information pane.

The *Disable Exit Notification command* turns off the notify-on-exit status. This is Turbo Debugger's default setting.

Step

The Step command toggles between *All* and *Single*:

When set to All (the default), all the threads in your program can run as you step through your program using *F7* or *F8*. If you're debugging a thread with a low priority, other threads might execute several statements before the thread you're debugging executes a single statement. (This can sometimes make it difficult to watch the behavior of a single thread in your program.)

When the Step command is set to Single, only the thread located at the current instruction pointer will run as you step. This is different from freezing threads because different threads can be created and destroyed, and you can step into these threads as your program's execution dictates.

The Threads Detail pane

The Thread Detail pane, shown in Figure 10.4, displays the details of the thread that's highlighted in the Threads List pane.

The first line of the Thread Detail pane displays the status of the highlighted thread (either suspended or runnable) and the thread's priority. The priority, which is set by the operating system, can be one of five different states:

–2 (lowest)	1 (above normal)
–1 (below normal)	2 (highest)
0 (normal)	

The second line of the Thread Detail pane displays the current execution point of the thread that's highlighted in the Threads List pane.

The third line, if present, indicates how Turbo Debugger gained control from the running thread. A complete list of the messages that Turbo Debugger can generate for this line is given in the "Status messages" section on page 187.

The fourth line of the Thread Detail pane, if present, lists the thread's settings. Possible settings are Frozen and Notify on Termination.

Tracking operating-system exceptions

In TD32, the OS Exceptions command (located on the SpeedMenu of the CPU window's Code pane) opens the Specify Exception Handling dialog box. This dialog box lets you specify how Turbo Debugger should handle the operating-system exceptions that are generated by your program.

Figure 10.6   The Specify Exception Handling dialog box
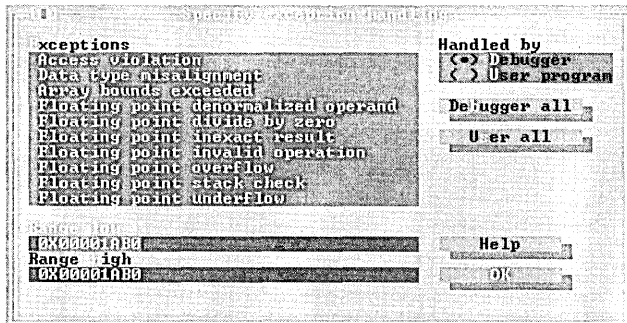
The Exceptions list box displays all the operating-system exceptions that can be handled by Turbo Debugger. For each exception in the list, you can specify whether Turbo Debugger should handle the exception or whether your program's exception-handling routine should take control.

By default, all exceptions generated by the operating system are handled by Turbo Debugger. This means that whenever your program generates an operating-system exception, Turbo Debugger pauses your program and activates the Module or CPU window with the cursor located on the line of code that caused the exception.

To change the debugger's default exception handling behavior,

1   Open the Specify Exception Handling dialog box using the OS Exceptions command on the SpeedMenu of the CPU window's Code pane.

2   Highlight the exception you want your program to handle.

3 Click the User Program radio button.

When you specify that your program will handle an operating-system exception, Turbo Debugger places a bullet (•) next to the exception listing in the Exceptions list box.

If you want your program to handle all the operating-system exceptions, click the User All button on the right side of the Specify Exception Handling dialog box. To have Turbo Debugger pause on all operating-system exceptions, click the Debugger All button (default).

Specifying user-defined exceptions

Turbo Debugger supports user-defined operating-system exceptions with the Range Low and Range High input boxes in the Specify Exception Handling dialog box.

By default, Turbo Debugger sets both the Range Low and Range High input boxes to 0. This default state indicates that there are no user-defined operating-system exceptions.

To have Turbo Debugger monitor a single user-defined operating-system exception, enter the hexadecimal number generated by the exception into the Range Low input box. The following line then appears at the bottom of the Exception list box, where *XXXXXXXX* equals the hexadecimal value of the exception:

```
Range: XXXXXXXX to XXXXXXXX
```

If you've defined more than one operating-system exception, enter the lowest user-defined operating-exception number into the Range Low input box, and the highest user-defined operating-exception number into the Range High input box. The Range listing the Exceptions list box will then indicate the range of user-defined operating-system exceptions that Turbo Debugger will monitor.

Obtaining memory and module lists

In TDW, you can write either the contents of the global heap, the contents of the local heap, or the list of modules used by your program to the Log window. The Windows Information dialog box (accessed by choosing the Display Windows Info command on the Log window SpeedMenu) lets you pick the type of list you want displayed, and where you want the list to start.

Figure 10.7 TDWs Windows Information dialog box

```
┌─[■]════Windows information═══════════
║ Display
║   (•) Global heap        ┌────────┐
║   ( ) Local heap         │   OK   │
║   ( ) Module list        └────────┘
║                          ┌────────┐
║  Start at                │ Cancel │
║   (•) Top                └────────┘
║   ( ) Bottom             ┌────────┐
║   ( ) Handle             │  Help  │
║                          └────────┘
║  Starting handle
║  <Not available>
```
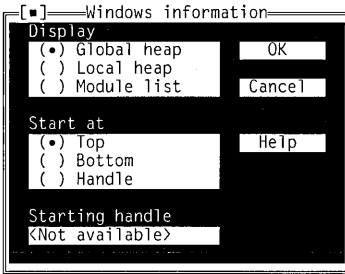
Listing the contents of the global heap

The *global heap* is the global memory Windows makes available to all applications. If you allocate resources like icons, bit maps, dialog boxes, and fonts, or if you allocate memory using the *GlobalAlloc* function, your application uses the global heap.

To see a list of the data objects in the global heap, select the Global Heap radio button in the Windows Information dialog box and choose OK. The data objects in the global heap are then listed in the Log window.

In addition to listing the global heap, the Start At radio buttons let you choose whether to display the list from the top or bottom of the heap, or from a location indicated by a starting handle.

A handle is the name of a global memory handle set in your application by a call to a Windows memory allocation routine like *GlobalAlloc*. Picking a handle causes Turbo Debugger to display the object at that handle and the next four objects that follow it in the heap.

Note Because the global heap listing is likely to exceed the number of lines in the Log window (the default is 50 lines), you should either write the contents to a log file (using the Log window's Open Log File SpeedMenu command) or increase the number of Log window lines (using TDWINST). The Log window can hold a maximum of 200 lines.

The following line shows an example of a global heap listing. Table 10.2 gives an explanation of each field in the output.

```
053E (053D) 00002DC0b PDB (0F1D)  DATA MOVEABLE LOCKED=00001 PGLOCKED=0001
```

Table 10.2 Format of a global heap list

Field	Description
053E	Either a handle to the memory object, expressed as a 4-digit hex value, or the word *FREE*, indicating a free memory block.
(053D)	A memory selector pointing to an entry in the global descriptor table. The selector isn't displayed if it's the same value as the memory handle.
00002DC0b	A hexadecimal number representing the length of the segment in bytes.
PDB	The allocator of the segment, usually an application or library module. A PDB is a process descriptor block; it is also known as a program segment prefix (PSP).
(0F1D)	A handle indicating the owner of a PDB.

Table 10.2 Format of a global heap list (continued)

Field	Description
DATA	The type of memory object. Possible types are: • *DATA* Data segment of an application or DLL. • *CODE* Code segment of an application or DLL. • *PRIV* Either a system object or global data for an application or DLL.
MOVABLE	A memory allocation attribute. An object can be *FIXED, MOVABLE,* or *MOVABLE DISCARDABLE.*
LOCKED=00001	For a movable or movable-discardable object, this is the number of locks on the object that have been set using either the *GlobalLock* or *LockData* functions.
PGLOCKED=0001	For 386 Enhanced mode, the number of page locks on the object that have been set using the *GlobalPageLock* function. With a page lock set on a memory object, Windows can't swap to disk any of the object's 4-kilobyte pages.

Listing the contents of the local heap

The *local heap* is a private memory area used by your program; it is not accessible to other Windows applications, including other instances of the same application.

A program doesn't necessarily have a local heap. Windows creates a local heap only if the application uses the *LocalAlloc* function.

To see a list of the data objects in the local heap, select the Local Heap radio button in the Windows Information dialog box, then choose OK. The local heap data objects will be listed in the Log window.

The following line shows an example local heap listing. Table 10.3 gives an explanation of each field in the output.

```
05CD:  0024  BUSY  (10AF)
```

Table 10.3 Format of a local heap list

Field	Description
05CD:	The object's offset in the local data segment.
0024b	The length of the object in bytes.
BUSY	The disposition of the memory object, as follows: • *FREE* An unallocated block of memory. • *BUSY* An allocated object.
(10AF)	A local memory handle for the object.

Listing the Windows modules

To see a list of the tasks and DLL modules that have been loaded by Windows, select the Module List radio button in the Windows Information dialog box, then choose OK. The modules will be listed in the Log window.

The following line shows an example module listing. Table 10.4 gives an explanation of each field in the output.

```
0EFD TASK GENERIC  C:\TPW\GENERIC.EXE
```

Table 10.4   Format of a Windows module list

Field	Description
0EFD	A handle for the memory segment, expressed as a 4-digit hex value.
TASK	The module type. A module can be either a task or a DLL.
GENERIC	The module name.
C:\TPW\GENERIC.EXE	The path to the module's executable file.

Converting memory handles to addresses

In a Windows program, you reference a data object using a symbolic name instead of using the object's physical address. This way, Windows can perform its own memory management, and can change the physical address of the object without creating conflicts with your program.

Turbo Debugger provides two special data types to help you obtain the physical address of a data object that's referenced by a memory handle: **lh2fp** and **gh2fp**. If you need the actual address referred to by a memory handle, use the typecast symbols **lh2fp** to dereference a local handle and **gh2fp** to dereference a global handle.

You use Turbo Debugger's special data types for typecasting, just as you can use any of C's built-in data types. For example, you could cast the local memory handle *hLocalMemory* using two methods:

• Use the Data | Inspect window to evaluate the expression (lh2fp)hLocalMemory.

• Use the Type Cast command in the Inspector local window and enter lh2fp as the type.

In either case, the expression evaluates to the first character of the memory block pointed to by *hLocalMemory*.

You could also use either of these techniques to do a more complicated cast. For example, a two-stage cast—from a handle into a character pointer into a pointer to the data in memory—could read as follows:

```
(Mystruct far *)(lh2fp)hLocalMemory
```

Debugging object-oriented programs

Turbo Debugger supplies the following features to help you debug object-oriented programs:

- The Hierarchy window
- Class Inspector windows
- Object inspector windows
- C++ and C exception handling

The Hierarchy window

The Hierarchy window, which is opened with the View | Hierarchy command, provides a graphic display of the class hierarchies in your program.

Figure 11.1   The Hierarchy window

Classes pane   Hierarchy pane

```
HRSRC__          TEventHandler::TEventHandler::TEventInfo
HWND__           TResponseTableEntry<GENERIC>
ObjectBuilder    TEventHandler
TApp             |--------TWindow**
TApplication     |              |--------TGameWindow
TApplication::TA |              |--------TDialog*
TBitmap          |              |--------TFrameWindow*
TBrush
TCVectorImp<TStr Parents of TDialog
TClientDC        |--------TEventHandler
TClipboard       |--------TStreamableBase
TColor           |--------TWindow
TCommandEnabler  |--------TEventHandler
```

Parents pane

The Hierarchy window displays the heritage of object-oriented program classes. The window is composed of two or three panes, depending on whether or not your program uses multiple inheritance.

The Classes pane

The Classes pane displays an alphabetical listing of the classes used by the currently loaded module. The class that's highlighted in this pane is detailed in the pane(s) on the window's right side.

The Classes pane uses incremental matching to help you quickly find the class you're interested in. As you type the name of a class into the pane, Turbo Debugger highlights the class whose name matches the keystrokes you've pressed.

The Classes pane SpeedMenu

The Classes pane contains two SpeedMenu commands:

```
Inspect
Tree
```
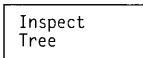
Inspect

The Inspect command opens a Class Inspector window for the currently highlighted class. Alternately, you can press *Enter* to open a Class Inspector window for the highlighted class. For a description of Class Inspector windows, see page 149.

Tree

The Tree command activates the Hierarchy pane, highlighting the currently selected class.

The Hierarchy pane

The Hierarchy pane displays the loaded module's classes and their hierarchies. Original base classes are placed at the left margin of the pane with derived classes displayed beneath their base classes.

Classes that inherit from multiple base classes are marked with asterisks. The first class in a group of multiply-inherited classes is marked with a double-asterisk (**); all other classes that are part of the same multiple-inheritance group are marked with a single asterisk (*).

To locate a class in a complex hierarchy, choose the Tree command from the Classes pane SpeedMenu. This navigates to the class in the Hierarchy pane.

The Hierarchy pane SpeedMenu

The Hierarchy pane SpeedMenu has one or two commands, depending on whether or not your program implements classes with multiple inheritance.

Inspect

When you choose Inspect (or press *Enter*), a Class Inspector window opens for the class that's highlighted in the pane.

Parents

If you're debugging an object-oriented program that implements classes derived through multiple inheritance, the Hierarchy pane SpeedMenu also contains the Parents command. The Parents command toggles on and off the display of the Hierarchy window's Parents pane. The default for Parents is Yes.

The Parents pane

The Hierarchy window's Parents pane appears only if your program contains classes that inherit from multiple base classes, and the Parents command on the Hierarchy pane SpeedMenu is set to Yes.

The Parents pane displays all base classes for the classes that are derived through multiple inheritance. A class' display begins with the message Parents of <ClassName>. Beneath this, the pane displays a reverse hierarchy tree for each set of base classes, with lines indicating the base class and derived class relationships.

The Parent pane SpeedMenu

The Parent pane, if displayed, contains a single SpeedMenu command: Inspect. Choosing Inspect (or pressing *Enter*), opens a Class Inspector window for the class highlighted in the pane.

Class Inspector windows

The Class Inspector window lets you inspect the details of object-oriented program classes. To open a Class Inspector window, activate the Hierarchy window (choose View | Hierarchy), highlight a class, and press *Enter*.

Figure 11.2   A Class Inspector window
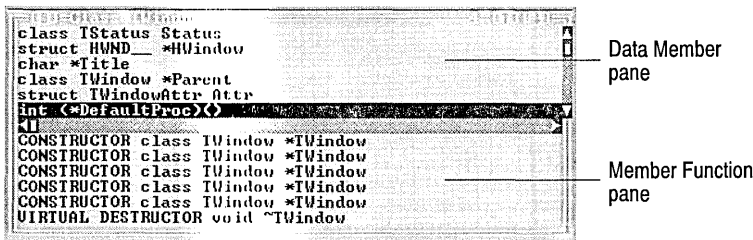
A Class Inspector window is divided horizontally into two panes. The top pane lists the class' data members and type information, and the bottom pane lists the class' member functions and their return types.

A Class Inspector window summarizes the data members and member functions contained in a C++ class; it doesn't, however, reflect the data of any particular instance. If you want to examine a member function's arguments, highlight the member function and press *Enter*. A Function Inspector window opens, displaying the code address for the object's implementation of the function and the names and types of all its arguments.

If the highlighted data member is a pointer to a class, pressing *Enter* opens another Class Inspector window for the highlighted class. (This action is identical to choosing Inspect in the SpeedMenu for this pane.) Using this functionality, you can inspect complex, nested classes with a minimum of keystrokes.

As with all Inspector windows, *Esc* closes the current Inspector window and *Alt+F3* closes them all.

The Class Inspector window SpeedMenus

The SpeedMenus in each pane of the Class Inspector window contain identical commands, although they behave slightly differently in each pane:

```
Inspect
Hierarchy
Show inherited    Yes
```
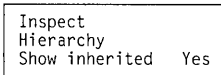
Inspect
The Data Member pane's Inspect command opens an Inspector window on the highlighted data member. If the data member is a pointer to another class, a Class Inspector window opens for that class.

The Member Function pane's Inspect command opens a Function Inspector window on the highlighted member function. To display a member function's source code, position the cursor over the address of the member function in the Function Inspector window, and press *Enter* to activate the Module window.

Hierarchy
The Hierarchy command on each SpeedMenu opens the Hierarchy window, displaying the currently inspected class. The Hierarchy window is described on page 147.

Show Inherited
The Show Inherited command toggles between Yes and No in each pane of the Class Inspector window. The default value in each pane is Yes.

When Show Inherited is set to Yes, Turbo Debugger shows either all the data members or all the member functions of the currently highlighted class, including all the items that the class inherits. If the toggle is set to No, Turbo Debugger displays only the data members or member functions defined within the class being inspected.

Object Inspector windows

While Class Inspector windows provide information about the structure of a class, they say nothing about the data contained in a particular class instance. To view the structure and the values of a specific class instance, use the Object Inspector window.

To open an Object Inspector window, place the cursor on an object name in the Module window, and press *Ctrl+I*.

Figure 11.3 An Object Inspector window
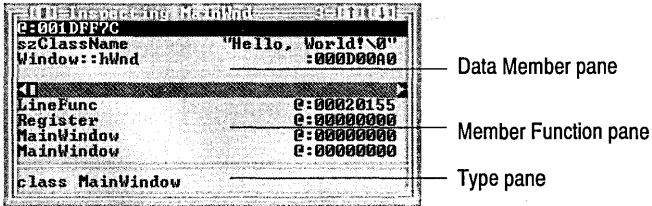
An Object Inspector window contains three panes. The Data Member pane displays the current values of the object's data members. The Member Function pane shows the current values and code addresses of the object's member functions. The Type pane displays the data type of the highlighted data member or member function.

The Object Inspector window SpeedMenus

The Object Inspector window's Data Member and Member Function panes both contain a SpeedMenu. Each menu contains identical commands, except that the Data Member pane contains the additional Change command.

```
Range...
Change...
Methods         Yes
Show inherited  Yes

Inspect
Descend
New expression...
Type cast
Hierarchy
```
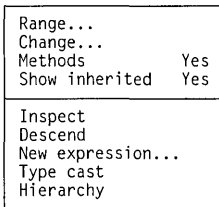
Range
The Range command lets you specify a range of array elements to be displayed. If the currently highlighted item is not an array or a pointer, the item cannot be accessed.

Change
The Change command, available only from the Data Member pane, lets you modify the value of the highlighted data member.

Methods
The Methods command can be toggled between Yes and No; Yes is the default setting. When set to Yes, Turbo Debugger opens the middle pane of the Object Inspector window, where member functions are summarized. When Methods is set to No, the middle pane is not displayed. The Methods setting is carried forward to the next opened Object Inspector window.

Show Inherited
The Show Inherited command is also a Yes/No toggle. When it's set to Yes, all data members and all member functions are shown, whether they are defined within the class being inspected or inherited from a base class. When the command is set to No, only

those data members and member functions defined within the class being inspected are displayed.

Inspect
The Inspect command (which can be opened from the SpeedMenu or by pressing *Enter*) opens an Inspector window on the currently highlighted data member or member function. Inspecting a member function opens the Module view, with the cursor positioned on the code that defines the member function.

Descend
The Descend command works like the Inspect SpeedMenu command, except that it *replaces* the current Inspector window with the new item you want to examine. Using this command reduces the number of Inspector windows onscreen; however, you can't return to a previous Inspector window as you could if you use the Inspect command.

New Expression
Use the New Expression command to inspect a different expression. The data in the current Inspector window is replaced with the data relating to the new expression you enter.

Type Cast
The Type Cast command lets you specify a different data type for the currently highlighted item. This command is useful if your class contains a symbol for which there is no type information, as well as for explicitly setting the type of pointers.

Hierarchy
The Hierarchy command opens the Hierarchy window, displaying the heritage of the class being inspected. The Hierarchy window is described on page 147.

Exceptions

The Exceptions command is found on the SpeedMenu of the Module window. If you have implemented C or C++ exception handling in your program, the Exception command becomes active. Choosing this command opens the Specify C and C++ Exception Handling dialog box:

Figure 11.4   The Specify C and C++ Exception Handling dialog box
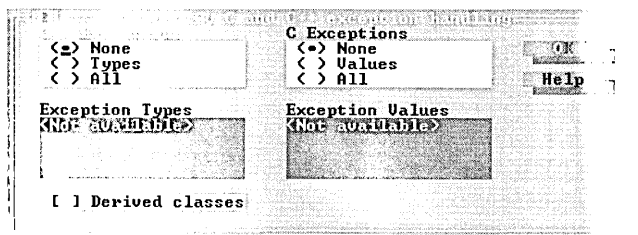
C++ exception handling

If your program implements C++ exception handling using **try, catch,** and **throw** statements, you can specify how you want Turbo Debugger to treat the exceptions your program generates.

Using the C++ Exceptions radio buttons, specify the exception handling in the following ways:

None   Specifies that Turbo Debugger should not interfere with your program's exception handling.

Types   Lets you specify the exception data types you want to trap with Turbo Debugger. Enter the data types of the exceptions you want to trap into the Exception Types input box.

        If you want Turbo Debugger to trap exceptions in classes derived from the ones you enter into the Exception Types input box, check the Derived Classes check box.

All   Specifies that you want Turbo Debugger to trap all exceptions generated by your program.

If, for example, you specify that Turbo Debugger should trap **char*** exceptions, then Turbo Debugger will pause whenever program execution encounters a throw(char *) statement. Once your program has paused, you can examine the Stack window and other views to determine why the exception occurred.

C exception handling

If your C program implements C exception handling, you can control how Turbo Debugger handles the exceptions that your program generates.

Using the C Exceptions radio buttons, specify the C exception handling in the following ways:

None   Specifies that Turbo Debugger should not interfere with your program's exception handling.

Values   Lets you specify the exception values you want to trap with Turbo Debugger. Enter the numbers of the exceptions you want to trap into the Exception values input box.

All   Specifies that you want Turbo Debugger to trap all exceptions generated by your program.

Debugging TSRs and device drivers

Using TD.EXE, you can debug DOS terminate and stay resident (TSR) programs and DOS device drivers. Turbo Debugger has three commands on the file menu that are specifically designed to be used for debugging these types of programs: File | Resident, File | Symbol Load, and File | Table Relocate.

This chapter gives a brief explanation of what TSRs and device drivers are and it provides information on how to debug them with Turbo Debugger.

What's a TSR?

Terminate and stay resident programs (TSRs) are programs that stay in RAM after you "exit" the program. Once you exit the program, you can reinvoke the TSR via special hot keys or from programs that issue special software interrupts. Borland's C and C++ compilers provide a function, **geninterrupt**, that issues such software interrupts.

TSRs consist of two parts: a *transient portion* and a *resident portion*. The transient portion is responsible for loading the resident portion into RAM and for installing an interrupt handler that determines how the TSR is invoked. If the TSR is to be invoked through a software interrupt, the transient portion places the address of the resident portion of the code in the appropriate interrupt vector. If the TSR is to be invoked through a hot key, the resident portion must intercept the DOS interrupt handler for keyboard presses.

When the transient portion is finished executing, it invokes a DOS function that allows a portion of the .EXE file to stay resident in RAM after execution is terminated—hence the phrase "terminate and stay resident." The transient portion of the TSR knows the size of the resident portion as well as the resident portion's location in memory, and passes this information along to DOS. DOS then leaves the specified block of memory alone, but is free to overwrite the unprotected portion of memory. Thus the resident portion stays in memory, while the transient portion can be overwritten.

The trick to debugging TSRs is that you want to be able to debug the resident portion as well as the transient portion. When the .EXE file executes, the only code that is executed

is the transient portion of the TSR. Therefore, when you run a TSR under Turbo Debugger, the only code you see executing is the transient portion as it installs the resident portion and its interrupt handlers. To debug the resident portion of a TSR, you must set a breakpoint in the resident code, and make Turbo Debugger itself go resident.

Debugging a TSR

Debugging the transient portion of a TSR is the same as debugging any other file. It's only when you start to debug the resident portion of your program that anything different happens.

Note If you're debugging the keyboard handler of your TSR (INT 9), use the mouse to navigate through Turbo Debugger. This way, the keyboard handler won't confuse which keys get trapped. If this doesn't work, try using the remote debugging capabilities of Turbo Debugger.

Here's how you debug a TSR program:

1 Compile or assemble the TSR with symbolic debug information.

2 Run Turbo Debugger and load the TSR program.

3 Set a breakpoint at the beginning of the resident portion of the TSR.

4 Run the transient portion of your program by choosing Run | Run.

5 Debug the transient portion of the program using normal debugging techniques.

6 After the transient portion is fully debugged, exit the TSR; the resident portion of the TSR program remains installed in RAM.

7 Choose the File | Resident command to make Turbo Debugger go resident.

This has nothing to do with making your TSR go memory-resident; the TSR goes resident when you run it from Turbo Debugger. Once Turbo Debugger is resident, you can return to DOS and invoke your TSR, which makes its resident portion execute.

8 At the DOS command line, execute the resident portion of your TSR by pressing its hot key (or by doing whatever is needed to invoke it), and run through your program as usual.

9 Exit the TSR program.

The resident portion of the TSR now executes, causing Turbo Debugger to encounter the breakpoint. When the breakpoint activates, Turbo Debugger pauses the TSR at the beginning of the resident portion of the program, and you can debug the resident code. (To reenter Turbo Debugger from DOS, press *Ctrl+Break* twice.)

A second method of debugging a TSR's resident portion involves executing the TSR from the DOS command line and using Turbo Debugger's CPU window to debug the area of RAM containing the TSR:

1 Compile your program with debug information.

2 Use TDSTRIP to strip the symbol table from the program and place it in a .TDS file.

The symbol table contains a set of symbols tied to relative memory locations in your code. The symbols in the symbol table are all prefixed by the characters #FILENAME#, where *FILENAME* is the name of your TSR source file. For example, if your source file was called TSR.ASM and contained a label *Intr*, the symbol #TSR#INTR marks a location in memory.

3 Execute your TSR from the DOS command line.

4 Run TDMEM (described in TD_UTILS.TXT) to obtain a memory map of your computer. Note the segment address at which the resident portion of your TSR is loaded.

5 Run Turbo Debugger and load your TSR's symbol table by choosing File | Symbol Load and specifying the .TDS file you created with the TDSTRIP utility.

6 Set a breakpoint at the beginning of the resident portion of the TSR.

7 Choose the File | Resident command to make Turbo Debugger go resident.

8 At the DOS command line, execute the resident portion of your TSR by pressing its hot key and run through your program as usual.

When your program hits the breakpoint, Turbo Debugger activates with your TSR paused at the beginning of the resident portion of the program. However, to make things easier, synchronize the symbol table with the code in memory.

The symbols in the symbol table are offset from each other by the correct number of bytes, but the absolute location of the first symbol isn't determined yet because DOS might have loaded your TSR at a different absolute memory location than the one at which it was assembled. For this reason, you must use the File | Table Relocate command to explicitly locate the first symbol in memory.

9 Use File | Table Relocate to place the first symbol from the symbol table at the proper location in memory. In this way, the symbolic information present corresponds with your code. To do this, add 10 hex to the segment address *Seg* of your TSR to account for the 256-byte program segment prefix (PSP). Use this number as the TSR segment address in the Table Relocate command.

The disassembled statements from memory are synchronized with information from the symbol table. If your source file is present, source statements are printed on the same line as the information from the symbol table.

10 Use the Goto command (*Ctrl+G*) in the CPU window to go to the segment of RAM containing your TSR. Do this either by giving the segment address of your TSR, followed by offset 0000H, or by going to a specific symbolic label in your code.

11 Debug the resident portion of your TSR.

Once you've finished debugging the TSR, exit the debugging session as follows:

• If you loaded the TSR through Turbo Debugger, exit the debugger by pressing *Alt+X*; the TSR will be unloaded automatically.

• If you're debugging a TSR that you loaded from DOS, run the TSR until Turbo Debugger goes resident and press *Ctrl+Break* twice to bring up Turbo Debugger. Press *Alt+X* to exit Turbo Debugger. This leaves the TSR resident.

What's a device driver?

Device drivers are collections of routines used by DOS to control low-level I/O functions. Installable device drivers (as opposed to those intrinsic to DOS) can be installed from your CONFIG.SYS using commands such as:

```
device = clock.sys
```

When DOS has to perform an I/O operation involving a single character, it scans through a linked list of device headers looking for a device with the appropriate logical name (for example, COM1). In the case of block device drivers (such as disk drives), DOS keeps track of how many block devices have been installed and designates each by a letter, with *A* for the first block device driver installed, *B* for the second, and so on. When you make a reference to drive *C*, for example, DOS knows to call the third block device driver.

The linked list of device headers contains offsets to the two components of the device driver itself, the *strategy routine* and the *interrupt routine*.

When DOS determines that a given device driver needs to be invoked, it calls the driver twice. The first time the driver is called, DOS talks to the strategy routine and passes it a pointer to a memory buffer called the *request header*. The request header contains information about what DOS wants the device driver to do. The strategy routine simply stores this pointer away for later use. On the second call to the device driver, DOS invokes the interrupt routine, which does the actual work specified by DOS in the request header, such as transferring characters in from a disk.

The request header specifies what the device driver is to do through a byte in the request header called a *command code*. This specifies one of a predefined set of operations all device drivers must perform. The set of command codes is different for character device drivers than for block device drivers.

The problem with debugging device drivers is that there is no .EXE file to load into Turbo Debugger; drivers are installed when your computer boots up and have extensions of .SYS, .COM or .BIN. To debug a device driver, it must be resident in memory when you start Turbo Debugger. Hence the functions to load and relocate symbol tables become very useful because they can restore symbolic information to the disassembled segment of memory where the device driver is loaded. The File | Resident command is also very useful.

Debugging a device driver

There are two approaches to debugging device drivers. The first approach is similar to the method shown on page 156 for debugging TSRs. Another approach involves the remote debugging capabilities of Turbo Debugger. To use this approach, read Appendix B for a description of remote debugging, then debug your device driver using the following steps:

1 Compile the device driver with symbolic debug information.

2 Strip the symbolic debug information from the device driver using TDSTRIP (described in TD_UTILS.TXT).

3 Copy the device driver to the remote system.

4 Modify your CONFIG.SYS file on the remote system so that it loads the device driver when it boots up. Then, reboot the remote system to load the device driver.

5 Run TDMEM on the remote system to obtain the memory location of your device driver.

6 Load TDREMOTE on the remote system.

7 Load Turbo Debugger on the local system, connecting it to the remote system.

8 Load in your device driver's symbol table into Turbo Debugger using the File | Symbol Load command.

9 Use the File | Table Relocate command to synchronize the first symbol of the symbol table with the proper location in memory. In this way, the symbolic information present will correspond with your code. To do this, specify the segment address for your device driver (which you determined using TDMEM) to the Table Relocate command prompt.

10 Set a breakpoint at the beginning of the device driver's code.

11 Choose the File | Resident command to make TDREMOTE go resident.

This has nothing to do with making your device driver memory resident; it goes resident when you boot up the remote system. You make TDREMOTE resident so you can return to DOS and do whatever is necessary to invoke your device driver.

12 At the DOS command line on the remote system, perform a command to activate your device driver. For example, send information to whatever device it controls.

13 When your program hits the breakpoint, Turbo Debugger displays the device driver's source code at the appropriate point and you can begin debugging your code. (To reenter Turbo Debugger while DOS is running, press *Ctrl+Break*.)
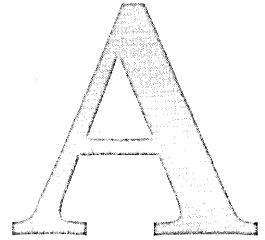
Command-line options

If you start Turbo Debugger from a command line (as described on page 19), you can use the following syntax to configure certain Turbo Debugger options:

```
TD | TDW | TD32 [options] [program_name [program_args]]
```

You can use this syntax to start TD.EXE, TDW.EXE, or TD32.EXE from a command line. In the syntax, items enclosed in square brackets are optional. The *options* item represents Turbo Debugger's command-line options.

Command-line option details

All Turbo Debugger command-line options start with a dash (–) and must be separated from other items in the command line by at least one space. To explicitly turn a command-line option off, follow the option with another dash. For example, –p– disables the mouse.

Any settings you specify using command-line options will take precedence over the settings loaded from Turbo Debugger's configuration files.

The following sections describe Turbo Debugger's command-line options in detail. Unless otherwise noted, all options work the same for TD, TDW, and TD32.

Attaching to a running process

The **–a** options, used only by TD32, lets you attach Turbo Debugger to a process that's already running under Windows NT. See "The Attach command" on page 46 for details on attaching to a running program.

–ar#   The **–ar** option attaches TD32 to process identification number #. The process will continue to run after the attachment is made.

–as#   The **–as** option is the same as the **–ar** option, except that TD32 gains control when the attachment is made.

Loading a specific configuration file (–c)

By default TD.EXE loads the configuration file TDCONFIG.TD, TDW.EXE loads TDCONFIG.TDW, and TD32.EXE loads TDCONFIG.TD2, if the files exist. The **–c***filename* option lets you load a different configuration file, specified by *filename*. There must not be a space between **–c** and the file name.

For example, the following command loads the configuration file MYCONF.TDW and the program MYPROG:

```
TDW -cMYCFG.TDW MYPROG
```

Display updating (–d options)

The **–d** options, used by TD and TDW, affect the way Turbo Debugger updates the display.

–do   The **–do** option enables *dual-monitor debugging*. This lets you view your program's screen on the primary display and Turbo Debugger's on the secondary one. For more information on dual-monitor debugging, see "Dual-monitor debugging" on page 9.

–dp   The **–dp** option, used only with TD.EXE, enables *screen flipping*—Turbo Debugger is displayed on one screen page and the program you're debugging is displayed on a second screen page. Screen flipping minimizes the time it takes to switch between the debugger's screens and your program's. To use this mode, your display adapter must support multiple screen pages and the program you're debugging must not use screen paging.

–ds   This option, known as *screen swapping*, maintains separate screen images in memory for both the debugger and for the program you're debugging. These images are then "swapped" back and forth from memory as each program runs.

        Although this technique is the most time-consuming method for displaying the screens, it is the most reliable method. Because of this, display swapping is turned on by default for all displays.

Getting help (–h and –? options)

The –h and –? options display a help window that describes the command-line syntax and command-line options that are available with each debugger.

Session restart modes (–j options)

The –j options specify how Turbo Debugger should handle the session-state files (described on page 22) when it starts. The options work as follows:

–ji   Don't use the session-state file if you've recompiled your program.

–jn   Turn off session-state restoring (do not use the restart file).

–jp   Prompt if the program has been recompiled since the session-state file was created.

–ju   Always use the session-state file, even if it's old.

Keystroke recording (–k)

The –k option, used only by TD.EXE, enables keystroke recording. When keystroke recording is turned on, all keystrokes you type during a debugging session will be recorded to a disk file, including the keys you press in Turbo Debugger and the keys you press inside your program. Keystroke recording lets you easily recover a previous point in your debugging session. For more information on keystroke recording, see "The Keystroke Recording pane" on page 29.

Assembler-mode startup (–l)

The –l (lowercase ell) option forces the debugger to start in assembler mode. In this mode, Turbo Debugger does not execute your program's startup code as it's loaded into the debugger (which it normally does). Use this option when you want to debug your program's startup code, or the startup code to a DLL.

Mouse support (disabling/enabling)

The –p option enables mouse support. However, since the default for mouse support is *On*, this option is normally used to turn mouse support off (–p–).

Note   If the mouse driver is disabled for Windows, it will also be disabled for Turbo Debugger. In this case, the –p option has no effect.

Remote debugging (–r options)

The –r, –rnL;R, –rp#, and –rs# options, used by TD and TDW, are fully described on page 173.

Source code handling (–s options)

–sc   The –sc option causes Turbo Debugger to ignore the case when you enter symbol names, even if your program was linked with case sensitivity enabled.

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

–sd   The –sd option lets you specify one or more directories that Turbo Debugger should search through to find the source code for your program. The syntax for this option is:

–sd*dirname*[;*dirname*...]

Note   The –sd option doesn't change the starting directory.

To specify multiple directories, separate each directory name with a semicolon (;). Turbo Debugger searches for directories in the order specified. *dirname* can be a relative or absolute path and can include a disk letter. If the configuration file specifies any directories, the ones specified by the –sd option are added to the end of that list. See page 22 for details on how Turbo Debugger searches for source code.

Starting directory (–t)

The –t*dirname* option changes the directory where Turbo Debugger looks for its configuration file and for .EXE files not specified with a full path. There must not be a space between the option and the directory path name, and only a single directory can be specified with this option.

Video hardware handling (–v options)

All –v options, used only by TD.EXE, affect how Turbo Debugger handles the video hardware.

–vg   Saves complete graphics image of your program's screen. Enabling this option uses an extra 8K of memory, but it lets you debug programs that use certain graphic display modes. Try this mode if your program's graphic screens become corrupted when you're running under TD.EXE.

–vn   Disables the 43/50-line display under TD.EXE. You can save some memory by using this option when you know you won't be switching to 43/50-line mode.

–vp   Enables the EGA/VGA palette save. If your program alters the EGA/VGA palette, use this option to have TD.EXE save your program's palette to memory.

Windows crash message checking (–wc)

The –wc option, used only by TDW, disables Turbo Debugger's system crash checking, which is turned on by default.

If your program generates Turbo Debugger's *System crash possible. Continue?* error message, you can use this option to turn the message off. Normally, this error message is generated after you have paused your program's execution with the system interrupt key and then begin to single-step. When you disable the system crash checking, Turbo Debugger issues the message only once, and not as you continue to single step through your program.

Windows DLL checking (–wd)

The **–wd** option, used only by TDW, enables DLL checking by Turbo Debugger. When this option is turned on (the default setting), Turbo Debugger makes a check when your program is loaded to see if all the DLLs used by your program are available. By turning this option off, you can disable the check for the DLLs.

Command-line option summary

Table A.1 lists all of Turbo Debugger's command-line options.

Table A.1   Turbo Debugger's command-line options

Option	Description
–ar#	Attach to process id number # and continue running process.
–as#	Attach to process id number # and give control to Turbo Debugger.
–c*filename*	Use *filename* configuration file.
–do	Display TD.EXE or TDW.EXE on secondary display.
–dp	Enable page flipping for TD.EXE.
–ds	Swap Turbo Debugger and user screens to memory.
–h, –?	Display help screen listing all command-line options.
–ji	Ignore old saved-state information.
–jn	Don't use saved-state information.
–jp	Prompt if saved-state information is old (default).
–ju	Use saved-state information, even if old.
–k	Enable keystroke recording for TD.EXE.
–l	Assembler startup code debugging for applications and DLLs (this option letter is a lowercase ell).
–p	Enable/disable mouse (default is on).
–r	Starts TD.EXE or TDW.EXE with default remote-debugging settings.
–rn*L;R*	Remote debugging over a network.
–rp#	Set port for remote serial debugging.
–rs#	Set speed for remote serial debugging.
–sc	No case-checking of symbols for search strings.
–sd*dir[;dir...]*	Source-file search directories.
–t*directory*	Set starting directory for loading configuration and executable files.
–vg	Save program graphics screen (TD.EXE only).
–vn	Disable 43/50 line display ability for TD.EXE.
–vp	Enable EGA/VGA palette save for TD.EXE.

Table A.1   Turbo Debugger's command-line options (continued)

Option	Description
−wc	Enable/disable System Crash Possible error message (default is enabled).
−wd	Enable/disable checking for the presence of all your program's DLLs (default is on).

Remote debugging

TD and TDW support *remote debugging*, which lets you run Turbo Debugger on one computer and the program you're debugging on another. The two systems can be connected either through serial ports or through a NETBIOS-compatible local area network (LAN).

Remote debugging is useful in several situations:

- If your program uses a lot of memory, and you can't run Turbo Debugger and your program on the same computer.

  If you receive any memory allocation errors while debugging your program, try using two systems to debug your program. The remote debugging drivers (TDREMOTE and WREMOTE) use far less memory than does Turbo Debugger, so the program you're debugging will behave more like it does when it's running without the debugger in the background.

- If you need to debug a device driver.

- If your system has a single monitor, and you don't want to swap screens between Turbo Debugger's character mode screens and your program's graphics mode screens. (However, you might also want to try dual-monitor debugging. For more information on this, see "Dual-monitor debugging" on page 9.)

Hardware and software requirements

You can use either a serial connection or a LAN connection for the remote session. Although the two setups use different hardware, both share the following requirements:

- A development system with enough memory to load Windows and Turbo Debugger. This is the *local* system.

- A second PC with enough memory to load Windows, the remote debugging driver (TDREMOTE or WREMOTE), and the Windows program you want to debug. This is the *remote* system.

For a serial connection, you'll need a *null-modem cable* to connect the serial ports of the two systems; regular serial cables won't send and receive the signals correctly. At the very least, the null-modem cable must swap the transmit and receive lines (lines 2 and 3 on 9-pin and 25-pin cables) of a regular serial cable.

For a LAN connection, you'll need a LAN running Novell Netware-compatible software (IPX and NETBIOS version 3.0 or later). NETBIOS must be loaded onto *both* the local and remote systems before either Turbo Debugger or the remote driver can be loaded.

Starting the remote debugging session

To initiate a remote debugging session, you must:

- Set up the remote system.
- Configure and start WREMOTE, the remote debugging driver.
- Start and configure TDW on the local system.
- Load the program for debugging.

"Remote DOS debugging" on page 174 describes debugging DOS applications with a remote connection.

Setting up the remote system

Before you can begin a remote debugging session, the remote system must contain the following files:

- The program you're debugging.

  The setup on the remote system must include all program support files, such as data input files, configuration files, help files, Windows DLL files, and so on. Set up these files as you would in a normal debugging session. For information on loading your program's .EXE file onto the remote system, see "Automatic file transfer" on page 172.

- WREMOTE.EXE, the remote debugging driver.

- WRSETUP.EXE, the configuration program for WREMOTE.EXE.

Configuring and starting WREMOTE

Before you run WREMOTE, you must first run WRSETUP to establish the communication settings. When you run WRSETUP (by clicking the Remote Setup icon), a window opens displaying the commands File, Settings, and Help. Choose Settings to access the Remote Driver Setting dialog box:

Figure B.1   WRSETUP main window and Settings dialog box
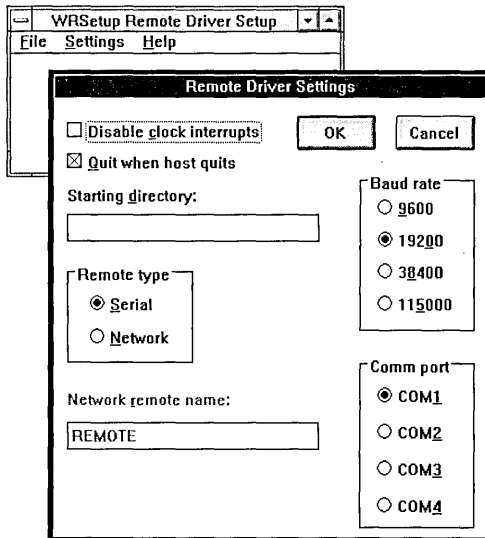
Serial configuration

If you're using a serial connection:

1   Click the Serial radio button.

2   Choose the rate of communications by clicking the appropriate Baud Rate radio button. If you're using the higher transmission speeds (38,400 or 115,000 baud), click the Disable Clock Interrupts check box to help TDW make a reliable connection with WREMOTE.

3   Choose the communications port that works for your hardware setup by clicking the appropriate Comm Port radio button.

4   Enter the directory location of your program in the Starting Directory input box.

5   If you want WREMOTE to return control to Windows when you terminate Turbo Debugger on the local machine, click the Quit When Host Quits check box.

By default, WREMOTE uses a link speed of 19,200 baud, with communications over COM1.

LAN configuration

If you're using a LAN connection:

1   Click the Network radio button.

2   Specify the remote system name in the Network Remote Name input box.

By default, the remote system name is REMOTE. For information on naming the local and remote systems, see "Local and remote system names" on page 173.

3   Enter the directory location of your program in the Starting Directory input box.

4 If you want WREMOTE to return control to Windows when you terminate Turbo Debugger on the local machine, check the Quit When Host Quits check box.

Saving the communication settings

After you've set your options and closed the WRSETUP window, WRSETUP saves your settings to TDW.INI in your Windows directory. The following excerpt from a TDW.INI file shows the WREMOTE settings when you have chosen a serial connection at 19,200 baud on COM2 with clock interrupts disabled and program control returning to Windows when Turbo Debugger terminates:

```
[WRemote]
BaudRate=19200
Port=2
Quit=1
Clock=0
Directory=C:MYPROJ
Type=1
RemoteName=REMOTE
```

Starting WREMOTE

Once WREMOTE is properly configured, you can load it by clicking the Remote Debugging icon, by using the Windows File | Run command, or by using the Windows File Manager. After starting WREMOTE, the mouse cursor on the remote system displays an hourglass, indicating that it's waiting for you to start TDW at the other end of the link. (To terminate WREMOTE while it's waiting to establish a connection with TDW, press *Ctrl+Break* on the remote machine.)

WREMOTE command-line options

If needed, you can use WREMOTE command-line options to override the remote settings in the TDW.INI file. Start an option with either a dash (–) or a slash (/), using the following syntax:

```
WREMOTE [options] [progname [progargs]]
```

Table B.1   WREMOTE command-line options

Option	Description
–c<*filename*>	Uses <*filename*> as the configuration (.INI) file
–d<*dir*>	Uses <*dir*> as the startup directory
–h or –?	Displays the help screen
–rc0	Enables clock interrupts
–rc1	Disables clock interrupts
–rn<*remotename*>	Uses remote LAN debugging
–rp1	Uses port 1 (COM1); default
–rp2	Uses port 2 (COM2)
–rp3	Uses port 3 (COM3)
–rp4	Uses port 4 (COM4)
–rq0	Doesn't return to Windows when you exit Turbo Debugger

Table B.1   WREMOTE command-line options (continued)

Option	Description
–rq1	Returns to Windows when Turbo Debugger exits
–rs1	Uses slowest speed (9,600 baud)
–rs2	Uses slow speed (19,200 baud); default
–rs3	Uses medium speed (38,400 baud)
–rs4	Uses fast speed (115,000 baud)

Starting and configuring TDW

After you've started WREMOTE, you can start TDW. However, before connecting TDW to WREMOTE, it must be configured for the remote session.

The easiest way to configure TDW for the remote debugging session is through the debugger's File | Open command. However, you can also use TDWINST's Options | Miscellaneous command or TDW's command-line options to configure the remote debugging session (for information on the command-line options, see "TDW's remote debugging command-line options" on page 173).

Serial configuration

When you use a null modem cable to connect the local and remote systems, you must specify both the communication rate and the serial port that TDW will use for the connection. To initiate a serial remote debugging session:

1 Start WREMOTE on the remote system (as previously described in this chapter).

2 Start TDW, and choose File | Open to open the Load a New Program to Debug dialog box.

3 Click the Session button to open the Set Session Parameters dialog box.

4 Click the Serial Remote radio button. (Click the Local radio button if you're not using remote debugging.)

5 Choose the serial port of the local system by clicking the appropriate Remote Link Port radio button.

6 Choose the serial communications speed by clicking the appropriate Link Speed radio button.

7 Choose OK to accept the serial communication settings and return you to the Load a New Program to Debug dialog box.

Note   Although the local and remote systems can use different serial ports for the remote link, the link speeds of the two systems must match for the serial connection to work.

LAN configuration

To configure TDW for a remote debugging session on a NETBIOS local area network:

1 Start WREMOTE on the remote system (as previously described in this chapter).

2 Start TDW, and choose File | Open to open the Load a New Program to Debug dialog box.

3 Click the Session button to open the Set Session Parameters dialog box.

4 Choose the Network Remote radio button.

5 Specify the local and remote system names:

By default, Turbo Debugger sets the local and remote system names to LOCAL and REMOTE, respectively. However, if there is more than one remote debugging session running over the same network, you'll have to specify your own system names to uniquely identify the systems you're using.

6 Choose OK to accept the LAN communication settings and return you to the Load a New Program to Debug dialog box.

Initiating the remote link

Once you've configured TDW for the remote debugging session, load your program using the Load a New Program to Debug dialog box (described on page 20). When you load your program, TDW displays the copyright and version information of TDW, and the following message:

```
Waiting for handshake from remote driver (Ctrl+Break to quit)
```

While waiting for a connection, an hourglass is displayed on the remote system. If the link is successful, the hourglass disappears, and Turbo Debugger's normal display appears on the local machine. (Press *Ctrl+Break* to exit TDW if the link is not successful.)

Once you start TDW in remote mode, the Turbo Debugger commands work exactly the same as they do on a single system; there is nothing new to learn. If you access TDW's CPU window, the remote system's CPU type is listed as part of the CPU window title with the word REMOTE before it.

Because the program you're debugging is actually running on the remote system, any screen output or keyboard input to that program happens on the remote system. The Window | User Screen command has no effect when you're running on the remote link.

Automatic file transfer

Once you make a remote connection and load a program into TDW, the debugger automatically checks to see if your program needs to be sent to the remote system.

TDW is smart about loading programs onto the remote system. First, a check is made to see if the program exists in the working directory of the remote system. If the program doesn't exist on the remote system, then it's sent over the link right away. If the program does exist on the remote system, Turbo Debugger checks the time stamp of the program on the local system and compares this with the copy on the remote system. If the program on the local system is later (newer) than the remote copy, Turbo Debugger presumes you've recompiled or relinked the program, and sends it over the link.

At the highest serial link speed (115,000 baud), file transfers move at a rate of approximately 10K per second. Thus, a 60K program takes roughly six seconds to

transfer. To indicate that the system is working, the screen on the remote system adds up the bytes of the file as Turbo Debugger transfers it.

Automatic file transfer can save time and energy. However, TDW transfers only .EXE files; Windows DLL files and other program support files are not transferred to the remote system via automatic file transfer.

TDW's remote debugging command-line options

If you use TDWINST or TDW's command-line options to configure TDW, you must do so before you load TDW. For instructions on using TDWINST, see the online file TD_UTILS.TXT. For details on TDW's remote command-line options, see Table B.2.

If you started TDW without first configuring it for remote debugging, use TDW's File | Open command to configure the remote settings.

Table B.2   TDW's remote debugging command-line options

Option	Description
−r	Initiates remote debugging using the default settings.
−rn$L;R$	Uses remote LAN debugging (see the following section titled "Local and remote system names" for more information).
−rp1	Uses port 1 (COM1); default
−rp2	Uses port 2 (COM2)
−rp3	Uses port 3 (COM3)
−rp4	Uses port 4 (COM4)
−rs1	Uses slowest speed (9,600 baud)
−rs2	Uses slow speed (19,200 baud); default
−rs3	Uses medium speed (38,400 baud)
−rs4	Uses high speed (115,000 baud)

Here's a typical TDW command to start a serial remote connection:

```
TDW -rs3 myprog
```

This command begins the link on the default serial port (usually COM1), at the link speed of 38,400 baud. In addition, the program *myprog* is loaded for debugging.

Local and remote system names

The −rn$L;R$ command-line option takes two optional parameters: the local system name and the remote system name, separated by a semicolon.

Since both parameters are optional, there are four ways to use the −rn command-line option with Turbo Debugger. The following commands all load Turbo Debugger, specify a remote LAN connection, and load the program *filename* for debugging.

```
TDW -rn filename

TDW -rnLOCAL1 filename

TDW -rn;REMOTE1 filename

TDW -rnLOCAL1;REMOTE1 filename
```

The first command uses default names for both the local and remote systems, LOCAL and REMOTE respectively. The second command specifies LOCAL1 as the local system name, but uses the default name (REMOTE) for the remote system. The third command uses the default name for the local system (LOCAL), but specifies REMOTE1 as the remote system name. Finally, the fourth command specifies both local and remote system names.

Note While you can create local and remote system names up to 16 characters in length, the need to specifically name local and remote systems arises only when there are simultaneous remote debugging sessions running on a network. If only one person on a network is using TDW's remote debugging feature, then it isn't necessary to define special local and remote system names.

Remote DOS debugging

You can use TD to debug DOS applications over a remote link just as you use TDW to debug Windows applications remotely. In fact, using TD over a remote link is exactly the same as using TDW over a remote link except that you use the remote driver TDREMOTE on the remote system instead of using WREMOTE. Because of this, you can follow the instructions for remote debugging a Windows application (starting on page 168 with "Starting the remote debugging session") to debug a DOS application over a remote link. To use the TDW instructions, substitute TD for TDW, and TDREMOTE for WREMOTE.

Differences between TDREMOTE and WREMOTE

Although the instructions for debugging a Windows application over a remote link can be used for DOS applications, there is one difference: TDREMOTE does not have a setup program (as does WREMOTE). Because of this, you must use command-line options to configure TDREMOTE when you start it. Use the following to configure TDREMOTE:

```
TDREMOTE [options]
```

The following table summarizes TDREMOTE's command-line options:

Table B.3   TDREMOTE command-line options

Option	Description
–h or –?	Displays the help screen
–rn<*remotename*>	Uses remote LAN debugging
–rp1	Uses port 1 (COM1); default
–rp2	Uses port 2 (COM2)
–rp3	Uses port 3 (COM3)
–rp4	Uses port 4 (COM4)
–rs1	Uses slowest speed (9,600 baud)
–rs2	Uses slow speed (19,200 baud)
–rs3	Uses medium speed (38,400 baud)
–rs4	Uses fast speed (115,000 baud); default

Each TDREMOTE command-line option must be prefixed with either a dash (–) or a slash (/), and it must be separated by other options by a space.

Before starting TDREMOTE, be sure the directory on the remote system is set to the directory that contains the program files. This is essential because TDREMOTE puts the program to be debugged into the directory that is current when you start Turbo Debugger.

When loaded, TDREMOTE signs on with a copyright message, then indicates that it's waiting for you to start TD.EXE at the other end of the link. To stop and return to DOS, press *Ctrl+Break*.

Transferring files to the remote system

To transfer files to the remote DOS system, you can use either floppy disks or TDRF.EXE, the remote file-transfer utility. (The online file TD_UTILS.TXT describes TDRF.EXE)

To send files over to the remote system while running Turbo Debugger, choose File | OS Shell to obtain a DOS prompt and use TDRF to transfer the necessary files. To return to Turbo Debugger, type EXIT at the DOS prompt.

Troubleshooting

Here's a list of troubleshooting techniques you can try if you experience problems with the remote setup:

• Check your cable hookups. This is the most common cause of problems.

• Check to make sure you're using the correct serial port settings (you must use the same link speed on both the local and remote systems) or that you're properly connected to the network.

• With serial connections, try successively slower baud rates until you find a speed that works.

• Some hardware and cable combinations don't always work properly at the highest speed. If the link works only at slower speeds, try a different cable or, if possible, different computers.

• If you can't get the serial connection to work at any speed when you're using TDW, use WRSETUP to *Disable clock interrupts* and try running the link at 9,600 baud. If that works, try successively higher communication speeds.
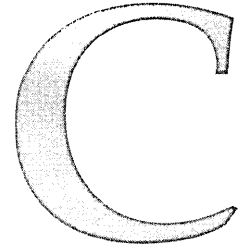
Turbo Debugger error messages

Turbo Debugger can display a variety of messages while you're debugging your program. This Appendix lists the following types of messages:

- Messages generated by TD, TDW, and TD32.

- Status messages listed in the Get Info dialog box and in the Thread Detail pane of the Threads window (page 187).

- Messages generated by TDREMOTE (page 188).

- Messages generated by WREMOTE (page 189).

TD, TDW, and TD32 messages

This section gives an alphabetical listing of the messages generated by TD, TDW, and TD32. Following each message listing is a description that suggests how to handle the message.

Messages can be either error messages (some of them fatal) or messages that prompt you for information. You can easily distinguish an error message from a prompt if you turn on Error Message Beeps in TDWINST or TD32INST.

Fatal messages cause Turbo Debugger to exit to Windows. Although some fatal errors occur when you start Turbo Debugger, others can occur while you're in the middle of debugging your program. In either case, after having solved the problem, your only remedy is to restart Turbo Debugger.

Turbo Debugger displays messages that prompt for information in a dialog box. The title bar of the dialog box contains a description of the type of information that's needed. In some cases, the dialog box will contain a history list of the previous responses you've given.

You can respond to message prompts in one of two ways:

- Enter a response and press *Enter*.
- Press *Esc* to cancel the dialog box.

')' expected
While evaluating an expression, Turbo Debugger found a left parenthesis without a matching right parenthesis.

':' expected
While evaluating a C expression, a question mark (?) separating the first two expressions of the ternary operator (? :) was encountered, but the colon (:) that separates the second and third expressions was not found.

']' expected
While evaluating an expression, Turbo Debugger found a left bracket ([) without a matching right bracket (]).

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.

All threads frozen
You've tried to run or step your Windows NT program after freezing all program threads. For the program to be able to run, you must unfreeze at least one thread using the Options command on the Threads window SpeedMenu.

Already logging to a file
You issued an Open Log File command after having already issued the same command without an intervening Close Log File command. If you want to log to a different file, first close the current log by issuing the Close Log File command.

Already recording, do you want to abort?
You're 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, or press *N* to continue recording.

Ambiguous symbol name
You used a symbol in an expression that does not uniquely identify a C++ member function or Object Pascal method name. Before the expression can be evaluated, you must pick a valid symbol from the list of member functions or methods.

Bad configuration file
Turbo Debugger's configuration file is corrupted.

Bad or missing configuration file
You have specified a nonexistent, corrupted, or outdated file name with the –c command-line option.

Can't do this when debugging an attached process
You cannot reset a program (*Ctrl+F2*) after you have attached to it using TD32's File | Attach command.

Can't execute DOS command processor
You've issued the File | OS Shell command, and Turbo Debugger cannot find COMMAND.COM. Either COMMAND.COM or the COMSPEC environment variable is corrupted.

Can't find *filename*.DLL
This message is generated by Turbo Debugger in two situations:

- You're attempting to load a program that requires one or more DLLs into Turbo Debugger, and the debugger can't locate one of the .DLL files. The DLLs with symbol tables required by your executable must be in the same directory as the program you're debugging.

- You are attempting to load TDW, and the program can't find TDWINTH.DLL. Either you have an invalid file name or path in the DebuggerDLL entry in TDW.INI, or TDW can't find TDW.INI.

  Either edit the DebuggerDLL entry in TDW.INI to reflect the correct path and file name, or if there is no TDW.INI, move TDWINTH.INI to the main Windows directory.

Can't have more than one segment override
You attempted to assemble an instruction where both operands have a segment override. Only one operand can have a segment override. For example,

```
mov es:[bx],ds:1
```

should have been

```
mov es:[bx],1
```

Can't load ___
You specified a bad DLL name in the TDW.INI file.

Can't run TDW on Windows NT
You must use TD32 to debug a 32-bit Windows NT program.

Can't set a breakpoint at this location
You tried to set a breakpoint in ROM or in segment 0. The only way to view the execution of ROM code is to step though it at the instruction level using *Alt+F7*.

Can't set any more hardware breakpoints
The hardware debugging registers have already been allocated by other hardware breakpoints. You can't set another hardware breakpoint without first deleting one you have already set.

Can't set hardware condition on this breakpoint
You've attempted to set a hardware condition on a breakpoint that isn't a global breakpoint. Hardware conditions can only be set on global breakpoints.

Can't set that sort of hardware breakpoint
The hardware device driver that you have installed in your CONFIG.SYS file can't do a hardware breakpoint with the combination of cycle type, address match, and data match that you have specified.

Cannot access an inactive scope
The expression you entered contains a symbol that isn't contained in the current scope. See page 101 for information on scope overrides.

Cannot be changed
You tried to change a symbol that can't be changed. The only symbols that can be changed directly are scalars (**int**, **long**, **real**, **integer**, and so forth) and pointers. If you want to change data in a structure or array, you must change the individual elements one at a time.

Constructors and destructors cannot be called
This error message appears only if you're debugging a program that uses C++ or Object Pascal objects. You tried to evaluate a member function or method that's either a constructor or a destructor; Turbo Debugger cannot evaluate expression that create or destroy objects.

Count value too large
In the Dump pane of the CPU window, you've entered too large a block length to one of the SpeedMenu Block commands. The block length can't exceed FFFFFh.

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.

Device error – Retry?
An error has occurred while writing to a character device, such as the printer. This could be caused by the printer being unplugged, offline, or out of paper. Correct the condition and then press *Y* to retry or *N* to cancel the operation.

Disk error on drive ___ – Retry?
A hardware error has occurred while accessing the indicated drive. This might mean you don't have a floppy disk in the drive or, in the case of a hard disk, it might indicate an unreadable or unwriteable portion of the disk. You can press *Y* to retry the disk read, or, press *N* to cancel the operation.

Display adapter not supported by *filename*
The video driver *filename* indicated in the VideoDLL entry in TDW.INI does not support your display adapter. For more information on video DLL, see the section describing TDWINI.EXE in the online file TD_UTILS.TXT.

Divide by zero
You entered an expression using a divide (/, **div**) or modulus operator (**mod**, %) where the divisor evaluates to zero.

DLL already in list
In the View | Modules dialog box, you tried to add a DLL to the DLLs & Programs list, but the DLL was already in the list.

DLL not loaded
You tried to load a DLL's symbol table before the DLL has been loaded by Turbo Debugger. Make sure that the DLL is loaded before explicitly trying to load its symbol table.

Edit program not specified
You must first specify an editor using TD32INST before you can issue TD32's Edit command.

Error ## loading ___
Error number ## occurred when you attempted to load the DLL listed in the error message.

Error loading *filename*
Turbo Debugger was unable to load the video driver *filename*. The video driver could be an invalid driver file or it could be corrupted. For more information on video drivers, refer to the section describing TDWINI.EXE in the online file TD_UTILS.TXT.

Error opening file ___
Turbo Debugger couldn't open the file that you want to view in the File window. Check to ensure that the file name and path are correct.

Error reading block into memory
The block you specified could not be read from the file into memory. You probably specified a byte count that exceeded the number of bytes in the file.

Error saving configuration
Turbo Debugger couldn't write your configuration to disk. Make sure that your disk contains enough free space for the file.

Error writing block to disk
The block you specified couldn't be written to the disk file. You probably entered a count that exceeded the amount of free space available on your disk.

Error writing log file ___
An error occurred while writing from the Log window to the log file. The file name you supplied for the Open Log File SpeedMenu command can't be opened because there's not enough room to create the file or because 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 writing to file
Turbo Debugger couldn't write your changes back to the disk. The file might be marked as read-only, or an error might have occurred while writing to disk.

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.

Expression with side effects not permitted
You have entered an expression that modifies a memory location when it gets evaluated. There are several places where Turbo Debugger doesn't allow this type of expression; for example, in Inspector windows.

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. You could also have entered a number in the wrong syntax for the language you're using. For example, you might have entered 0xF000 instead of 0F000h as an assembler expression.

Fatal memory error
The Windows memory manager reported a fatal error to Turbo Debugger.

Help file ___ not found
You asked for help, but Turbo Debugger's help file couldn't 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 execute to the start of your source code before you can access most program variables.

Invalid argument list
The expression you entered contains a 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.

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.

Invalid format string
You have entered an invalid format control string after an expression. See Chapter 7 for a description of format strings.

Invalid function parameter(s)
You entered an expression that calls a function, but you supplied incorrect arguments to the call.

Invalid instruction
You entered an instruction to assemble that had a valid instruction mnemonic, but the operand you supplied was invalid.

Invalid instruction mnemonic
When entering an instruction to be assembled, you failed to supply an instruction mnemonic. An instruction consists of an instruction mnemonic followed by optional arguments. For example,

```
AX,123
```

should have been

```
MOV ax,123
```

Invalid number entered
You entered an invalid number in a dialog box. For example, in a File window, you typed an invalid number to go to. Here, entries must be integers greater than zero.

Invalid operand(s)
The instruction you're trying to assemble has one or more operands that aren't allowed. For example, a MOV instruction cannot have two operands that reference memory, and some instructions only work on word-sized operands. For example,

```
POP al
```

should have been

```
POP ax
```

Invalid operator/data combination
You've entered an expression where the operator can't perform its function with the type of operand supplied. For example, you cannot multiply a constant by the address of a function.

Invalid pass count entered

You have entered a breakpoint pass count that is not between 1 and 65,535. Pass counts must be greater than 0; a pass count of 1 means that the breakpoint can activate the first time it's encountered.

Invalid register

You entered an invalid floating-point register as part of an instruction being assembled. A floating-point register consists of the letters *ST*, optionally followed by a number between 0 and 7 within parentheses; for example, ST or ST(4).

Invalid register combination in address expression

When entering an instruction to assemble, you supplied an operand that did not contain one of the permitted combinations of base and index registers. An address expression can contain a base register, an index register, or one of each. The base registers are BX and BP, and the index registers are SI and DI. Here are the valid address register combinations:

```
BX      BX+SI
BP      BP+SI
DI      BX+DI
SI      BP+DI
```

Invalid register in address expression

You entered an instruction to assemble using 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 switch: __

You supplied an invalid option switch on the command line. Appendix A discusses each command-line option in detail.

Invalid symbol in operand

When entering an instruction to assemble, you started an operand with a character that cannot be used to start an operand: for example, the colon (:).

Invalid typecast

A correct C or C++ typecast 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
```

Invalid value entered

When prompted to enter a memory address, you supplied a floating-point value instead of an integer value.

Invalid window handle

In TDW, you tried to indicate a window using a window handle. The handle must be initialized before it can be used to specify a window for message tracking. Run your program past the point where the handle is initialized.

Invalid ___, missing ___

This fatal error message occurs when you have written your own video or keyboard DLL to work with Turbo Debugger, but have left out a section in the DLL. The name of the DLL is given in the first field, and the missing section is listed in the second field.

Keyword not a symbol

The expression you entered contains a keyword where a variable name was expected. You can only use keywords as part of typecast operations, with the exception of the sizeof special operator. For example,

```
floatval = char charval
```

should have been

```
floatval = char (charval)
```

Left side not a record, structure, or union

You entered an expression that used one of the C structure member selectors (. or ->) symbol, however, was not preceded by a structure name, nor was it preceded by a pointer to a structure.

No C or C++ exception handler
You tried to access the Module window SpeedMenu Exception command. To access this command, your program must include exception-handling routines.

No coprocessor or emulator installed
You tried to open a Numeric Processor window using the View | Numeric Processor command, but there is no numeric processor chip installed in your system, and the program you're debugging either doesn't use the software emulator or the emulator has not been initialized.

No hardware debugging available
You have tried to set a hardware breakpoint, but you don't have the hardware debugging device driver installed. You can also get this error if your hardware debugging device driver does not find the hardware it needs. See page 74 for more information on hardware breakpoints.

No help for this context
You pressed *F1* to get help, but Turbo Debugger could not find a relevant help screen. Please report this to Borland Technical Support.

No modules have line number information
You issued the View | Module command, but Turbo Debugger can't find any modules with debug information. This message usually occurs when you're debugging a program without a symbol table. See the "Program has no symbol table" error message entry on page 185 for more information on symbol tables.

No network present
You have attempted to start Turbo Debugger using a remote network connection, but Turbo Debugger couldn't detect a NETBIOS network connection.

No pending status from program being debugged
You've issued TD32's Run | Next Pending Status command, but your program has no events waiting in the operating system.

No previous search expression
You attempted to perform a Next command from the SpeedMenu of a text pane, but you had not previously issued a Search command to specify what to search for.

No program loaded
You attempted to issue a command that requires a program to be loaded. For example, none of the commands in the Run menu can be performed without first loading a program.

No type information for this symbol
You entered an expression that contains a symbol not found in the debug information. Check to ensure that you typed the symbol name correctly.

Not a function name
You entered an expression that contains a call to a routine, but the routine cannot be found. Any time a pair of parentheses immediately follows a symbol, the expression parser presumes that you intended to call a routine.

Not a record, structure, or union member
You entered an expression that used one of the C structure member selectors (. or ->) the symbol wasn't preceded by a structure name or a pointer to a structure.

Not a 32-bit program
You've tried to load a 16-bit program into TD32 running under Windows 32s or Windows NT. Exit TD32, and use TDW to debug the 16-bit program.

Not a Windows program
You can only use TDW to debug Windows programs.

Not enough memory
Turbo Debugger ran out of working memory while loading.

Not enough memory for selected operation
Your system ran out of working memory while trying to open a new Turbo Debugger window. Try closing some other windows before you reissue the command.

Not enough memory to load *filename*
Turbo Debugger ran out of working memory while loading the video driver *filename*.

Not enough memory to load program
Your program's symbol table has been successfully loaded into memory, but there is not enough memory left to load your program.

Not enough memory to load symbol table
There is not enough room to load your program's symbol table into memory. When this message is issued, you must free enough memory to load both your program and its symbol table. Try making the symbol table smaller by generating debug information for only the necessary source modules.

Old or invalid configuration file
You've attempted to start Turbo Debugger using a configuration file from a previous version of the debugger.

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. Some things that must reference memory include the assignment operator and the C increment and decrement (++ and --) operators.

Operand size unknown
You entered an instruction to assemble, but did not specify the size of the operand. Some instructions that can act on bytes or words require you to specify which size to use if it cannot be deduced from the operands. For example,

```
add [bx],1
```

should have been

```
add BYTE PTR[bx],1
```

Overwrite ___?
You tried to write to an already existing file. You can choose to overwrite the file, replacing its previous contents, or you can cancel the command and leave the previous file intact.

Overwrite existing macro on selected key
You have pressed a key to record a macro, and that key already has a macro assigned to it. If you want to overwrite the existing macro, press *Y*; otherwise, press *N* to cancel the command.

Path not found
You entered a drive and directory combination that does not exist. Check that you have specified the correct drive and that the directory path is spelled correctly.

Path or file not found
You specified a nonexistent or invalid file name or path when prompted for a file name to load. If you do not know the exact name of the file you want to load, you can pick the file name from a list by pressing *Browse*.

Press key to assign macro to
Press the key that you want to assign the macro to. Then, press the keys to do the command sequence that you want to assign to the macro key. The command sequence will actually be performed as you type it. To end the macro recording sequence, press the key you assigned the macro to, or press *Alt+-* (the Alt key plus the hyphen key).

Program already terminated, Reload?
You have attempted to run or step your program after it has already terminated. If you choose *Y*, your program will be reloaded. If you choose *N*, your program will not be reloaded, and your run or step command will not be executed.

Program has invalid symbol table
The symbol table attached to your program has become corrupted. You must recompile your program with debug information.

Program has no objects or classes
You've attempted to open a View | Hierarchy window on a program that isn't object-oriented.

Program has no symbol table
The program you want to debug has been successfully loaded, but it doesn't contain symbolic debug information. You'll be able to use the CPU view to debug your program, but you wont be able to use the program's source code or symbol names while debugging. Refer to Chapter 2 for information on compiling your program for debugging.

Program has no threads
You tried to open the Threads window in TD32 (using *Ctrl+T*) while running Windows 32s. Windows 32s doesn't support process threads.

Program is running
You issued a command to run your program in TD32 under Windows NT, but the program was already running.

Program linked with wrong linker version
You loaded a program with out-of-date debug information. Recompile your program using the latest version of the compiler.

Program not found
The program name you specified does not exist. Either supply the correct name or pick the program name from the file list.

Program out of date on remote, send over link?
When you start a remote debugging session, Turbo Debugger checks to see if the .EXE file on the remote system is the latest version of the program. If the program on the local system is newer than the copy on the remote system, you will receive this prompt. Enter *Y* if you want to send your program over the link, or *N* if you don't.

Register cannot be used with this operator
You have entered an instruction to assemble that attempts to use a base or index register as a negative displacement. You can only use base and index registers as positive offsets. For example,

```
INC WORD PTR[12-BX]
```

should have been

```
INC WORD PTR[12+BX]
```

Register or displacement expected
The instruction you tried to assemble has a badly formed expression between brackets ([ ]). You can only put register names or constant displacement values between the base-index brackets.

Remote link timeout
The connection to the remote system has been disrupted. Try rebooting both the systems and starting again. For details on remote debugging, see Appendix B.

Restart info is old, use anyhow?
When starting Turbo Debugger, it restores the settings of the previous debugging session. If the program has been changed since you last loaded it into the debugger, you will receive the prompt. See page 22 for more information on session-state saving.

Run out of space for keystroke macros
The macro you are recording has run out of space. You can record up to 256 keystrokes for all macros.

Search expression not found
The text or bytes that you specified could not be found. The search starts at the current location in the file, as indicated by the cursor, and proceeds forward. If you want to search the entire file, press *Ctrl+PgUp* before issuing the search command.

Source file ___ not found
Turbo Debugger can't find the source file for the module you want to examine. See page 22 for more information on how Turbo Debugger searches for source code.

Symbol not found
You entered an expression that contains an invalid variable name. Make sure that you correctly spelled the symbol name, and that it's in scope.

Symbol table file not found
The symbol table file that you have specified does not exist. You can specify either a .TDS or .EXE file for the symbol file.

Syntax error
You entered an expression that doesn't conform to the syntax of the selected language parser.

System crash possible, continue?
After pressing the program interrupt key, Turbo Debugger gained control while your program was executing Windows kernel code. If you try to exit Turbo Debugger, or reset your program, this error message is generated. Exiting Turbo Debugger or reloading your program while paused inside Windows kernel code will have unpredictable results, most likely hanging the system and forcing a reboot.

To remedy this situation, set a breakpoint in your code and run your program to that breakpoint. When the breakpoint activates, you can either exit Turbo Debugger, or reset your program.

The –wc command-line option controls the generation of this error message.

Too many files match wildcard mask
You specified a wildcard file mask that specifies more files than can be handled. TDW can display up to 1,000 file names, and TD32 can display up to 10,000 file names.

Unexpected end of line
While evaluating an expression, the end of your expression was encountered before a valid expression was recognized.

Unknown character
You entered an expression that contains an illegal character, such as a reverse single quote (').

Unknown record, structure, or union name
You have entered an expression that contains a typecast with an unknown record or **enum** name. (Note that assembler structures have their own name space different from variables.)

Unknown symbol
You entered an expression that contained an invalid symbol name. Make sure the module name, symbol name, or line number is correct.

Unterminated string
You entered a string that did not end with a closing quote ("). To enter a string with quote characters, you must precede each quote with a backslash (\) character.

Value must be between *nn* and *nn*
You have entered an invalid numeric value for an editor setting (such as the tab width) or printer setting (such as the number of lines per page). The error message will tell you the allowed range of numbers.

Value must be between 1 and 32 tenths of a second
The value entered for the background screen updating must be an integer between 1 and 32.

Value out of range
You have entered a value for a variable that is outside the range of allowed values.

Variable not available
The variable in question has been optimized away by the compiler and cannot be accessed by the debugger. For best results, compile without optimizations while you're developing your program.

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 use 43/50-line mode only with EGA, VGA or SVGA video adapters.

Video mode not supported by *filename*
The video mode Windows is using isn't supported by the video DLL indicated in the VideoDLL entry in the TDW.INI file. Refer to the description of TDWINI.EXE in the online file TD_UTILS.TXT for more information on video DLLs.

Video mode switched while flipping pages
Your program has changed the video display mode when Turbo Debugger is in page flipping mode. This means that the contents of your program's screen might be lost. You can avoid this by using the –ds command-line option to turn on the video swapping mode.

Waiting for remote driver. Press Esc to stop waiting
You've configured TDW for remote debugging either through a serial or network connection, and it is now waiting to connect to WREMOTE on the remote system. Press *Esc* to exit the debugger. See Appendix B for details on remote debugging.

Wrong version of remote driver
TDW tried making a remote connection with WREMOTE, but the version of WREMOTE does not match that of TDW. Make sure that TDW and WREMOTE are installed from the same Borland software package.

You must run WREMOTE on remote system
Make sure that the remote system is running WREMOTE, and not a copy of TDREMOTE used with earlier versions of Turbo Debugger.

Status messages

Here are the messages you'll see on the Status line of the Get Info text box and in the Thread Detail pane of the Threads window. These messages describe how Turbo Debugger gained control from your running process.

Breakpoint at __
Your program encountered a breakpoint that was set to pause your program. The text after "at" is the address of the breakpoint.

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

Exception __
A processor exception has occurred, which usually happens when your program attempts to execute an illegal instruction opcode. The Intel processor documentation describes the exception codes in detail.

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

Global breakpoint __ at __
A global breakpoint has been activated. This status message includes the breakpoint number and the address where the breakpoint occurred.

Interrupt
You pressed the program interrupt key to regain control.

Loaded
You either reset your program or loaded it without executing any startup code. Because no instructions have been executed at this point (including those that set up your stack and segment registers), most of Turbo Debugger's windows show incorrect data.

No program loaded
You started Turbo Debugger without loading a program. You cannot execute any code until you either load a program or assemble some instructions using the Assemble SpeedMenu command in the Code pane of a CPU window.

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

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

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

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

Window message breakpoint at __
Your program encountered a message breakpoint that paused your program. The text after "at" is the window procedure that handles the message received.

TDREMOTE messages

Here's the list of error messages that can be generated by TDREMOTE.

Can't create file
TDREMOTE can't create a file on the remote system. This can happen if there isn't enough room on the remote disk to transfer the executable program across the link.

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 must delete some files before TDREMOTE can successfully download the file.

Interrupted
You pressed *Ctrl+Break* while waiting for communications to be established with the other system.

Invalid command-line option
You gave an invalid command-line option when you started TDRF from the DOS command line.

Link broken
The program communicating with TDREMOTE has stopped and returned to DOS.

Program load failed, EXEC failure
DOS could not load the program into memory. This can happen if the program has become corrupted or truncated. Delete the program file from the remote system's disk to force Turbo Debugger to send a new copy over the link. If this message happens again after deleting the file, you should relink your program using TLINK on the local system and try again.

Program load failed; not enough memory
The remote system doesn't have enough free memory to load the program you want to debug.

Program load failed; program not found
TDREMOTE could not find the program on its disk. This should never happen because Turbo Debugger downloads the program to the remote system if TDREMOTE can't find it.

Unknown request: *message*
TDREMOTE has received an invalid request from the local system (where you're running Turbo Debugger). If you get this message, check that the link cable is in good working order. If you keep getting this error, try reducing the link speed (use the –rs command-line option).

WREMOTE messages

Here's the list of error messages that can be generated by WREMOTE.

Can't find configuration file: ___
The file you specified using the –c command-line option cannot be found. Check to ensure the path and file name are spelled correctly.

Can't open COM*x* serial port
WREMOTE is trying to use a COM port that is either in use or doesn't exist.

Invalid switch
You specified an unknown option on the WREMOTE command line. Refer to Appendix B for a description of WREMOTE command-line options.

No network present
WREMOTE is unable to detect a NETBIOS compatible network. Make sure you have loaded NETBIOS (version 3.0 or greater), and are logged onto the network.

Index

Symbols

!! (exclamation points), in Load
Module Source or DLL
Symbols dialog box 138
(cross hatch)
    in CPU window 117
    in expressions 99, 102
* (asterisk)
    in Breakpoints window 74
    in Clipboard window 45
    in Hierarchy window 148
    in Load Module Source or
       DLL Symbols dialog
       box 136
** (asterisks), in Hierarchy
    window 148
▸ (arrow)
    in CPU window 117, 124
    in Module window 108
-? command-line option 163
???? (four question marks)
    in CPU window 116
    in Variables window 86
    in Watches window 84
• (bullet)
    in Load Module Source or
       DLL Symbols dialog
       box 136
    in Module window 108
    in Specify Exception
       Handling dialog box 143
≡ menu (System) 35

Numerics

80x87 processors 39

A

-a command-line options 162
Action Expression input box 75
Action radio buttons 71, 74–75
activity indicator 48
    READY 42
    RECORDING 42
    REMOTE 172
adapters See video adapters
Add command
    breakpoint groups 78
    Breakpoints window 70
    Windows Messages
       window 130, 132

Add Comment command (Log
    window) 81, 82
Add DLL button 137
Add Group dialog box 78
Add Watch command (Data
    menu) 84
Add Window or Handle to
    Watch dialog box 130
Add Window Procedure to
    Watch dialog box 130
Address input box 70, 73
addresses
    expressions 99
    navigating to 111, 118
    running to specified 25
    setting breakpoints 70, 73
    shifting 116
    viewing invalid (CPU
       window) 116
All Threads check box 80
All Threads command (Threads
    window) 141
allocating memory 46
Alt+key shortcuts See hot keys
Animate command (Run
    menu) 25
Another command (View
    menu) 40
arguments
    calling function 38
    command-line 26
    defined 3
    this 84
Arguments command (Run
    menu) 26
arrays
    displaying character
       strings 101
    inspecting 89, 90
       See also Inspector windows
       subranges 91
    modifying 179
arrow keys, in CPU window 116
ASCII files, viewing 112
Assemble command (CPU
    window) 120
assembler
    instructions See machine
       instructions
    registers See CPU window,
       registers
assignment operator 100

At command (Breakpoints
    window) 70
Attach command (TD32's File
    menu) 46
Attach to and Debug a Running
    Process dialog box 47
automatic name completion 41

B

Back Trace command (Run
    menu) 26
Background Delay input box 11
backward trace See reverse
    execution
Baud radio buttons 169
Block command (CPU
    window) 124
Borland, contacting 6
Breakpoint Options dialog
    box 71
breakpoints
    See also Breakpoints window
    action sets 76
    actions 68, 74–75
    changed-memory 72
    condition sets 76
    conditions 68
    CPU window 118
    defined 67
    disabling/enabling 75, 79
    expression-true 70
    global 68, 73
       Always action and 73
    groups 75, 77–79
    hardware 74
       problems with 179, 183
    inspecting source 79
    instrumentation 75
    line numbers and 23
    location 67
    logging values 75
    modifying 71
    pass counts 68, 72
    reloading programs 31
    removing 79
    saving 23
    scope of expressions 77
    setting 69
       in different modules 77
    simple 69
    templates and 80
    threads and 80

Windows 32s, support files 13
Windows Information dialog
  box 143
Windows Messages
  window 40, 129
    See also window messages

WREMOTE.EXE 13
  command-line options 170
  configuring 168
  error messages 189
WRSETUP.EXE 14

Z

Zero command (CPU
  window) 121
Zoom command (Window
  menu) 36

Borland