

Turbo Debugger for Windows[®]

Version 3.0

User's Guide

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

Copyright © 1991 by Borland International. All rights reserved. All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders. Windows, as used in this manual, shall refer to Microsoft's implementation of a windows system.

R1

10 9 8 7 6 5 4 3 2 1

C O N T E N T S

Introduction	1	Menus and dialog boxes	19
New features and changes for version 3.0	1	Using the menus	19
Hardware and software requirements	2	Dialog boxes	20
A note on terminology	3	Knowing where you are	21
What's in the manual	3	Local menus	22
How to contact Borland	5	History lessons	24
Resources in your package	5	Automatic name completion	25
Borland resources	5	Incremental matching	25
Recommended reading	6	Making macros	26
Chapter 1 Getting started	9	Window shopping	26
The distribution disks	9	Windows from the View menu	26
Online text files	9	Breakpoints window	26
The README file	10	Stack window	27
The MANUAL.TDW file	10	Log window	27
The HELPME!.TDW file	10	Watches window	27
The ASMDEBUG.TDW file	10	Variables window	27
The UTILS.TDW file	11	Module window	28
Installing TDW	12	File window	28
Installing TDDEBUG.386	12	CPU window	28
Hardware debugging	13	Dump window	28
Where to now?	13	Registers window	29
Programmers learning Turbo C++	13	Numeric Processor window	29
Turbo C++ pros, but Turbo Debugger	13	Execution History window	29
novices	13	Hierarchy window	29
Programmers experienced with Turbo	13	Windows Messages window	30
Debugger	14	Clipboard window	30
Chapter 2 TDW basics	15	Duplicate windows	30
Is there a bug?	15	User screen	30
Where is it?	16	Inspector windows	31
What is it?	16	The active window	31
Fixing it	16	What's in a window	32
What TDW can do for you	16	Working with windows	33
What TDW won't do	18	Window hopping	33
How TDW does it	18	Moving and resizing windows	35
The TDW advantage	18	Closing and recovering windows	36
		Saving your window layout	36

Copying and pasting	36	Starting directory (-t)	63
The Pick dialog box	37	Configuration files	63
The Clipboard window	38	The Options menu	64
Clipboard item types	38	The Language command	64
The Clipboard window local menu	39	The Macros menu	64
Dynamic updating	39	Create	64
Tips for using the Clipboard	40	Stop Recording	64
Getting help	40	Remove	65
Online help	41	Delete All	65
The status line	42	Display Options command	65
In a window	42	Display Swapping	65
In a menu or dialog box	42	Integer Format	66
Chapter 3 A quick example	43	Screen Lines	66
The demo program	43	Tab Size	66
Using TDW	45	Path for Source command	66
The menus	45	Save Options command	66
The status line	46	Restore Options command	67
The windows	46	Returning to Windows	67
Using the C demo program	48	Chapter 5 Controlling program execution	69
Setting breakpoints	49	Examining the current program state ...	70
Using watches	50	The Variables window	70
Examining simple C data objects	51	The Global pane local menu	71
Examining compound C data objects	53	Inspect	71
Changing C data values	53	Change	72
Chapter 4 Starting TDW	57	Watch	72
Preparing programs for debugging	57	The Static pane local menu	72
Starting TDW	58	Inspect	72
Entering command-line options	59	Change	72
Directly entering command-line options	59	Watch	73
Entering command-line options from TCW	59	Show	73
Things to remember	60	The Stack window	73
Running TDW	60	The Stack window local menu	74
Command-line options	60	Inspect	74
Loading the configuration file (-c)	61	Locals	74
Display updating (-d)	61	The Origin local menu command	75
Getting help (-h and -?)	61	The Get Info command	75
Assembler-mode startup (-l)	62	Global memory information	75
Mouse support (-p)	62	Status line messages	76
Source code handling (-s)	62	The Run menu	78
		Run	78
		Go to Cursor	78
		Trace Into	78

Step Over	79	Pointers	98
Execute To	79	Arrays	99
Until Return	79	Structures and unions	100
Animate	80	The Inspector window local menu	100
Back Trace	80	Range	101
Instruction Trace	80	Change	101
Arguments	81	Inspect	101
Program Reset	81	Descend	101
The Execution History window	81	New Expression	102
The local menu	82	Type Cast	102
Inspect	82	Chapter 7 Breakpoints	103
Reverse Execute	82	The Breakpoints menu	104
Full History	83	Toggle	105
Interrupting program execution	83	At	105
Program termination	84	Changed memory global	105
Restarting a debugging session	85	Expression true global	105
Opening a new program to debug	85	Hardware breakpoint	105
Changing the program arguments	86	Delete all	105
Chapter 6 Examining and modifying data	87	The Breakpoints window	105
The Data menu	88	The Breakpoints window local menu	106
Inspect	88	Set Options	106
Evaluate/Modify	88	Add	106
Add Watch	91	Remove	107
Function Return	91	Delete all	107
Pointing at data objects in source files	91	Inspect	107
The Watches window	92	Group	107
The Watches window local menu	93	Groups	108
Watch	93	Add	108
Edit	93	Delete	109
Remove	93	Enable	109
Delete All	93	Disable	109
Inspect	93	The Breakpoint Options dialog box	109
Change	94	Address	110
Inspector windows	94	Group ID	110
C data Inspector windows	95	Global	110
Scalars	95	Disabled	110
Pointers	95	Conditions and Actions	111
Structures and unions	96	Change	111
Arrays	97	Add	111
Functions	97	Delete	111
Assembler data Inspector windows	98	The Conditions and Actions dialog box	111
Scalars	98	The condition radio buttons	112

Always	112	Previous	126
Changed memory	112	Line	126
Expression true	112	Search	126
Hardware	112	Next	127
The action radio buttons	113	Origin	127
Break	113	Goto	127
Execute	113	Examining other disk files	127
Log	113	The File window	127
Enable group	113	The File window local menu	128
Disable group	113	Goto	129
Setting conditions and actions	114	Search	129
Condition Expression	114	Next	130
Action Expression	115	Display As	130
Pass count	115	File	130
Customizing breakpoints	116	Chapter 9 Expressions	131
Simple breakpoints	116	Choosing the language for expression	
Global breakpoints	116	evaluation	132
Changed memory breakpoints	117	Code addresses, data addresses, and line	
Conditional expressions	118	numbers	132
Scope of breakpoint expressions ..	118	Accessing symbols outside the current	
Hardware breakpoints	118	scope	133
Logging variable values	119	Scope override syntax	134
Breakpoints and templates	119	Overriding scope in C, C++, and	
Breakpoints on class templates	119	assembler programs	134
Breakpoints on function templates ..	120	Scope override tips	136
Breakpoints on template class		Overriding scope in Pascal	
instances and objects	120	programs	137
The Log window	120	Scope override tips	138
The Log window local menu	121	Scope and DLLs	138
Open Log File	121	Implied scope for expression	
Close Log File	122	evaluation	139
Logging	122	Byte lists	139
Add Comment	122	C expressions	140
Erase Log	122	C symbols	140
Display Windows Info	122	C register pseudovariables	140
Chapter 8 Examining files	123	C constants and number formats	142
Examining program source files	123	Escape sequences	142
The Module window	124	C operator precedence	143
The Module window local menu	125	Executing C functions in your pro-	
Inspect	125	gram	144
Watch	125	C expressions with side effects	144
Module	126	C reserved words and type	
File	126	conversion	145

Assembler expressions	146
Assembler symbols	146
Assembler constants	146
Assembler operators	147
Format control	147

Chapter 10 Object-oriented debugging

Chapter 10 Object-oriented debugging	149
The Hierarchy window	149
The Class List pane	150
The Class List pane local menu ...	150
Inspect	150
Tree	151
The Hierarchy Tree pane	151
The Hierarchy Tree pane local menu(s)	151
The Parent Tree pane	151
The Parent Tree pane local menu ..	152
Class Inspector windows	152
The class Inspector window local menus	153
The Data Member (top) pane	153
Inspect	153
Hierarchy	153
Show Inherited	153
The Member Function (bottom) pane	154
Inspect	154
Hierarchy	154
Show Inherited	154
Object Inspector windows	154
The object Inspector window local menus	155
Range	155
Change	155
Methods	155
Show Inherited	156
Inspect	156
Descend	156
New Expression	156
Type Cast	156
Hierarchy	156
The middle and bottom panes	156

Chapter 11 Using Windows debugging features

Chapter 11 Using Windows debugging features	157
Windows features	157
Logging window messages	158
Selecting a window for a standard Windows application	158
Adding a window selection for a standard Windows application ..	159
Selecting a window for an ObjectWindows application	159
Obtaining a window handle	159
Specifying a window with ObjectWindows support enabled	160
Adding a window with ObjectWindows support enabled	161
Deleting a window selection	162
Specifying a message class and action	162
Adding a message class	162
Deleting a message class	164
Window message tips	165
Viewing messages	165
Obtaining memory and module lists ..	166
Listing the contents of the global heap	166
Listing the contents of the local heap	168
Obtaining a list of modules	168
Debugging dynamic link libraries (DLLs)	169
Using the Load Modules or DLLs dialog box	170
Changing source modules	170
Working with DLLs and programs	171
Adding a DLL to the DLLs & Programs list	172
Setting debug options in a DLL ...	173
Controlling TDW's loading of DLL symbol tables	173
Debugging DLL startup code	173

Converting memory handles to addresses	175	Instructions pane	194
Chapter 12 Assembler-level debugging	177	File window	194
When source debugging isn't enough ..	177	Log window menu	194
The CPU window	178	Module window	195
The Code pane	180	Windows Messages window	195
The disassembler	180	Window Selection pane	195
The Register and Flags panes	181	Message Class pane	196
The Selector pane	181	Messages pane	196
The Selector pane local menu	182	Clipboard window	196
Selector	182	Numeric Processor window	197
Examine	182	Register pane	197
The Data pane	183	Status pane	197
The Stack pane	183	Control pane	197
The Dump window	184	Hierarchy window	197
The Registers window	184	Class pane	197
Chapter 13 Command reference	185	Hierarchy Tree pane	198
Hot keys	185	Parent Tree pane	198
Commands from the menu bar	187	Registers window menu	198
The ≡ (System) menu	187	Stack window	198
The File menu	187	Variables window	198
The Edit menu	188	Global Symbol pane	198
The View menu	188	Local Symbol pane	199
The Run menu	189	Watches window	199
The Breakpoints menu	189	Inspector window	200
The Data menu	189	Class Inspector window	200
The Options menu	189	Object Inspector window	200
The Window menu	190	Text panes	201
The Help Menu	190	List panes	202
The local menu commands	190	Commands in input and history list boxes	202
Breakpoints window	191	Window movement commands	203
The CPU window menus	191	Wildcard search templates	204
Code pane	191	Complete menu tree	204
Selector pane	192	Chapter 14 Debugging a standard C application	207
Data pane	192	When things don't work	207
Flags pane	193	Debugging style	208
Register pane	193	Run the whole thing	209
Stack pane	193	Incremental testing	209
Dump window	194	Types of bugs	209
The Execution History window menus	194	General bugs	210
		Hidden effects	210
		Assuming initialized data	210

Not cleaning up	210	WMRButtonDown	226
Fencepost errors	211	WMMouseMove	226
C-specific bugs	211	The pen-color routines	226
Using uninitialized automatic		Creating the application	226
variables	211	Debugging the program	227
Confusing = and ==	212	Finding the first bug	227
Confusing operator precedence ...	212	Finding the function that called	
Bad pointer arithmetic	212	Windows	227
Unexpected sign extension	213	Debugging WMLButtonDown	228
Unexpected truncation	213	Debugging MoveTo	228
Misplaced semicolons	213	Fixing the bug	229
Macros with side effects	214	Testing the fix	229
Repeated autovisible names	214	Finding the pen color bug	230
Misuse of autovisibles	214	Setting a window message	
Undefined function return value ..	214	breakpoint	230
Misuse of break keyword	215	Setting a window message	
Code has no effect	215	breakpoint with a handle	231
Accuracy testing	215	Setting a window message	
Testing boundary conditions	216	breakpoint with a window	
Invalid data input	216	object	232
Empty data input	216	Inspecting wParam	233
Debugging as part of program design ..	216	Testing the fix	234
The sample debugging session	217	Finding the off-screen drawing bug .	234
Looking for errors	217	Logging the window messages ...	234
Deciding your plan of attack	218	Discovering the bug	234
Starting Turbo Debugger	218	Fixing the bug	235
Inspecting	219	Testing the fix	236
Breakpoints	219	Finding the erase-screen bug	236
The Watches window	220	Analyzing the cause of the bug ...	237
The Evaluate/Modify dialog box	220	Fixing the bug	238
Eureka!	221	Testing the fix	238
Chapter 15 Debugging an		Appendix A Summary of command-line	
ObjectWindows		options	239
application	223	Appendix B Error and information	
About the program	223	messages	241
The Color Scribble window type		Dialog box messages	241
definition	224	Error messages	248
ScribbleWindow	225	Fatal errors	248
GetWindowClass	225	Other error messages	249
WMLButtonDown	226	Index	263
WMLButtonUp	226		

T A B L E S

2.1: What goes in a dialog box	21	13.1: The function key and hot key commands	186
2.2: Clipboard item types	38	13.2: Text pane key commands	201
2.3: Clipboard local menu commands . . .	39	13.3: List pane key commands	202
11.1: Windows message classes	163	13.4: Dialog box key commands	203
11.2: Format of a global heap list	167	13.5: Window movement key commands	203
11.3: Format of a local heap list	168	A.1: TDW command-line options	239
11.4: Format of a Windows module list . .	169		
11.5: DLLs & Programs list dialog box controls	171		

F I G U R E S

2.1: Global and local menus	23	6.4: A C pointer Inspector window	96
2.2: A history list in an input box	24	6.5: A C structure or union Inspector window	97
2.3: The active window has a double outline	32	6.6: A C array Inspector window	97
2.4: A typical window	32	6.7: A C function Inspector window	98
2.5: The Pick dialog box	37	6.8: An assembler scalar Inspector window	98
2.6: The Clipboard window	38	6.9: An assembler pointer Inspector window	99
2.7: The normal status line	42	6.10: An assembler array Inspector window	100
2.8: The status line with <i>Alt</i> pressed	42	6.11: An assembler structure Inspector window	100
2.9: The status line with <i>Ctrl</i> pressed	42	7.1: The Breakpoints window	106
3.1: The startup screen showing TDDEMO	44	7.2: The Edit Breakpoint Groups dialog box	108
3.2: The menu bar	45	7.3: The Add Group dialog box	108
3.3: The status line	46	7.4: The Breakpoint Options dialog box	110
3.4: The Module and Watches windows, tiled	47	7.5: The Conditions and Actions dialog box	111
3.5: Program stops on return from function showargs	49	7.6: The Log window	120
3.6: A breakpoint at line 44	50	8.1: The Module window	124
3.7: A C variable in the Watches window	51	8.2: The File window	128
3.8: An Inspector window	52	8.3: The File window showing hex data	128
3.9: Inspecting a structure	53	10.1: The Hierarchy window	149
3.10: The Change dialog box	54	10.2: A class Inspector window	152
3.11: The Evaluate/Modify dialog box	55	10.3: An object Inspector window	155
4.1: The Display Options dialog box	65	11.1: The Windows Messages window for a standard Windows application	158
4.2: The Save Options dialog box	67	11.2: The Add Window dialog box for a standard Windows application	159
5.1: The Variables window	70	11.3: The Windows Messages window with ObjectWindows support enabled	161
5.2: The Local Display dialog box	73	11.4: The Add Window dialog box with ObjectWindows support enabled	162
5.3: The Stack window	74	11.5: The Set Message Filter dialog box	163
5.4: The Get Info text box	75		
5.5: The Execution History window	82		
5.6: The Enter Program Name to Load dialog box	85		
6.1: The Evaluate/Modify dialog box	89		
6.2: The Watches window	92		
6.3: A C scalar Inspector window	95		

11.6: The Windows Information dialog box	166	12.1: The CPU window	178
11.7: The Load Modules or DLLs dialog box	170	12.2: The Dump window	184
		12.3: The Registers window	184
		13.1: The Turbo Debugger menu tree ...	205

Turbo Debugger for Windows (TDW) is a state-of-the-art, source-level debugger designed to work with Turbo C++ for Windows.

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

Here are just some of TDW's features:

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

OOP

New features and changes for version 3.0



For version 3.0, TDW has the following enhancement:

- The Clipboard lets you copy from windows and paste either into text entry boxes on dialog boxes or into other windows. This feature is described on page 36.
- There are new breakpoint features (see Chapter 7) that let you
 - set multiple conditions and actions on a breakpoint
 - set and remove breakpoints in groups

- set and remove breakpoints on all functions or procedures in a module
- set and remove breakpoints on all methods in an object type or all member functions in a class
- C++ templates and nested classes are supported (see page 119).
- International sort orders are supported through the Windows Language setting. You turn this feature on by using the configuration program TDWINST.EXE (see the file TDWINST.DOC).
- The CPU window has a new pane that shows protected mode selectors and lets you look at the contents of memory locations referenced by these selectors (see page 181).
- The device driver TDDEBUG.386 provides support for *Ctrl-Alt-SysRq* program. In addition, this device driver supports the hardware debug registers of the Intel 83086 processor (and higher). See page 12 for TDDEBUG.386 installation information. See page 118 and the online file HDWDEBUG.TDW for information on hardware debugging.
- Debugging of DLLs is faster now that TDW simultaneously loads both the application's symbol table and the symbol table of any DLL you explicitly load or whose code you step into (see page 138).

Hardware and software requirements

TDW has the same hardware and system software requirements as Turbo C++ for Windows.

TDW supports Super VGA video through the use of a DLL named TDVIDEO.DLL. A number of DLLs are distributed with TDW that support different Super VGA cards (described in the README file on your distribution diskettes). To use one of these DLLs with TDW, copy it to the same directory that TDW.EXE is in and name it TDVIDEO.DLL.

See page 5 to find out how to contact Borland.

If you can't find a DLL that matches your Super VGA video adapter, contact Borland Technical Support.



To use TDW, you must have Turbo C++ for Windows. You must already have compiled your source code into an executable (.EXE file) with full debugging information turned on.

- ⇒ When you run TDW, you'll need *both* the .EXE file and the original source files. TDW searches for source files first in the directory where the compiler found them when it compiled, second in the directory specified in the **Options | Path for Source** command, third in the current directory, and fourth in the directory the .EXE file is in.

A note on terminology

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

- Module** Refers to what is usually called a module in C++ and assembler, but also to what is called a *unit* in Pascal.
- Argument** Is used interchangeably with *parameter* in this manual. This applies to references to command-line arguments (or parameters), as well as to arguments (or parameters) passed to functions.

What's in the manual

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

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

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

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

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

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

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

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

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

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

OOP

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

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

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

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

Chapter 14: Debugging a standard C application is an introduction to strategies for effective debugging of your programs.

OOP

Chapter 15: Debugging an ObjectWindows application leads you through a debugging session on a sample Windows program written using the ObjectWindows class library.

Appendix A: Summary of command-line options summarizes all the command-line options described in Chapter 4.

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

How to contact Borland

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

Resources in your package

This product contains many resources to help you:

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

Borland resources

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

*800-822-4269 (voice)
TechFax*

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

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

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

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

Service	Command
CompuServe	GO BORLAND
BIX	JOIN BORLAND
GEnie	BORLAND

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

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

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

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

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

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

Recommended reading

The manuals accompanying your language compiler contain excellent information on programming Windows applications. The Help system also has a complete Windows API reference.

In addition, the following books on programming for Windows might be helpful to you, although they don't take into account the ObjectWindows library or Resource Workshop, both of which make Windows programming much easier than these books indicate:

Microsoft staff. *Microsoft Windows Software Development Kit, Guide to Programming*, Microsoft Corporation. (Redmond, WA: 1990).

Microsoft staff. *Microsoft Windows Software Development Kit Reference, Vols. 1 and 2*, Microsoft Corporation. (Redmond, WA: 1990).

Microsoft staff. *Microsoft Windows Software Development Kit, Tools*, Microsoft Corporation. (Redmond, WA: 1990).

Petzold, Charles. *Programming Windows*, Microsoft Press. (Redmond, WA: 1990).

Getting started

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

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

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

The distribution disks

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

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

Online text files

There are a number of online files the installation program puts on your hard disk. The three you should definitely look at are

README, FILELIST.DOC, and MANUAL.TDW. The first two are accessible on the disk labeled "Installation Disk," and are also copied to your main language directory. The other is in the DOC subdirectory of the main language directory, along with files describing TDW features.

Additional files that provide information not found in the manual are UTILS.TDW (descriptions of utilities), HDWDEBUG.TDW (hardware debugging), ASMDEBUG.TDW (debugging of Assembler programs), and TDWINST.TDW (configuring TDW using TDWINST.TDW).

The README file

You can use the Turbo C++ for Windows editor or the Windows Notepad program to access the README file.

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

The MANUAL.TDW

file

After installation, the \TCW\DOC directory on your hard disk also contains a file called MANUAL.TDW that indicates corrections and additions to the TDW manual. Be sure to consult this file before making extensive use of the manual.

The HELPME!.TDW

file

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

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

The ASMDEBUG.TDW

file

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

The UTILS.TDW file

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

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

- TDWINST.EXE lets you customize TDW. Using this utility, you can permanently set such things as the display options and screen colors.
- TDSTRIP.EXE lets you strip the debugging information (the *symbol table*) from your programs without relinking. You can perform this operation with a .COM file and save the stripped symbol table information in a .TDS file to use in debugging the .COM file.

Use TDSTRIP to prepare .COM files for debugging.

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

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

- TDUMP.EXE displays the contents of object modules and .EXE files in a readable format.



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

```
TDUMP
```

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

```
TDWINST -?
```

Installing TDW

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

Installing TDDEBUG.386

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

The installation program should copy this file to your hard disk and alter your Windows SYSTEM.INI file so that Windows loads TDDEBUG.386 instead of WINDEBUG.386. If the installation program can't complete this task for you, it tells you. You then have to do it by hand, as follows:

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

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

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

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

Hardware debugging

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

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

Where to now?

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

Programmers learning Turbo C++

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

Turbo C++ pros, but Turbo Debugger novices

If you're an experienced Turbo C++ programmer but you're unfamiliar with Turbo Debugger, you can learn about the features of the TDW environment by reading Chapter 2, "TDW basics." If it suits your style, you can then work through the tutorial in Chapter 3, or, if you prefer, move straight on to Chapter 4, "Starting TDW." For a complete rundown of all commands, turn to Chapter 13, "Command reference."

Programmers experienced with Turbo Debugger

If you've used Turbo Debugger in the past, you're probably already familiar with TDW's standard features. In that case, you can go directly to Chapter 11, "Using Windows debugging features," which discusses the features of TDW that support Windows debugging. Another chapter you'll find helpful is Chapter 15, "Debugging an ObjectWindows application," which takes you through a debugging session on a Windows application written using the ObjectWindows library.

TDW basics

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

The debugging process can be broadly divided into four steps:

1. realizing you have an error
2. finding where the error is
3. finding the cause of the error
4. fixing the error

Is there a bug?

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

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

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

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

See Chapter 14 for a more detailed discussion of the debugging process.

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

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

What TDW can do for you

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

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

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

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

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

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

What TDW won't

do

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

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

How TDW does it

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

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

The TDW advantage

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

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

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

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

Menus and dialog boxes

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

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

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

Using the menus

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


Getting in

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




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

Getting around

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

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

Getting out

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

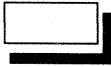
Some menu commands have a shortcut *hot key* that you press to execute them. The hot key appears in the menu to the right of these commands.

Figure 13.1 in Chapter 13 shows the complete pull-down menu tree for TDW. Table 13.1 on page 186 lists all the hot keys. For a summary of all the commands available in TDW, see Chapter 13.

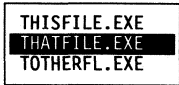
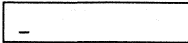
Dialog boxes

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

Table 2.1
What goes in a dialog box



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



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

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



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



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

Knowing where you are

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

To understand this feature, you must realize that first and foremost, TDW is context-sensitive. That means it keeps tabs on

exactly which window you have open, what text is selected, and which subdivision, or *pane*, of the window your cursor is in. In other words, it knows precisely what you're looking at and where the cursor is when you choose a command. And it uses this information when it responds. Let's take an example to illustrate.

Suppose your program has a line like this:

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

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

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

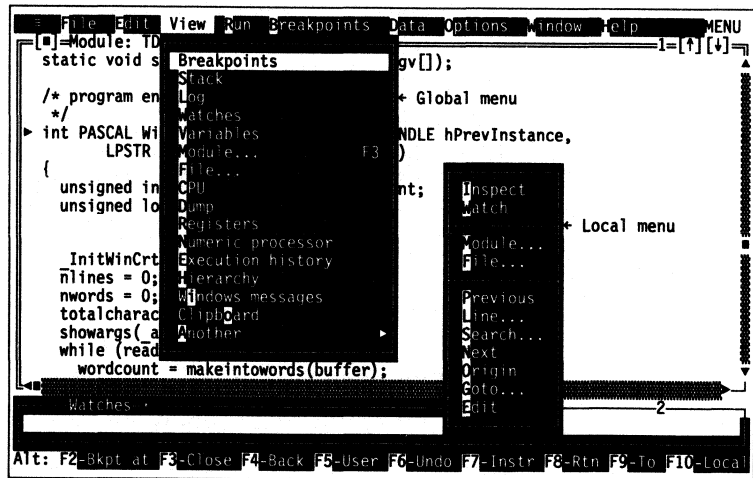
This sort of context-sensitivity makes TDW extremely easy to use. It saves you the trouble of memorizing and typing complicated strings of menu commands or arcane command-line switches. You simply move to the item you want to examine (or select it using the *Ins* key or drag over it with the mouse),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

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

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

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

Figure 2.1
Global and local menus



Compare the following two lists:

Global menus

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

Local menus

- You call up a local menu by pressing *Alt-F10* or by clicking the right button on your mouse.
- The placement and contents of the menu depends on which window or pane you are in and where your cursor is.
- Contents can vary from one local menu to another. (Even so, many of the local commands appear in almost all of the local menus, so that there's a predictable core of commands from one to another.) The *results* of like-named commands can be different, however, depending on the context.
- Every command on a local menu has a hot key shortcut consisting of *Ctrl* plus the highlighted letter in the command.
- Because of this arrangement, a hot key, say *Ctrl-S*, might mean one thing in one context but quite another in a different context. (A core of commands, however, is still consistent across the local menus. For example, the Goto command and the Search

command always do the same thing, even when they are invoked from different panes.)

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

History lessons

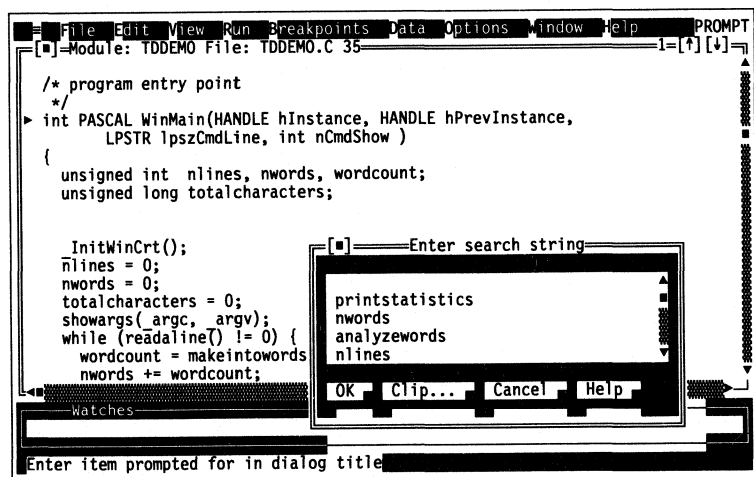
Menus and context-sensitivity comprise just two aspects of the convenient environment of TDW. Another habit-forming feature is the *history list*.

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

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

The search input box might look like this:

Figure 2.2
A history list in an input box



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

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

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

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

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

Automatic name completion

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

Warning!

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



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

Incremental matching

TDW also lets you use *incremental matching* to find entries in a dialog box list of file and directory names. Start typing the name of the file or directory; if the file is available from the names at or below the current position in the list box, the highlight bar moves

to the name as soon as you have typed enough characters to identify it uniquely. Then all you have to do is choose the OK button.

Making macros

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

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

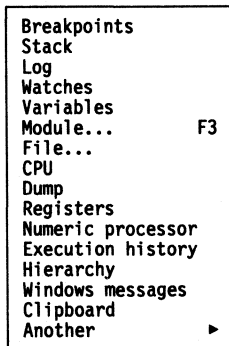
You can assign any series of TDW commands and keystrokes to a single key, for playback whenever you want.

See page 64 in Chapter 4 for an explanation of how to define macros.

Window shopping

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

Windows from the View menu



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

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

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

Breakpoints window

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

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

Stack window

Chapter 5 provides more information on the Stack window.

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

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

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

Log window

Chapter 7 tells you more about the Log window.

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

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

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

Watches window

See Chapter 6 for more about the Watches window.

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

Variables window

Chapter 5 describes the Variables window in more detail.

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

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

Module window

Displays the program code that you're debugging. You can move around inside the module and examine data and code by positioning the cursor on program variable names and issuing the appropriate local menu command.

Chapter 8 details the Module window and its commands.

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

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

File window

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

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

CPU window

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

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

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

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

Dump window

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

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

you modify the displayed data, change the format in which you view the data, and manipulate blocks of data.

Registers window

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

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

Numeric Processor window

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

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

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

Execution History window

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

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

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

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

Hierarchy window

OOP

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

Lists and displays a hierarchy tree of all 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's local menu commands, you can examine any class's data members and member functions.

Windows Messages window

Chapter 11 explains how to use the Windows Messages feature.

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

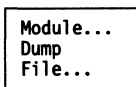
- The left pane shows which procedures or handles you're tracking messages for.
- The right pane shows the type of messages you're tracking.
- The bottom pane displays the messages being tracked.

Clipboard window

See page 36 for an explanation of how to use the Clipboard.

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

Duplicate windows



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

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

User screen

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

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

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

Inspector windows



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

You can open an Inspector window to look at an array of items or at the contents of a variable or expression. The number of panes in the window depends on the nature of the data you are inspecting. An Inspector window adapts to the type of data being displayed. It can display not only simple scalars (**int**, **float**, and so on), but also pointers, arrays, structures, and unions. 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.



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

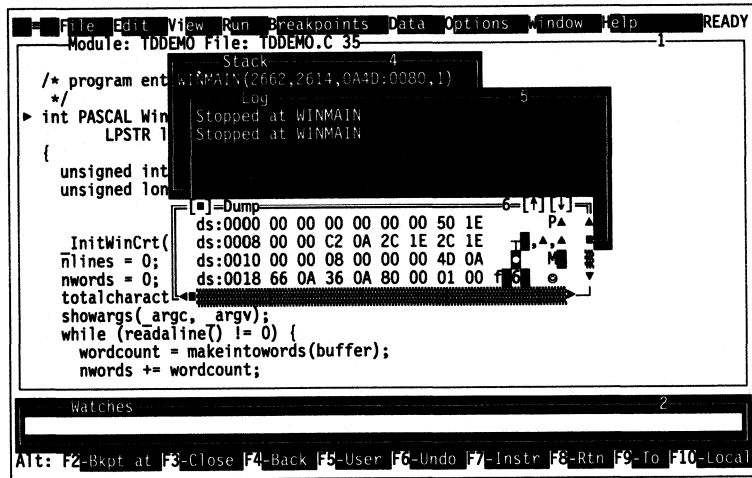
The active window

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

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

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

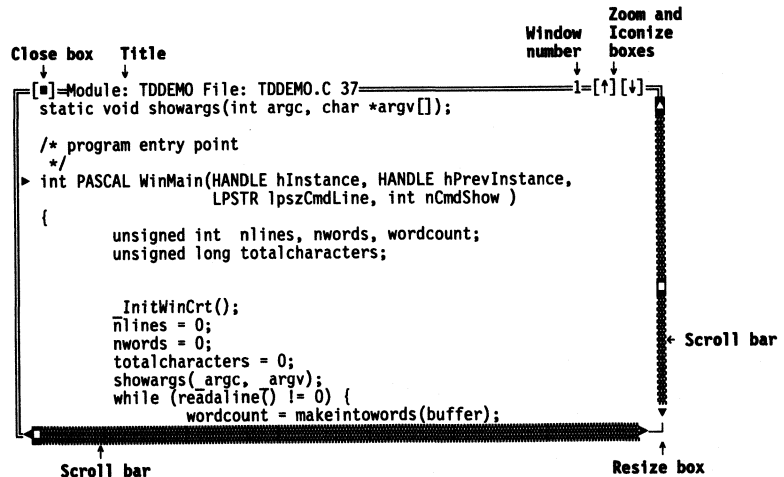
Figure 2.3
The active window has a double outline






What's in a window

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

Figure 2.4
A typical window



- An outline (double if the window is active, single otherwise).
- A title, located at the left top.
- A scroll bar or bars on the right or bottom if the window opens on more information than it can hold at one time. You operate the scroll bars with the mouse:
 - Click the direction arrows at the ends of the bar to move one line or one character in the indicated direction.

- Click the gray area in the middle of the bar to move one window size in the indicated direction.
 - Drag the scroll box to move as much as you want in the direction you want.
-  A resize box in the lower right corner. Drag it with your mouse to make the window larger or smaller. If no scroll bar is present on the bottom or right side of a window, that side of the window border also activates window resizing.
 - A window number in the upper right, reflecting the order in which the window was opened.
 -  A zoom box and iconize box in the upper right corner. The one on the left contains the zoom icon, the one on the right the iconize icon. Click these with your mouse to expand the window to full screen size, restore it to its original size, or iconize it. (When a window is zoomed to full size, only the unzoom box is available, and when it is iconized, only the zoom box is available.)
 -  A close box in the upper left corner. Click it with your mouse to close the window.

Working with windows

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

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

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

Window hopping

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

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

the Watches window active, any commands you choose will affect that window and the items in it.

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

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



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

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

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

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

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



You can also click the pane with the mouse.

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

Refer to Chapter 13 for a table of keystroke commands in panes.

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

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

Once an item is selected (highlighted) from a list, you can press *Alt-F10* to choose a command relevant to it from its local menu. In many lists, you can also just press *Enter* once you have selected an item. This acts as a hot key to one of the commonly used local menu commands. The exact function of the *Enter* key in these cases is described in the reference section starting on page 190.

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

Moving and resizing windows

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

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

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



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

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

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

If you want to enlarge or reduce a window quickly, choose **Window | Zoom**, or click the mouse on the zoom box or the iconize box in the upper right corner.

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

Closing and recovering windows

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

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



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

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

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

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

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

Saving your window layout

Use the **Options | Save Options** command to save a specific window configuration once you have the screen arranged the way you like. In the Save Configuration dialog box, tab to **Layout** and press *Spacebar* to toggle it on. The screen will then appear with your chosen layout each time you start TDW, if the configuration has been saved to a file called **TDCONFIG.TDW**. This configuration file is the only one loaded automatically when TDW is loaded. Other configurations *can* be loaded by using the **Options | Restore Options** command if they have been saved to configuration files with a different name.

Copying and pasting

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

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

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

To copy an item into the Clipboard, position the cursor on the item or highlight it with the *Ins* key, then press *Shift-F3*. To paste something into a window or dialog box from the Clipboard, press

Shift-F4 or use the Clip button in the dialog box to bring up the Pick dialog box.

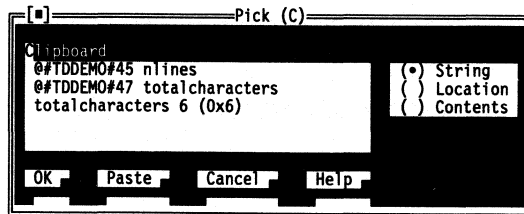


You can paste into any dialog box prompt (any place in a dialog box where you can type text) by pressing *Shift-F4*, even if the dialog box doesn't have a Clip button. You can also paste into dialog box prompts with multiple fields.

The Pick dialog box

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

Figure 2.5
The Pick dialog box



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

For example, if you clip text from the Log window, it can be pasted only as a string. If you clip text from the Module window, it can be pasted elsewhere as a string or as an address, but not as contents. If you clip a variable from an Inspector window, it can be pasted as a string, a location, or as contents (unless it's a C register variable, in which case you can paste it only as a string or as contents, not as an address).

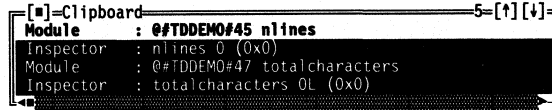
To paste an item into a dialog box, highlight the item, select the appropriate category, then either press *Enter* or the Paste button, depending on what effect you want to have on the dialog box.

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

The Clipboard window

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

Figure 2.6
The Clipboard window



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

Clipboard item types

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

Table 2.2
Clipboard item types

Type	Description
String	A text string, like a marked block from the File window
Module	A module context, including a source code position, like a variable from the Module window
File	A position in a file (in the File window) that isn't a module in the program
CPU code	An address and byte list of executable instructions from the Code pane of the CPU window
CPU data	An address and byte list of data in memory from the Data pane of the CPU window or the Dump window
CPU stack	A source position and stack frame from the Stack pane of the CPU window
CPU register	A register name and value from the Register pane of the CPU window or the Registers window
CPU flag	A CPU flag value from the Flags pane of the CPU window
Inspector	One of the following: A variable name from an Inspector window A constant value from an Inspector or Watches window

Table 2.2: Clipboard item types (continued)

	A register-based variable from an Inspector window
	A bit field from an Inspector window
Address	An address without data or code attached
Expression	An expression from the Watches window
Coprocessor	An 80x87 numeric coprocessor register
Control flag	An 80x87 control flag value
Status flag	An 80x87 status flag value

The Clipboard window local menu

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

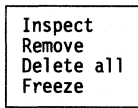


Table 2.3
Clipboard local menu
commands

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

Dynamic updating

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

See Chapter 6 for more information on watching expressions.

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

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

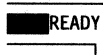
Tips for using the Clipboard

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

- clipping from lines in Module windows as a way of marking locations that you can later return to using the local menu **Goto** command (by pasting a location into the dialog box displayed by the **Goto** command)
- watching an expression (see the previous section)
- pasting new values into variables using the **Data | Evaluate** dialog box or the dialog box for the **Change** command of the Inspector window or the Watches window
- pasting strings into the Log window to help you keep track of what you did during a debugging session
- pasting an address (the *location* category of an item) into any of the places where an address is requested (such as the **Breakpoints | Options** dialog box Address field, or the **Run | Execute To** dialog box)
- pasting expressions into conditions and actions of breakpoints
- pasting parameters into the **Run | Arguments** dialog box
- pasting a Window proc name or an OWL object name into the Windows Messages window
- pasting a string into the dialog box for the Module window Search command
- copying data from and pasting it to the CPU data pane
- copying code from one part of the CPU window to another and then running the program with the copied code

Getting help

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



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

- The active window is always topmost and has a double line around it.



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

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

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

Online help

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



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

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

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

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

the Help system), or choose **Index** from the Help menu. To get help on Help, choose **Help | Help on Help**. To exit from Help, press *Esc*.

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

In a window

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

Figure 2.7
The normal status line

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

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

Figure 2.8
The status line with *Alt* pressed

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

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

Figure 2.9
The status line with *Ctrl* pressed

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

In a menu or dialog box

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

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

A quick example

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

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

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

The demo program

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

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

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

⇒ Make sure your working directory contains the two files needed for the tutorial: TDDEMO.C and TDDEMO.EXE.

Getting In To start the demo program,

1. Make sure Windows is running in standard or 386 enhanced mode. (TDW doesn't run in real mode.)
2. In the Windows Program Manager, open the program group that contains TDW and choose the TDW icon.
3. When TDW starts up, choose File | Open and enter the full path to TDDEMO (you might need to compile TDDEMO.C first with debugging information), then press *Enter*.

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

Figure 3.1
The startup screen showing
TDDEMO

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

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

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

watches ?

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

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

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

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

F1

Using TDW

The menus

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

Figure 3.2
The menu bar



File Edit View Run Breakpoints Data Options Window Help READY

F10

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

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



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

Esc

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

The status line

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

Figure 3.3
The status line



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

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

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

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

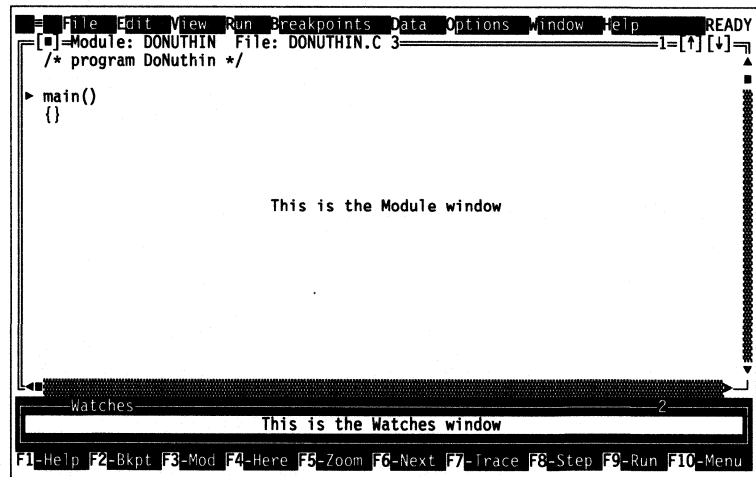


The windows

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

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

Figure 3.4
The Module and Watches
windows, filled



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

F6

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

Alt F3

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

Alt F6

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

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

Choose **Window | Size/Move** and use the arrow keys to reposition the active window (the Stack window) on the screen. Next, hold **Shift** down and use the arrow keys to adjust the size of the

window. Press *Enter* when you have defined a new size and position that you like.

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

Using the C demo program

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

F7

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

To position the cursor on a line in the Module window, press Ctrl-G, type the line number, and press Enter.

As you can see from the **Run** menu, there are a number of ways to control the execution of your program. Let's say you want to execute the program until it reaches line 48.

F4

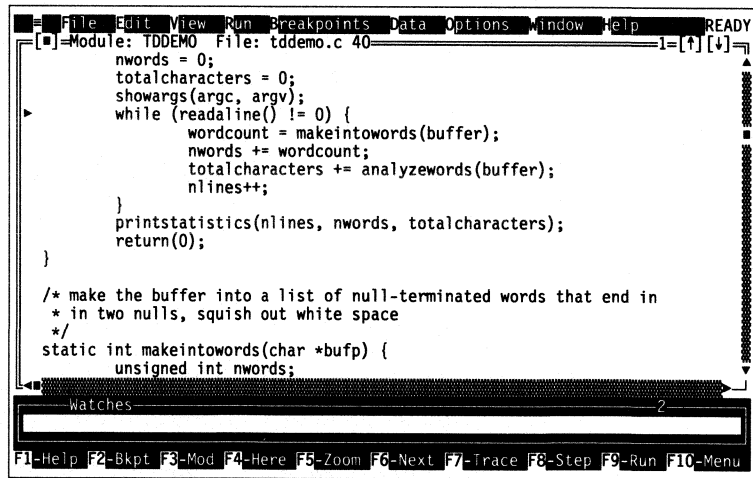
First, position the cursor on line 39, then press *F4* to run the program up to (but not including) line 39. Now press *F7*, which executes one line of source code at a time; in this case, it executes line 39, a call to the function **showargs**. The cursor immediately jumps to line 151, where the definition of **showargs** is found.

Alt F8

Continuing to press *F7* would step you through the function **showargs** and then return you to the line following the call—line 40. Instead, press *Alt-F8*, which causes **showargs** to execute and then return, at which point the program stops. This command too, returns you to line 40, and is very useful when you want to jump past the end of a function.

If you had pressed *F8* instead of *F7* on line 39, the cursor would have gone directly to line 40 instead of into the function. *F8* is similar to *F7* in that it executes a procedure or source line, but it skips any function calls.

Figure 3.5
Program stops on return from
function showargs



```
File Edit View Run Breakpoints Data Options Window Help READY
Module: TDDemo File: tddemo.c 40
nwords = 0;
totalcharacters = 0;
showargs(argc, argv);
while (readline() != 0) {
    wordcount = makeintowords(buffer);
    nwords += wordcount;
    totalcharacters += analyzewords(buffer);
    nlines++;
}
printstatistics(nlines, nwords, totalcharacters);
return(0);
}

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

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

To execute the program until a specific place is reached, you can directly name the function or line number, without moving the cursor to that line in a source file and then running to that point.

Alt F9

Press *Alt-F9* to specify a label to run to. A dialog box appears. Type *readline* and press *Enter*. The program runs, then stops at the beginning of function **readline** (line 142).

Setting breakpoints

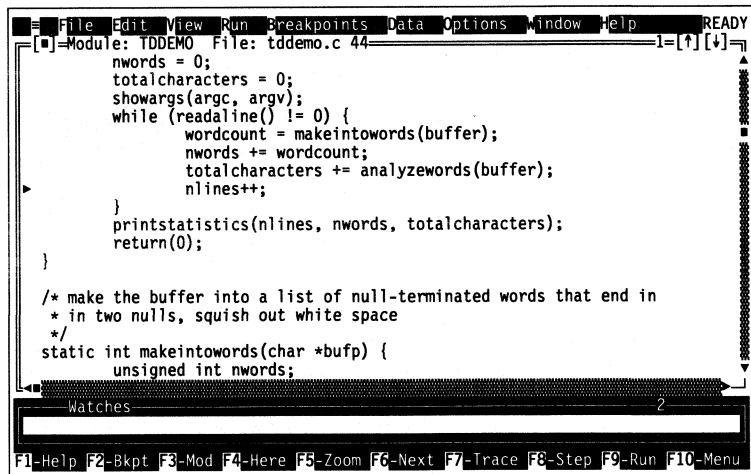
F2

Another way to control where your program stops running is to set breakpoints. The simplest way to set a breakpoint is with the *F2* key. Move the cursor to line 44 and press *F2*. TDW highlights the line, indicating there is a breakpoint set on it.



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

Figure 3.6
A breakpoint at line 44



F9 Now press *F9* to execute your program without interruption. The screen switches to the program's display. The demo program is now running and waiting for you to enter a line of text. Type *abc*, a space, *def*, and then press *Enter*. The display returns to the TDW screen with the arrow on line 44, where you set a breakpoint that has stopped the program. Now press *F2* again to toggle it off.

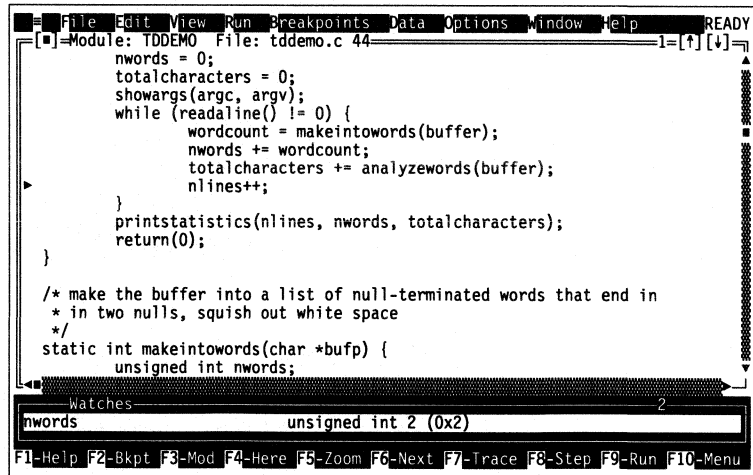
See Chapter 7 for a complete description of breakpoints, including conditional and global breakpoints.

Using watches

Alt F10

The Watches window at the bottom of the screen shows the value of variables you specify. For example, to watch the value of the variable *nwords*, move the cursor to the variable name on line 42 and choose **Watch** from the Module window local menu (bring it up with *Alt-F10* or the right-hand mouse button, or use the shortcut *Ctrl-W*).

Figure 3.7
A C variable in the Watches window



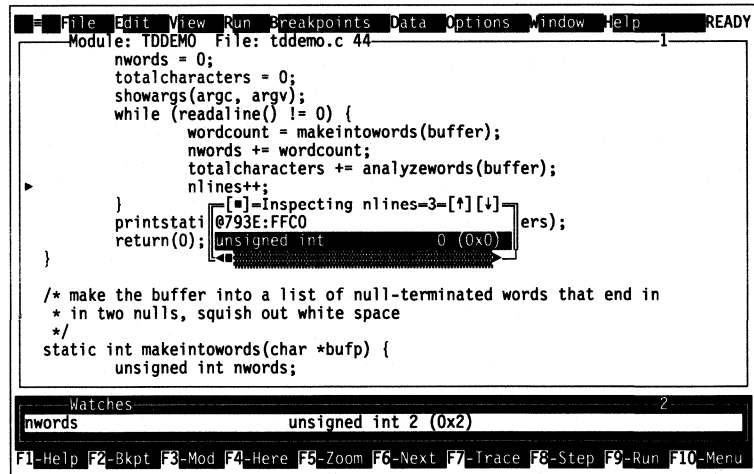
nwords now appears in the Watches window at the bottom of the screen, along with its type (**unsigned int**) and value. As you execute the program, TDW updates this value to reflect the variable's current value.

Examining simple C data objects

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

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

Figure 3.8
An Inspector window



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

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

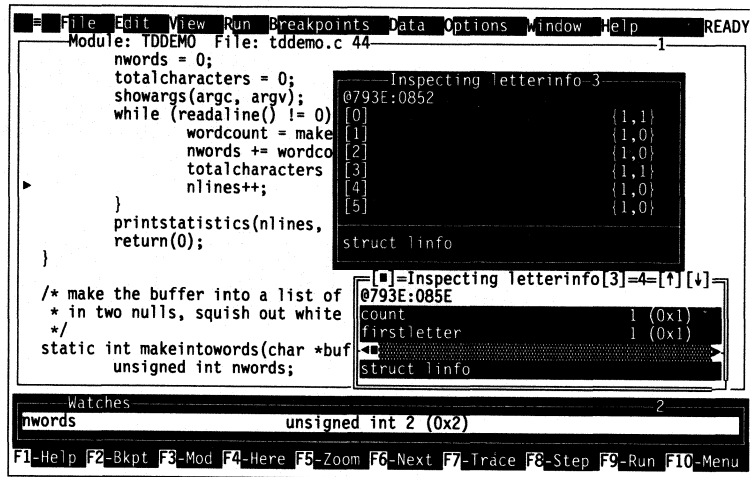
Let's review what you actually did here. By pressing `Ctrl`, you took a shortcut to the local menu commands in the Module window. Pressing `I` specified the **Inspect** command.

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

Examining compound C data objects

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

Figure 3.9
Inspecting a structure



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

Alt **F3**

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

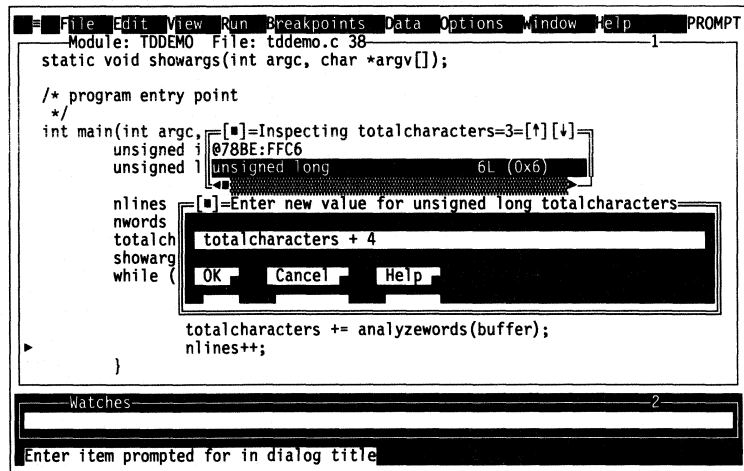
Changing C data values

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

Use the arrow keys to go to line 38 in the source file. Place the cursor at the variable *totalcharacters* and press *Ctrl-I* to inspect its

value. With the Inspector window open, press *Alt-F10* to bring up the Inspector's local menu, and choose the **Change** option. (You could also have done this directly by pressing *Ctrl-C*.) A dialog box appears, asking for the new value.

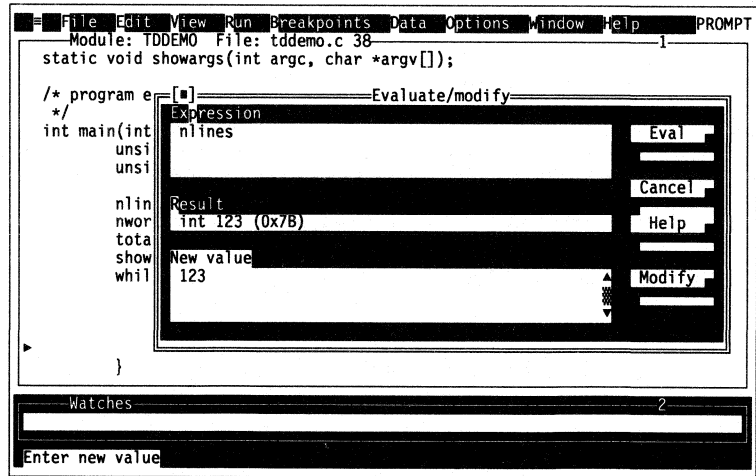
Figure 3.10
The Change dialog box



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

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

Figure 3.11
The Evaluate/Modify dialog
box



That's a quick introduction to using TDW with a program written using Turbo C++ for Windows. Chapter 14 offers a more extensive debugging sample.

Starting TDW

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

Preparing programs for debugging

When you compile and link with Turbo C++ for Windows, you can tell the compiler to generate full debugging information. If you have compiled your program's object modules without any debugging information, be sure to recompile them with debugging information before invoking TDW.

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

If you're using the integrated environment of Turbo C++ for Windows, the generation of debug information is turned on by default. If these options have been turned off, you need to do the following before compiling to produce debug information:

1. Use the Options | Linker | Settings command to bring up the Linker Settings window, then check the Include Debug Information check box.
2. Use the Options | Compiler | Advanced Code Generation command to bring up the Advanced Code Generation window, then check the Debug Info in OBJs check box.
Alternatively, you can use the **options -v** pragma directive to add debug information to each of your modules by inserting the following line at the beginning of each module:

```
#pragma option -v
```

Starting TDW

There are four ways to run TDW:

- If you are in Turbo C++ for Windows, you can debug the program in the active window by choosing Run | Debugger. You can also choose Run | Debugger Arguments if you want to set TDW command-line arguments.
- If you are in Windows, the easiest method is to open the appropriate program group in the Windows Program Manager and choose the TDW icon. Then choose File | Open to load the program you're debugging.

Warning!

For this and the next option, unless TDW is in your path and your program is in your Windows directory, you must be careful to type in the correct path for both TDW and your application.

- If you are in Windows and you want to enter command-line options, you can start TDW by using the Windows Program Manager **File | Run** command to open the Run dialog box. Then, in the Command Line input box, just type `TDW`, followed by any command-line options and, optionally, the name of the program you're debugging, as if you were at the DOS prompt.
- If you are at the DOS prompt, you can start TDW by entering the following and pressing *Enter*:

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

Entering command-line options

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

If you start TDW from Turbo C++ for Windows, you can enter command-line options by choosing **Run | Debugger Arguments**.

Directly entering command-line options

The generic command-line format is

```
TDW [options] [programe [progargs]]
```

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

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

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

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

Entering command-line options from TCW

If you start TDW from Turbo C++ for Windows and you want to indicate command-line options, chose **Run | Debugger Arguments** to enter any TDW command-line options except the program name and program arguments.

The program to be run in TDW is the one in the current Edit window. If you want to enter arguments for the program before running TDW on it, choose **Run | Arguments** and type the arguments in the Program Arguments window.

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

- ⇒ You must already have compiled your source code into an executable (.EXE) file with full debugging information turned on before debugging with TDW.
- ⇒ TDW works only with Windows programs compiled with a Borland compiler.
- ⇒ If you're running your program from Windows and notice a bug, you have to exit your program and load it under TDW before you can begin debugging.

Running TDW

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

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

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

Command-line options

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

All TDW command-line options start with a hyphen (-) and are separated from the TDW command and each other by at least one space. You can explicitly turn a command-line option off by following the option with another hyphen. For example, **-p-**

disables the mouse. Turning a command-line option off works even if an option has been permanently enabled in the configuration file. You can modify the configuration file by using the TDWINST configuration program described in the file TDWINST.DOC.

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

Loading the configuration file (-c)

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

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

```
TDW -cMYCFG.TDW TDDEMO
```

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

Display updating (-d)

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

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

Getting help (-h and -?)

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

Assembler-mode startup (-l)

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

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

Mouse support

(-p)

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



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

Source code handling (-s)

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

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

This option doesn't change the starting directory.

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

`-sdirname[;dirname...]`

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

Starting directory

(-t)

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

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

-tdirname

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

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

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

Configuration files

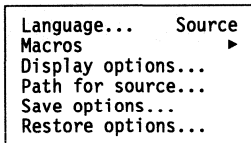
See the file TDWINST.TDW for a description of how to use TDWINST to create configuration files.

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

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

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

The Options menu

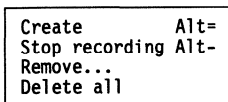


The **O**ptions menu lets you set or adjust a number of parameters that control the overall appearance and operation of TDW. The following sections describe each menu command and refer you to other sections of the manual where you can find more details.

The Language command

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

The Macros menu



The **M**acros command displays another menu that lets you define new keystroke macros or delete ones that you have already assigned to a key. It has the following commands: **C**reate, **S**top Recording, **R**emove, and **D**elete All.

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

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

Stop Recording The **S**top 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 **O**ptions | **M**acro | **S**top Recording menu selection to stop recording your macro, as these keystrokes will then be added to your macro! (The menu item is added to remind you of the *Alt-* hot key.)

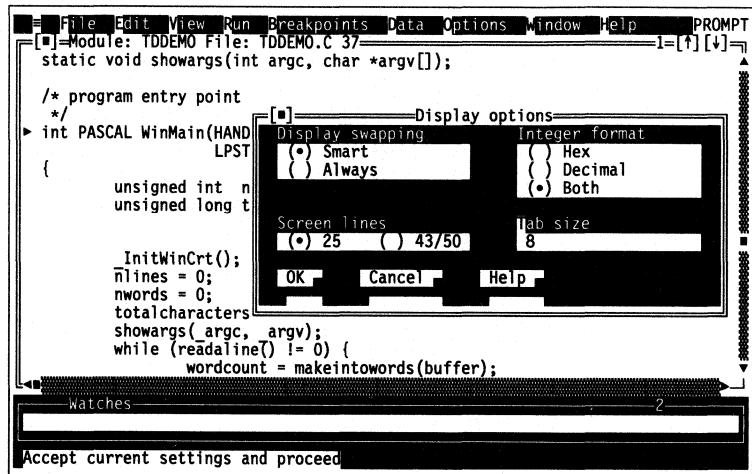
Remove Displays a dialog box listing all current macros. To delete a macro, select one from the list and press *Enter*

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

Display Options command

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

Figure 4.1
The Display Options dialog
box



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

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

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

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

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

Decimal Shows integers as ordinary decimal numbers.

Both Shows integers as both decimal numbers and as hex numbers in parentheses after the decimal value.

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

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

**Path for Source
command**

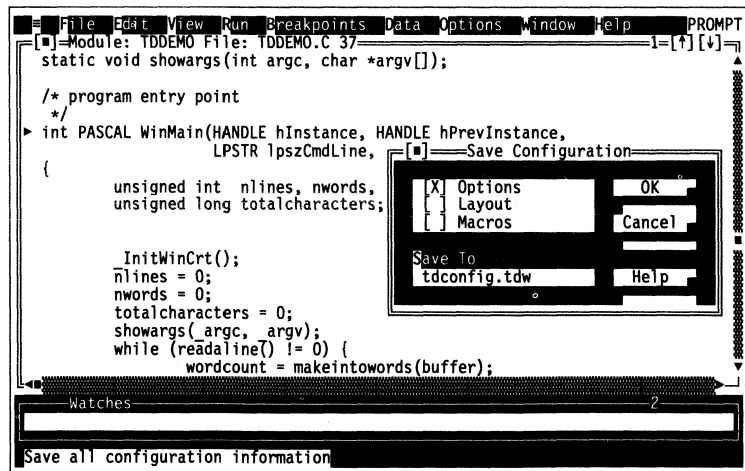
Sets the directories that TDW searches for your source files. See the discussion of the Module window in Chapter 8 for more information.

**Save Options
command**

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

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

Figure 4.2
The Save Options dialog box



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

Options Saves all settings made in the **Options** menu.

Layout Saves only the windowing layout.

Macros Saves only the currently defined macros.

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

Restore Options command

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

Returning to Windows

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

Controlling program execution

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

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

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

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

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

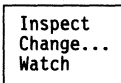
Both panes show the name of the variable at the left margin and its value at the right margin. If TDW can't find any data type information for the symbol, it displays four question marks (????).

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

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

The Global pane local menu

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



Inspect

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

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

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

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

Change

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

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

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

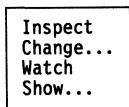
Watch

See Chapter 6 for more information on the Watches window.

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

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

The Static pane local menu



Inspect
Change...
Watch
Show...

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

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

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

Inspect

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

Change

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

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

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

Watch

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

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

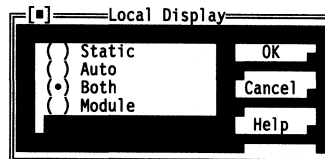
Show

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

The following radio buttons appear in this dialog box:

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

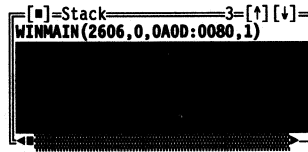
Figure 5.2
The Local Display dialog box



The Stack window

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

Figure 5.3
The Stack window



OOP

The Stack window likewise displays the names of member functions, each of which is prefixed with the name of the class that defines the member function:

```
SHAPES::ACIRCLE(174, 360, 75.0) /* C++ */
```

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

The Stack window local menu

Inspect
Locals

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

Inspect

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

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

Locals

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

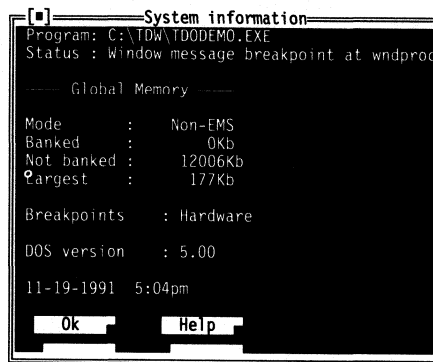
The Origin local menu command

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

The Get Info command

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

Figure 5.4
The Get Info text box



The following information appears in the System Information box:

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

Global memory information

TDW provides you with the following information about global memory:

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

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

Status line messages

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

Stopped at __

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

No program loaded

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

Trace

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

Step

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

Breakpoint at __

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

Window message breakpoint at __

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

Terminated, exit code __

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

Loaded

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

Interrupt

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

Exception __

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

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

Divide by zero

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

Global breakpoint __ at __

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

The Run menu

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

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

Run

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

F9

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

Go to Cursor

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

F4

Trace Into

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

F7

OOP

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

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

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

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

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

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

OOB

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 dialog box is reached. The address you specify might never be reached if a breakpoint action is encountered first or you interrupt execution.

Until Return

Alt F8

Executes until the current function returns to its caller. This is useful in two circumstances: When you have accidentally executed into a function you aren't interested in with **Run | Trace** instead of **Run | Step**, or when you've determined that the current

procedure works to your satisfaction, and you don't want to slowly step through the rest of it.

Animate

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

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

Back Trace

Alt **F4**

Some restrictions apply to using the Execution History window. See page 82 for more information.

If you are tracing (**F7** or **Alt-F7**) through your program, Back Trace reverses the order of execution. Reverse execution is handy if you trace beyond the point where you think there may be a bug, and want to reverse program execution back to that point. This feature lets you “undo” the execution of your program by stepping backward through the code, either a single step at a time or to a specified point highlighted in the Execution History window.



Reverse execution is always available in the CPU window. However, you can only execute source code in reverse if full history is *On*. (Use the **View | Execution History** command to bring up the Execution History window, then in the local menu set **Full History On**.)



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

Instruction Trace

Alt **F7**

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

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

Arguments

This command lets you set new command-line arguments for your program. For a discussion of this command, see “Changing the program arguments” on page 86.

Program Reset

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

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

Ctrl **F2**

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

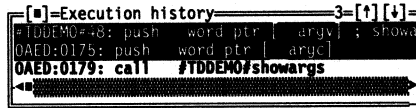
If you chose **Program Reset** because you just executed one source statement more than you intended, you can position the cursor up a few lines in your source file and press *F4* to run to that location. Alternatively, if Full History had been on (see the local menu of the **View | Execution History**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

The Execution History window

TDW has a special feature called the *execution history* that keeps track of each instruction as it’s executed (provided that you’re tracing into the code). You can examine these instructions and, if you wish, undo them to return to a point in the program where

you think there might be a bug. TDW can record about 400 instructions.

Figure 5.5
The Execution History window



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

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

- ➞ The execution history only keeps track of instructions that have been executed with the **Trace Into** command (*F7*) or the **Instruction Trace** command (*Alt-F7*). It also tracks for **Step Over**, as long as you don't encounter one of the commands listed on page 79 or 83. As soon as you use the **Run** command or execute an interrupt, the execution history is deleted. (It starts being recorded again as soon as you go back to tracing.)
- ➞ You cannot backtrace into an interrupt call.
- ➞ If you step over a function call, you will not be able to trace back beyond the instruction following the return.
- ➞ Backtracing through a port-related instruction has no effect, since you can't undo reads and writes.

The local menu

Inspect	
Reverse execute	
Full history	No

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

Inspect

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

Reverse Execute

Alt F4

This command reverses program execution to the location highlighted in the Instructions pane. If you selected a line of

source code, you are returned to the Module window; otherwise, the CPU window appears with the highlight bar of the Code pane on the instruction.

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

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

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

IN	INSW
OUT	OUTSB
INSB	OUTSW

Full History

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

Interrupting program execution

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

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

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

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



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

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

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

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

Program termination

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

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

Restarting a debugging session

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

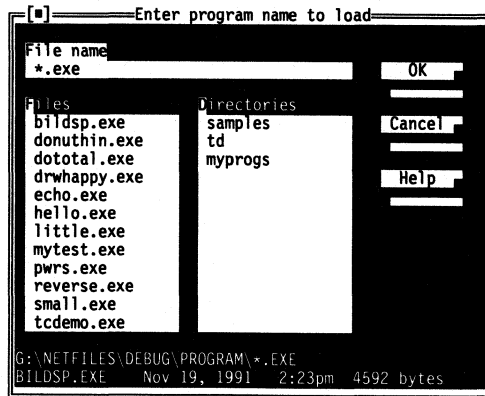
To reload the program you were debugging, choose **Run | Program Reset (Ctrl-F2)**. TDW reloads the program from disk, with any data you have added since you last saved to disk. Reloading is the safest way to restart a program. Restarting by executing at the start of the program can be risky, since many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

➡ **Program Reset** leaves breakpoints and watchpoints intact.

Opening a new program to debug

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

Figure 5.6
The Enter Program Name to Load dialog box



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

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

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

```
myprog a b c
```

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

Changing the program arguments

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

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

Examining and modifying data

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

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

This chapter assumes that you understand the various data types that can be used in Turbo C++ for Windows. If you are fairly new to the language and have not yet explored all its data types (**char**, **int**, **short**, **long**, **unsigned**, **float**, **double**, and so on), this chapter can give you valuable information about them. When you have delved into the more complex data types (arrays, pointers, structures, files, classes, and so on), return to this chapter to learn more about looking at them with TDW.

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

This chapter shows you how to examine and modify variables in your program. First, we explain the **Data** menu and its options. We then discuss how you can modify program data by evaluating expressions that have side effects, and show you how to point directly at data items in your source modules. Finally, we

introduce the Watches window and describe the way that the data types of each language appear in Inspector windows.

The Data menu

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

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

Inspect

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

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

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



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

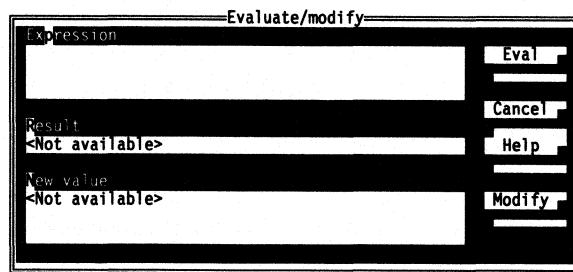
Evaluate/Modify

See Chapter 9 for a complete discussion of expressions.

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

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

Figure 6.1
The Evaluate/Modify dialog
box



See Chapter 9 for a
discussion of format control.

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

The dialog box has three fields.

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

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



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

OOP

If you're debugging a C++ program, the Evaluate/Modify dialog box also lets you display the members of a class instance. You can use any format specifier with an instance that can be used in evaluating a record.

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

You can't execute constructors or destructors in the Evaluate window.

Turbo Debugger also lets you call a member function from inside the Evaluate/Modify dialog box. Just 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). With these declarations,

```
class point {
public:
    int x, y, visible;

    point ();
    ~point();
    int Show();
    int Hide();
    void MoveTo(int NewX, int NewY);
};

point APoint;
```

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

Expression	Result
<i>APoint.x</i>	int 2 (0x2)
<i>APoint</i>	class point {1,2,27489}
<i>APoint.MoveTo</i>	void () @6B61:0299
<i>APoint.Show</i>	int () @6B61:0285
<i>APoint.Show()</i>	int 1 (0x1)

Expressions with side effects

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

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

If you don't intend to modify the value of any variable but merely want to evaluate an expression containing some of your program variables, don't use any of the operators that have side effects. On the other hand, side effects can be a quick and easy way to change

the value of a variable or memory area. For example, to add 1 to the value of your variable named *count*, evaluate the C expression *count++*.

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

Add Watch

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

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

Function Return

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

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

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

Pointing at data objects in source files

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

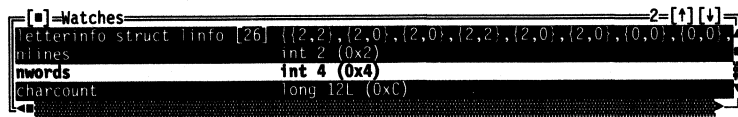
TDW has a powerful mechanism to relieve you from always typing in the names of program variables that you want to inspect. From within any Module window, you can place the cursor anywhere within a variable name and use the local menu Inspect command to create an Inspector window showing the contents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

variable to inspect by pressing *Ins* and using the cursor keys to highlight it before choosing *Inspect*.

The Watches window

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

Figure 6.2
The Watches window



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

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

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

Warning!

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

Unless you use the scope-overriding mechanism discussed in Chapter 9, TDW evaluates expressions in the Watches window in the scope of the current location where your program is stopped. Hence an expression in the Watches window is evaluated as if it appeared in your program at the place where the program is stopped. If a watch expression contains a variable name that is not accessible from the current scope—for example, if it's private to

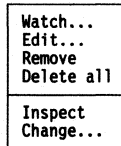
another module—the value of the expression is undefined and is displayed as four question marks (????).

OOP

When you're tracing inside a member function, you can add the **this** parameter to the Watches window.

The Watches window local menu

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



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

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

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

Remove Removes the currently selected item from the Watches window.

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

Inspect Opens an Inspector window to show you the contents of the currently highlighted item in the Watches window. If the item is a compound object (array, **class**, or **struct**), you can view all its elements, not just the ones that fit in the Watches window. (The next section, "Inspector Windows," explains all about Inspector windows.)

Change

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

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

Inspector windows

An Inspector window displays your program data appropriately, depending on the data type you're inspecting. Inspector windows behave differently for scalars (for example, **char** or **int**), pointers (**char ***), arrays (**long x[4]**), functions, and structures.

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

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

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

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

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

C data Inspector windows

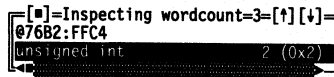
Scalars Scalar Inspector windows show you the value of simple data items, such as

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

Following the top line, these Inspector windows have only a single line of information that gives the address of the variable. To the left on the following line appears the type of the scalar variable (**char**, **unsigned long**, and so forth), and to the right appears its present value. The value can be displayed as decimal, hex, or both. It's usually displayed first in decimal, with the hex values in parentheses (using the standard C hex prefix of 0x). Use TDWINST to change how the value is displayed.

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

Figure 6.3
A C scalar Inspector window



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

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

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

If the pointer is of type **char** and appears to be pointing to a null-terminated character string, more information appears, showing the value of each item in the character array. To the left in each

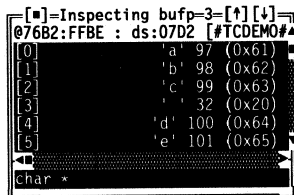
line appears the array index ([1], [2], and so on), and the value appears to the right as it would in a scalar Inspector window. In this case, the entire string is also displayed on the top line, along with the address of the pointer variable and the address of the string that it points to.

You also get multiple lines if you open the Inspector window and then use the **Range** local menu command. This is an important technique for C programmers who use pointers to point to arrays of items as well as single items. For example, if you had the code

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

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

Figure 6.4
A C pointer Inspector
window



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

Structures and unions

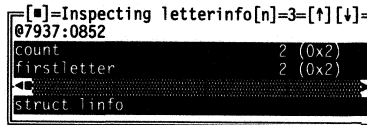
Structure and union Inspector windows show you the value of the members in your structure and union data items. For example,

```
struct linfo {
    unsigned int count;
    unsigned int firstletter;
} letterinfo [26];

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

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

Figure 6.5
A C structure or union
Inspector window



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

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

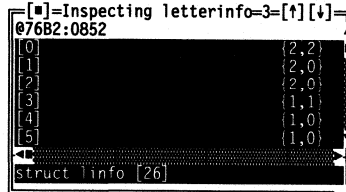
```

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

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

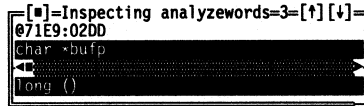
You can use the **Range** local menu command to examine any portion of an array. This is useful if the array has a lot of elements, and you want to look at something in the middle of the array.

Figure 6.6
A C array Inspector window



Functions Function Inspector windows show each parameter with which a function is called. The parameters are displayed below the memory address at the top of the window.

Figure 6.7
A C function Inspector
window



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

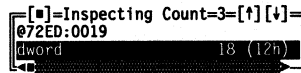
Assembler data Inspector windows

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

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

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

Figure 6.8
An assembler scalar
Inspector window



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

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

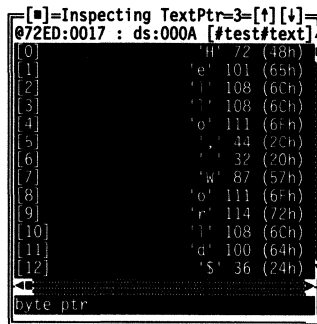
Pointer Inspector windows usually have only a single line of information following the top line that gives the address of the variable. To the left appears [0], indicating the first member of an array. To the right appears the value of the item being pointed to. If the value is a complex data item such as an array, however, only

as much of it as possible is displayed, with the values enclosed in braces ({}).

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

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

Figure 6.9
An assembler pointer
Inspector window



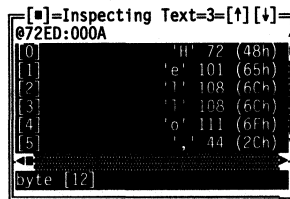
Arrays Array Inspector windows in assembler programs show you the value of arrays of data items, such as

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

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

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

Figure 6.10
An assembler array inspector window



Structures and unions

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

```

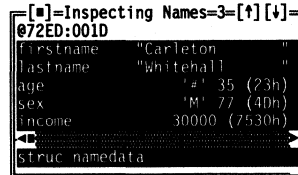
X      STRUC
MEM1      DB      ?
MEM2      DD      ?
X      ENDS
ANX      X      <1,ANX>

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

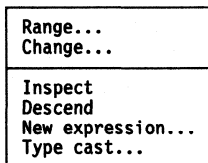
```

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

Figure 6.11
An assembler structure inspector window



The Inspector window local menu



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

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

Range

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

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

Change

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

Inspect

Opens a new Inspector window that shows you the contents of the currently highlighted item. This is useful if an item in the Inspector window contains more items itself (like a structure or array), and you want to see each of those items.

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

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

Descend

This command works like the **Inspect** local menu command except that instead of opening a new Inspector window to show the contents of the highlighted item, it puts the new item in the

current Inspector window. This is like a hybrid of the **New Expression** and **Inspect** commands.



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

New Expression

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

OOP

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

Type Cast

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

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

Breakpoints

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

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

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

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

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

The *condition* can be

- always
- when an expression is true

See Chapter 11, page 164 for a description of message breakpoints.

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

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

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

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

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

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



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

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

The Breakpoints menu

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

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

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

At *See page 109 for a description of the Breakpoint Options dialog box.* At lets you set a breakpoint at a specific location in your program. When selected, **At** opens the Breakpoint Options dialog box, from which you can set all breakpoint options. *Alt-F2* is the hot key for **At**.

Changed memory global *For more information, see the "Changed Memory" section on page 117.* Changed Memory Global sets a global breakpoint that's triggered when an area of memory changes value. You are prompted for the area of memory to watch with the Enter Memory Address, Count input box. The variable expression entered is checked for change each time a line of source code is executed.

Expression true global *For more information, see "Conditional Breakpoints" on page 118.* Expression True Global sets a global breakpoint that is triggered when the value of a supplied expression is true (nonzero). You are prompted for the expression to evaluate with the Enter Expression for Condition Breakpoint input box. The expression entered is evaluated each time a line of source code is executed.

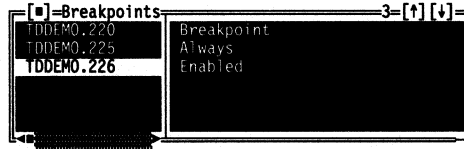
Hardware breakpoint *For more information on hardware debugging, see page 118.* Use this command to access the Hardware Breakpoints Options dialog box. You must have the proper system setup in order to use hardware debugging.

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

The Breakpoints window

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

Figure 7.1
The Breakpoints window

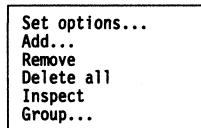


The Breakpoints window has two panes; the Breakpoint List (left pane) shows a list of all the addresses at which breakpoints are set and the Breakpoint Detail (right pane) shows the details of the breakpoint highlighted in the left pane. Although a breakpoint can have several sets of actions and conditions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

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

The Breakpoints window local menu

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



Set Options

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

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

Add

The **Add** command on the Breakpoints local menu opens the Breakpoint Options dialog box, much like the **Set Options** command does. The difference is that the cursor is positioned on an empty **Address** text box. Enter into the **Address** text box the address for which you'd like the breakpoint to be set. For

For a detailed explanation of the Breakpoint Options dialog box, see page 109.

example, if you'd like to set a breakpoint at line number 3201 in your C source code, enter #3201 into the text box. If the line of code is in a module not displayed in the Module window, type a pound sign (#), followed by the module name, followed by another pound sign, and then the line number. For example:
#OTHERMOD#3201.

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

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

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

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

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

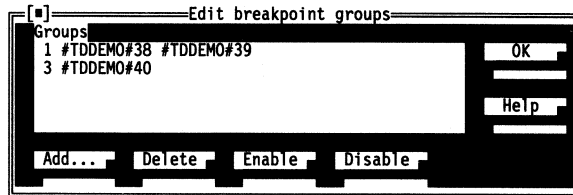
You can also invoke this command by pressing *Enter* once you have the highlight bar positioned over a breakpoint.

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

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

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

Figure 7.2
The Edit Breakpoint Groups dialog box



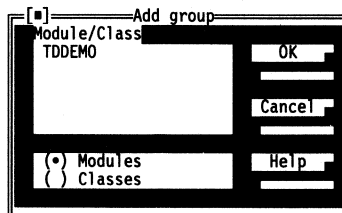
Groups

The **Groups** list box displays the currently assigned breakpoint groups.

Add

The **Add** button activates the Add Group dialog box.

Figure 7.3
The Add Group dialog box



The Add Group dialog box has a single list box and a single set of radio buttons that allow you to add all 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 program loaded into the Module window. Highlight the desired module or class, then press **OK**

to set breakpoints on all functions in the module or class. All breakpoints set are collected into a single breakpoint group.

- Two radio buttons allow you to select the type of functions that are displayed in the Module/Class list box:
 - The *Modules* radio button selects all modules contained in the current program, displaying them in the Module/Class list box.
 - The *Classes* radio button selects all the C++ classes contained in the current program for display in the Module/Class list box.

Delete

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

Enable

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

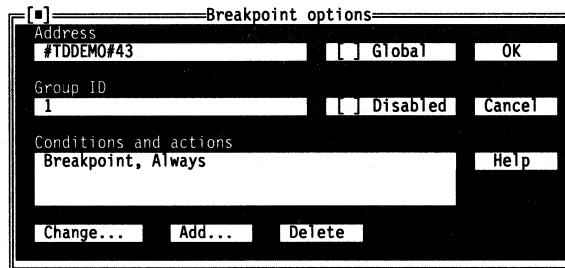
Disable

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

The Breakpoint Options dialog box

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

Figure 7.4
The Breakpoint Options dialog box



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

Group ID The **Group ID** text box allows you to assign the current breakpoint to a new or existing group. A breakpoint *group* is identified by a unique positive integer.
See page 107 for a description of breakpoint groups.

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

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

For more information on global breakpoints, see page 116.

Disabled The **Disabled** check box turns off the current breakpoint. While this command is similar to the **Toggle** command on the **Breakpoints** menu (see page 105), **Disable** does not clear the breakpoint of its settings (as does the **Toggle** command). **Disable** simply masks the breakpoint until you want to reenabling it by unchecking this box. When the breakpoint is reenabling, all settings previously made to the breakpoint will become effective.

This check box is useful if you have defined a complex breakpoint that you don't want to use just now, but will want to use again

later. It saves you from having to delete the breakpoint, and then reenter it along with its complex conditions and actions.

Conditions and Actions

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

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

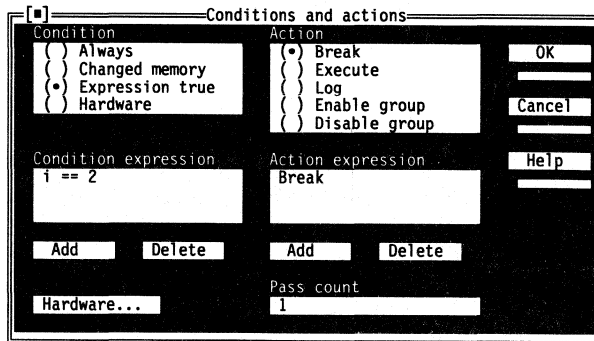
Add To add a new set of conditions and actions to the current breakpoint, select **Add**. Like the **Change** command above, **Add** opens the **Conditions and Actions** dialog box.

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

The Conditions and Actions dialog box

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

Figure 7.5
The **Conditions and Actions** dialog box



See “*Customizing breakpoints*” on page 116 for details on modifying breakpoints.

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

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

The condition radio buttons

The Condition radio buttons have four settings:

Always

When **Always** is chosen, it indicates that no additional conditions need be true for the breakpoint to trigger; it will be triggered each time program execution encounters the breakpoint.

Changed memory

See page 117 for more information on Changed Memory breakpoints.

A **Changed Memory** breakpoint watches a memory variable or object; the breakpoint is triggered if the object changes value. Use the **Condition Expression** input box to enter an expression representing the data object you want to watch.

Expression true

See page 118 for details on expressions.

The **Expression True** button allows the breakpoint to be triggered when an expression becomes true (nonzero). Use the **Condition Expression** input box to enter an expression that's evaluated each time the breakpoint is encountered.

Hardware

See page 118 for more information about hardware breakpoints.

Causes the breakpoint to be triggered by the hardware-assisted device driver. Because you can use hardware assistance only with a global breakpoint, you must check the **Global** check box in the Breakpoint Options dialog box before you can access this option.

The Hardware | Breakpoint command offers an easy way to set hardware breakpoints.

You must select the **Hardware** radio button before the **Hardware** button at the bottom of the dialog box can become active. Pushing that button displays the Hardware Breakpoint Options dialog box. The choices you can make in this box are described in the online text file HDWDEBUG.TD.

The action radio buttons

The Action radio buttons have five settings:

Break

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

Execute

Execute causes an expression to be executed. Enter the expression in the Action Expression input box. The expression should have some side effect, such as setting a variable to a value. By executing an expression that has side effects each time a breakpoint is triggered, you can effectively "splice in" new pieces of code before a given source line. This is useful when you want to alter the behavior of a routine to test a diagnosis or bug fix. This saves you from going through the compile-and-link cycle just to test a minor change to a routine.

Of course, this technique is limited to the insertion of an expression before an already existing line of code is executed; you can't use this technique to modify existing source lines directly.

Log

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

The **Log** button causes the value of an expression to be recorded in the Log window. You are prompted for the expression whose value you want to log. Be careful that the expression doesn't have any unexpected side effects.

Enable group

The **Enable Group** action button allows for a breakpoint to reactivate a group of breakpoints that have been previously disabled.

Disable group

The **Disable Group** radio button lets you disable a group of breakpoints. When a group of breakpoints is disabled, the

breakpoints are not erased, they are simply masked for the debugging session.

Setting conditions and actions

The most important step when setting up breakpoints is specifying the conditions under which the breakpoint triggers and specifying the actions to be taken once the breakpoint takes effect. Two text boxes control these settings, the Condition Expression text box and the Action Expression text box.

Condition Expression

When you choose either a **Changed Memory**, **Expression True**, or **Hardware Condition** radio button, you must provide a set of conditions so TDW knows when to trigger the breakpoint. A *condition set* consists of one or more expressions; each condition has to evaluate true in order for the whole set to evaluate true.

For more information on specifying breakpoint actions, see the "Action Expression" section that follows.

A condition set is associated with a set of actions. When the condition set evaluates true, the corresponding action set is performed.

To add a condition set to a breakpoint,

1. Select either the **Changed memory**, **Expression True**, or **Hardware** radio button.
2. Select the **Add** button located under the **Condition Expression** text box.
3. Enter the condition or variable expression into the **Condition Expression** text box.
4. If you want more than one variable or condition to be tested for a particular action set, repeat steps 2 and 3 until all expressions have been added to the **Condition Expression** text box.
5. Once you've specified a condition set, use the **Action Expression** text box to list the action(s) you'd like to take when the breakpoint triggers.



A single breakpoint may have several condition and action sets associated with it. If you want more than one set of conditions and actions assigned to a single breakpoint, choose **OK** after you have entered the first series of conditions and actions. This will close the **Conditions and Actions** dialog box and return you to the **Breakpoint Options** dialog box. From here, choose **Add** to enter a new set of conditions and actions. When a breakpoint has multiple condition and action sets, each one will be evaluated in

the order that they were entered. If more than one action set evaluates to true, then more than one set of actions will be performed.

The **Delete** button located below the **Condition Expression** text box lets you remove the currently highlighted expression from the **Condition Expression** text box. Select this button if you want to delete a condition from the condition set.

Action Expression

When either an **Execute**, **Log**, **Enable Group**, or **Disable Group** Action radio button is chosen, an action set must be provided so TDW knows what to do when the breakpoint triggers. An *action set* is composed of one or more actions.

To add an action set to a breakpoint,

1. Select either the **Execute**, **Log**, **Enable Group**, or **Disable Group** radio button.
2. Select the **Add** button located under the Action Expression text box.
3. Enter the action into the Action Expression text box.

To perform more than one action when the breakpoint triggers, repeat steps 2 and 3 until all actions have been added to the Action Expression text box.

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



If the **Enable Group** or **Disable Group** radio button is chosen, simply type the breakpoint group number into the Action Expression text box to reference the group that you want to enable or disable.

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

Pass count

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

Customizing breakpoints

In addition to simply stopping your program at a particular point, greater control can be given to debugging by stipulating when a breakpoint should take action, and what it should do when it triggers.

Simple breakpoints

When a breakpoint is initially set, it is given the default setting of **Always Break**. Once a simple breakpoint is set, the actions and conditions of the breakpoint may be customized. There are a number of ways to set a simple breakpoint, each one being convenient in different circumstances:

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

Global breakpoints

When a breakpoint is made global, TDW will check the breakpoint on the execution of every line of code. If the set of conditions evaluates true, then the corresponding set of actions will be executed.



If you want a global check to occur on every machine code instruction, set a global breakpoint, and press *F9* from within the CPU window. This type of code monitoring should only be done once you have isolated a small area of your program known to contain a problem. The CPU window can then be used to locate the exact position of the difficulty.

You must check Global if you want to set hardware breakpoints.

Since a debugger action will occur on every line of source code or machine instruction, global breakpoints greatly slow the

execution of your program. Be careful with your use of global breakpoints; they should be used only if you want to find out exactly when a variable changes value, when some condition becomes true, or when your program is “bashing” data.

*The **B**reakpoints menu offers shortcuts for defining global breakpoints. For more information on the **C**hanged Memory Global and **E**xpression True Global commands, see page 105.*

Often, global breakpoints are used to watch for when a data item changes value. In this situation, TDW checks the area of memory for change after the execution of every line of code. As an alternative to a global breakpoint, you may want to specify a breakpoint that only watches for a change when a specific source statement is reached. This is a lot more efficient, since it reduces the amount of processing TDW does in order to detect the change (in this case, TDW isn't concerned with when the item has changed, only that it has changed).

Changed memory breakpoints

When you want to find out where in your program a certain data object is being changed, first set a breakpoint using one of the techniques outlined in the preceding section. Then, using the **C**hanged Memory radio button in the Conditions and Actions dialog box, enter an expression that refers to the memory area you want to watch along with an optional count of the number of objects to track. The total number of bytes in the watched area is the size of the object that the expression references times the number of objects. For example, suppose you have declared the following C array:

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

If you want to watch for a change in the first ten elements of this array, enter the following item into the **C**ondition Expression input box:

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

The area watched is 20 bytes long, since an **int** is 2 bytes and you said to watch ten of them.

If the **C**hanged Memory breakpoint is global, your program executes much more slowly because the memory area is checked for change after every source line has been executed. If you've installed a hardware device driver, TDW will try to set a hardware breakpoint to watch for a change in the data area. Different hardware debuggers support different numbers and types of hardware breakpoints. You can see if a breakpoint is using the hardware by opening a Breakpoint window with the

View | Breakpoints command. Any breakpoint that is hardware assisted will have an asterisk (*) beside it. These breakpoints are much faster than global breakpoints that are not hardware assisted.

Conditional expressions

There are many occasions when you won't want a breakpoint to be triggered every time a certain source statement is executed, particularly if that line of code is executed many times before the occurrence you are interested in. TDW gives you two ways to qualify when a breakpoint is actually triggered: *pass counts* and *conditions*.

Scope of breakpoint expressions

See Chapter 9 for a complete discussion of scopes and scope overrides.

Both the action that a breakpoint performs and the condition under which it is triggered can be controlled by an expression you supply. That expression is evaluated using the scope of the address at which the breakpoint is set, not the scope of the current location where the program is stopped. This means that your breakpoint expression can use only variable names that are valid at the address in your program where you set the breakpoint, unless you use scope overrides.

If you want to set a breakpoint for an expression in a module that isn't currently loaded and TDW cannot find that expression, you can use either a scope override to specify the file that contains the expression or the **View | Module** command to change modules.

If you use variables that are local to a routine as part of an expression, that breakpoint will execute much more slowly than a breakpoint that uses only global or module local variables.

Hardware breakpoints

See page 12 for information on setting up device drivers for hardware debugging.

A hardware breakpoint uses hardware debugging support, either through a hardware debugging board or through the debugging registers of the Intel 80386 (or higher) processor. If your system is set up for hardware debugging (**File | Get Info** shows Breakpoints set to Hardware), you can set a hardware breakpoint using one of the following methods:

- Choose **Breakpoints | Changed Memory Global**, the most common use of hardware breakpoints.
- Choose **Breakpoints | Hardware**.

- Display the Breakpoint Options menu (choose **B**reakpoints | **A**t or the Set Options command of the **V**iew | **B**reakpoints window local menu), then do the following:
 1. Check the Global check box.
 2. Push the Change button.
 3. In the Conditions and Actions dialog box, choose the Hardware radio button to turn on the Hardware pushbutton at the bottom of the dialog box.
 4. Push the Hardware push button to display the Hardware Breakpoint Options dialog box.
 5. Choose the options you want from this dialog box. The options are described in the online text file HDWDEBUG.TD.

Logging variable values

For more information on the Log window, see page 120.

Be careful of side effects when logging expressions.

Sometimes, you may find it useful to log the value of certain variables each time you reach a certain place in your program. You can log the value of any expression, including, for example, the values of the parameters a function is called with. By looking at the log each time the function is called, you can determine when it was called with erroneous parameters.

Choose the **L**og radio button from the Breakpoint Options dialog box. You are prompted for the expression whose value is to be logged each time the breakpoint is triggered.

Breakpoints and templates

TDW supports breakpoints on C++ templates. Breakpoints get set differently depending on if you use **F2** in the Module window, **F2** in the CPU window, or the Breakpoint Options dialog box to set them.

Breakpoints on class templates

There are several methods for setting breakpoints in templates:

- If you set a breakpoint in the template itself by pressing **F2** while the cursor is on a line of template source code in the Module window, breakpoints are set in all class instances of that template. This feature allows you to debug overall behavior of the template.

- If you set a breakpoint in the template by pressing *Alt-F2* to display the Breakpoint Options dialog box, entering the Module window address of a template expression brings up a dialog box that lets you choose the class instance for which you want to set the breakpoint.
- If you open the CPU window, you can see where template code appears in each class instance of a template. If you position the cursor on a line of template code in one of the class instances, pressing *F2* will set a breakpoint on that class instance only.

You can remove a template breakpoint just as you remove any breakpoint, by positioning in the Module window on the highlighted line in the template and pressing *F2* or by using the delete command of the **B**reakpoints window. When you do so, any associated class instance breakpoints are removed as well.

If you position in the CPU window on a breakpoint in a class instance and press *F2*, only the breakpoint for that class instance is removed.

Breakpoints on function templates

You set and remove breakpoints for function templates just as you do for class templates. The two methods for setting breakpoints, pressing *F2* or using the Breakpoint Options dialog box, have the same effects on function instances as they do on class instances.

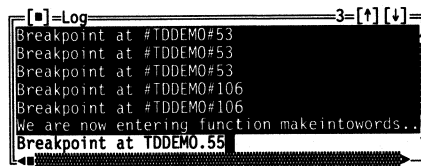
Breakpoints on template class instances and objects

You set breakpoints for template class instances and objects of template class instances just like you do for ordinary classes and objects.

The Log window

You create a Log window by choosing the **V**iew | **L**og command. This window lets you review a list of significant events that have taken place in your debugging session.

Figure 7.6
The Log window



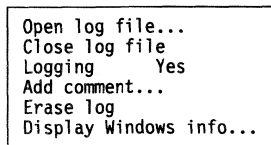
Log windows show a scrolling list of the lines output to the window. If more than 50 lines have been written to the log, the

oldest lines are lost from the top of the scrolled list. If you want to change the number of lines in the list, use the TDWINST customization program (described in the file TDWINST.DOC). You can also preserve the entire log, continuously writing it to a disk file, by using the **O**pen Log File local menu command.

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

- Your program stops at a location you specified. The location it stops at is recorded in the log.
- You issue the **A**dd Comment local menu command. You are prompted for a comment to write to the log.
- A breakpoint is triggered that logs the value of an expression. This value is put in the log.
- You use the **E**dit | **D**ump Pane to Log command (from the menu bar) to record the current contents of a pane in a window.
- You are debugging a Windows application and use the Display Windows Info command on the Log window local menu to write global heap information, local heap information, or the module list to the log.
- You are debugging a Windows application, have used the **V**iew | **W**indows Messages command to display the Windows Messages window, and are now in the local menu of the Messages pane of that window. You toggle **S**end to Log Window to *Yes* so all messages coming to this window will also go to the Log window.

The Log window local menu



A screenshot of the local menu for the Log window. The menu items are: Open log file..., Close log file, Logging Yes, Add comment..., Erase log, and Display Windows info...

The commands in this menu let you control writing the log to a disk file, stopping and starting logging, adding a comment to the log, clearing the log, and writing information about a Windows program to the log.

Alt-F10 pops up the Log window local menu. If you have control-key shortcuts enabled, pressing *Ctrl* and the first letter of the command accesses the command directly.

Open Log File

Causes all lines written to the log to be written to a disk file as well. A dialog box appears that prompts you for the name of the file to write the log to (or you can select a directory and file from the list boxes).

When you open a log file, all the lines already displayed in the log window's scrolling list are written to the disk file. This lets you open a disk log file *after* you see something interesting in the log that you want to record to disk.


If you want to start a disk log that does not start with the lines already in the Log window, first choose **Erase Log** before choosing **Open Log File**.

Close Log File Stops writing lines to the log file specified in the **Open Log File** local menu command, and closes the file.

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

Add Comment Lets you insert a comment in the log. You are prompted for a line of text that can contain any characters you want.

Erase Log Clears the log list. The Log window will now be blank. Only the log in memory is affected, not the parts of the log that have been written to a disk file.

Display Windows Info  Displays the Windows Information dialog box, which lets you list global heap information, local heap information, or the list of modules making up your application. See page 166 in Chapter 11 for an explanation of how to use this feature.

Examining files

TDW treats disk files as a natural extension of the program you're debugging. You can examine any file on the disk, viewing it either as ASCII text or as hex data.

This chapter shows you how to examine disk files that contain your program source code and other files on disk.

Examining program source files

Loading and debugging Windows DLL modules is described in Chapter 11 on page 169.

Program source files are your source files that are compiled to generate an object module (an .EXE file). You usually examine them when you want to look at the behavior or design of a portion of your code. During debugging, you often need to look at the source code for a routine to verify either that its arguments are valid or that it is returning a correct value.

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



Files that are included in a source file by a compiler directive and generate line numbers are also considered to be program source files, even though they don't appear in the Pick a Module list pane when you choose **View | Module**. To select one of these files, you must use the local menu **File** command.

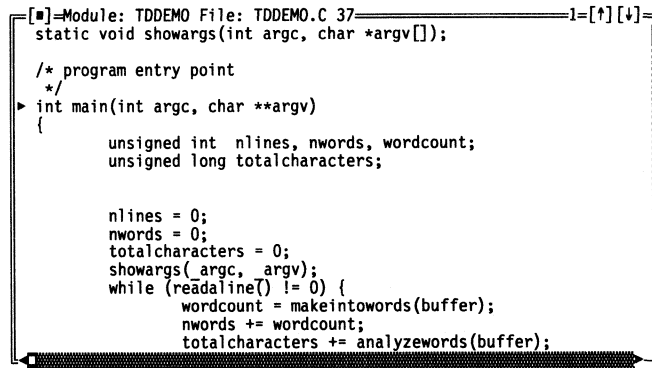
You should always use a Module window to look at your program source files because doing so informs TDW that the file

is a source module. TDW then lets you do things like setting breakpoints or examining program variables simply by moving to the appropriate place in your file. These techniques and others are described in the following sections.

The Module window

Figure 8.1
The Module window

Before you can open a module window, you must have a program loaded. You create a Module window by choosing the **View | Module** command from the menu bar (or pressing the hot key, **F3**).



```
[*]=Module: TDDEMO File: TDDEMO.C 37
static void showargs(int argc, char *argv[]);

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

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

See page 169 for a description of this Dialog box.

When you run TDW, you need both the .EXE file and the original source file.

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

TDW then loads the source file for the module you select. If you select a source module (and not a DLL), TDW searches for the source file in the following places:

1. in the directory where the compiler found the source file
2. in the directories specified by the **Options | Path for Source** command or the **-sd** command-line option
3. in the current directory
4. in the directory that contains the program you're debugging

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

If the abbreviation *opt* appears after the file name in the title, the program has been optimized by the compiler. You might have trouble finding some variables that have been optimized away. In addition, variables that have become register variables won't have an address.

If the word *modified* appears after the file name in the title, the file has been changed since it was last compiled or linked to make the program you are debugging. In this case, the routines in the updated source file may no longer have the same line numbers as those in the version used to build the program you are debugging. If the line numbers are different, the arrow that shows the current program location (CS:IP) will be displayed on the wrong line.

The Module window local menu

Inspect Watch
Module... File...
Previous Line... Search... Next Origin Goto...

The Module window local menu provides a number of commands that let you move around in the displayed module, point at data items and examine them, and set the window to display a new file or module.

You will probably use this menu more than any other menu in TDW, so you should become quite familiar with its various options.

Use the *Alt-F10* key combination to pop up the Module window local menu. If you have control-key shortcuts enabled, you can access local menu commands without popping up the menu: Use the *Ctrl* key with the highlighted letter of a command to access that command (for example, *Ctrl-S* for **S**earch).

Inspect Opens an Inspector window to show you the contents of the program variable at the current cursor position. If the cursor isn't currently on a variable, you're prompted to enter one.

Because this command saves you from having to type in each name you are interested in, you'll end up using it a lot to examine the contents of your program variables.

Watch 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 the cursor isn't currently on a variable, you're prompted to enter one.

Module Lets you view a different module by picking the one you want from the list of modules displayed. This command is useful when you are no longer interested in the current module, and you don't want to end up with more Module windows onscreen.

File Lets you switch to view one of the other source files that makes up the module you are viewing. Pick the file that you want to view from the list of files presented. Most modules only have a single source file that contains code. Other files included in a module usually only define constants and data structures. Use this command if your module has source code in more than one file.

Use **View | Module** to look at the first file. If you want to see more than one, use **View | Another | Module** to open subsequent Module windows.

Previous Returns you to the last source module location you were viewing. You can also use this command to return to your previous location after you've issued a command that changed your position in the current module.

Line Positions you at a new line number in the file. Enter the new line number to go to. If you enter a line number after the last line in the file, you will be positioned at the last line in the file.

Search Searches for a character string, starting at the current cursor position. Enter the string to search for. If the cursor is positioned over something that looks like a variable name, the Search dialog box will come up initialized to that name. Also, if you have marked a block in the file using the *Ins* key, that block will be used to initialize the Search dialog box. This saves you from typing if what you want to search for is a string that is already in the file you are viewing.

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

Next Searches for the next instance of the character string you specified with the **Search** command; you can only use this command after initially choosing **Search**.

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

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

Goto Positions you at any location within your program. Enter the address you want to examine; you can enter a procedure name or a hex address. See Chapter 9 for a complete description of the ways to enter an address.

If the address doesn't have a corresponding source line, a CPU window is opened.

You can also invoke this command by simply starting to type the label to go to. This brings up a dialog box exactly as if you had chosen the **Goto** command. Entering the label name is a handy way to invoke this frequently used command.

Examining other disk files

You can examine any file on your system by using a File window. You can view the file either as ASCII text or as hex data bytes, using the **Display As** command described in a later section of this chapter.

The File window

You create a File window by choosing **View | File** from the menu bar. You can use DOS-style wildcards to get a list of file choices, or you can type a specific file name to load.

Figure 8.2
The File window

```
[*]=File ...S\DEBUG\DEMOS\TDDEMO.C 1==3=[↑][↓]
/*      file <tddemo.c>
*
*      Demonstration program to show off Turbo
*      Reads words from standard input, analyz
*      Copyright (c) 1988, 1991 - Borland Inte
*/
```

File windows show the contents of the file you've selected. The name of the file you are viewing is displayed at the top of the window, along with the line number the cursor is on if the file is displayed as ASCII text.

When you first create a File window, the file appears either as ASCII text or as hexadecimal bytes, depending on whether the file contains what TDW thinks is ASCII text or binary data. You can switch between ASCII and hex display at any time using the **Display As** local menu command described later.

Figure 8.3
The File window showing hex data

```
[*]=File ...S\DEBUG\DEMOS\TDDEMO.C ==3=[↑][↓]
00000: 2f 2a 09 66 69 6c 65 20 /*ofile
00008: 3c 74 64 64 65 6d 6f 2e <tddemo.
00010: 63 3e 0d 0a 20 2a 0d 0a c>] *J
00018: 20 2a 09 44 65 6d 6f 6e *oDemon
00020: 73 74 72 61 74 69 6f 6e stration
00028: 20 70 72 6f 67 72 61 6d program
00030: 20 74 6f 20 73 68 6f 77 to show
```

The File window local menu

The File window local menu has a number of commands for moving around in a disk file, changing the way the contents of the file are displayed, and making changes to the file.

Goto...	
Search...	
Next	
Display as	Ascii
File...	

Use the **Alt-F10** key combination to pop up the File window local menu or, if you have control-key shortcuts enabled, use the **Ctrl** key with the highlighted letter of the desired command to access the command without invoking the local menu.

Goto Positions you at a new line number or offset in the file. If you are viewing the file as ASCII text, enter the new line number to go to. If you are viewing the file as hexadecimal bytes, enter the offset from the start of the file at which to start displaying. You can use the full expression parser for entering the offset. If you enter a line number after the last line in the file or an offset beyond the end of the file, TDW positions you on the last line of the file.

Search Searches for a character string, starting at the current cursor position. You are prompted to enter the string to search for. If the cursor is positioned on something that looks like a symbol name, the Search dialog box comes up initialized to that name. Also, if you have marked a block in the file using the *Ins* key, that block will be used to initialize the Search dialog box. This saves you from typing if what you want to search for is a string that is already in the file you are viewing. The format of the search string depends on whether the file is displayed in ASCII or hex.

If the file is displayed in ASCII, you can use simple DOS wildcards, with ? indicating a match on any single character, and * matching 0 or more characters.

See page 139 for complete information about byte lists.

If the file is displayed in hexadecimal bytes, enter a byte list consisting of a series of byte values or quoted character strings, using the syntax of whatever language you are using for expressions.

For example, if the language is C++, a byte list consisting of the hex numbers 0408 would be entered as follows:

```
0x0804
```

If the language is Pascal, the same byte list is entered as

```
$0804
```

The search does not wrap around from the end of the file to the beginning. To search the entire file, go to the first line of the file by pressing *Ctrl-PgUp*.

You can also invoke this command by simply starting to type the string that you want to search for. This brings up a dialog box exactly as if you had specified the **Search** command.

Next Searches for the next instance of the character string you specified with the **Search** command; you can only use this command after initially choosing **Search**.

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

Display As Toggles between displaying the file as ASCII text or as hexadecimal bytes.

- If you choose ASCII display, the file appears as you are used to seeing it on the screen in an editor or word processor.
- If you choose Hex display, each line starts with the hex offset from the beginning of the file for the bytes on the line. Eight bytes of data are displayed on a line. To the right of the hex display of the bytes, the display character for each byte appears. The full display character set can be displayed, so byte values less than 32 or greater than 127 appear as the corresponding display symbol.

File Lets you switch to a different file. You can use DOS wildcards to get a list of file choices, or you can type a specific file name to load. **File** lets you view a different file without putting a new File window onscreen. If you want to view two different files or two parts of the same file simultaneously, choose **View | Another | File** to make another File window.

Expressions

Expressions can be a mixture of symbols from your program (that is, variables and names of routines), and constants and operators from one of the supported languages: C, Pascal, or assembler.

Each language evaluates an expression differently.

TDW can evaluate expressions and tell you their values. You can also use expressions to indicate data items in memory whose value you want to know. You can supply an expression in any dialog box that asks for a value or an address in memory.

Use **Data | Evaluate/Modify** to open the Evaluate/Modify dialog box, which tells you the value of an expression. (You can also use this dialog box or the Watches window as a simple calculator.)

In this chapter, you'll learn how TDW chooses which language to use for evaluating an expression and how you can make it use a specific language. We describe the components of expressions that are common to all the languages, such as source-line numbers and access to the processor registers. We then describe the components that can make up an expression in each language, including constants, program variables, strings, and operators. For each language, we also list the operators that TDW supports and the syntax of expressions.

For a complete discussion of C, C++, and assembler expressions, refer to your *Turbo C++ for Windows User's Guide*.

Choosing the language for expression evaluation

TDW normally determines which expression evaluator and language to use from the language of the current module. This is the module in which your program is stopped. You can override this by using the **Options | Language** command to open the Expression Language dialog box; in it you can set radio buttons to **Source**, **Pascal**, **C**, or **Assembler**. If you choose **Source**, expressions are evaluated in the manner of the module's language. (If TDW can't determine the module's language, it uses the expression rules for inline assembler.)

Usually, you let TDW choose which language to use. Sometimes, however, you'll find it useful to set the language explicitly; for example, when you are debugging an assembler module that is called from one of the other languages. By explicitly setting expression evaluation to use a particular language, you can access your data in the way you refer to it with that language, even though your current module uses a different language.

Sometimes it's convenient to treat expressions or variables as if they had been written in a different language; for example, if you're debugging a C++ program, assembly language conventions might offer an easier way to change the value of a byte stored in a string.

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

Even if you deliberately choose the wrong language when you enter TDW, it will still be able to get some information about the original source language from the symbol table and the original source file. Under some circumstances, however, it may be possible to cause TDW to store data incorrectly.

Code addresses, data addresses, and line numbers

Normally, when you want to access a variable or the name of a routine in your program, you simply type its name. However, you can also type an expression that evaluates to a memory pointer, or

specify code addresses as source line numbers by preceding the line number with a number sign (#), like #123 (C, C++, and Assembler only). The next section describes how to access symbols outside the current scope.

Of course, you can also specify a regular segment:offset address, using the hexadecimal syntax for the source code language of your program:

Language	Format	Example
C	0xnnnn	0x1234:0x0010
Assembler	nnmh	1234h:0010h 1234h:0B234h

In assembler, hex numbers starting with A to F must be prefixed with a zero.

Accessing symbols outside the current scope

Where the debugger looks for a symbol is known as the *scope* of that symbol. Accessing symbols outside of the current scope is an advanced concept that you don't really need to understand in order to use TDW in most situations.

Normally, TDW looks for a symbol in an expression the same way a compiler would. For example, Pascal first looks in the current procedure or function, then in an "outer" subprogram (if the active scope is nested inside another), then in the implementation section of the current unit (if the current scope resides in a unit), and then for a global symbol.

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

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

Scope override syntax

Depending on your current language setting, you use a different symbol to override the scope of a symbol name. Because you can change the language setting with **Options | Language** in order to use features of the different scope override syntaxes, we show you both sets of syntax.

- With C, C++, and Turbo Assembler, use the cross hatch (#) symbol to override scope.
- With Pascal, use the period (.) to override scope.

You can enter qualified identifier expressions anywhere an expression is valid, including

- the Evaluate/Modify dialog box
- the Watches window
- a **Data | Inspector** window
- the dialog box displayed by the Goto local menu command of the Module window (when you want to go to an address in the source code)

Overriding scope in C, C++, and assembler programs

Scope operators don't work with register variables.

Use a pound sign (#) to separate the components of the scope.

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

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

or

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

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

For example, in the Watches window, you could enter different line numbers for the TDDEMO variable *nlines* so you could see how its value changes in different routines in the current module. To watch the variable both on line 51 and on line 72, you would make the following entries in the Watches window:

```
#51#nwords  
#72#nwords
```

Here are some examples of valid symbol expressions with scope overrides. There is one example for each of the legal combinations of elements that you can use to override a scope.

The first six examples show various ways of using line numbers to generate addresses and override scopes:

```
#123
    Line 123 in the current module

#123#myvar1
    Symbol myvar1 accessible from line 123 of the current
    module

#mymodule#123
    Line 123 in module mymodule

#mymodule#123#myvar1
    Symbol myvar1 accessible from line 123 in module mymodule

#mymodule#file1.cpp#123
    Line 123 in source file file1.cpp, which is part of module
    mymodule

#mymodule#file1.cpp#123#myvar1
    Symbol myvar1 accessible from line 123 in source file
    file1.cpp, which is part of mymodule
```

The next six examples show various ways of overriding the scope of a variable by using a module, file, or function name:

```
#myvar2
    Same as myvar2 without the #

myfunc#myvar2
    Variable myvar2 accessible from routine myfunc

#mymodule#myvar2
    Variable myvar2 accessible from module mymodule

#mymodule#myfunc#myvar2
    Variable myvar2 accessible from routine myfunc in module
    mymodule

#mymodule#file2.c#myvar2
    Variable myvar2 accessible from file2.c, which is included in
    mymodule

#mymodule#file2.c#myfunc
    myfunc defined in file file2.c, which is included in mymodule
```

OOP

The following four examples show how to use scope override syntax with C++ classes, objects, member functions, and data members:

AnObject#AMemberVar

Data member *AMemberVar* accessible in object *AnObject*
accessible in the current scope

AnObject#AMemberF

Member function *AMemberF* accessible in object *AnObject*
accessible in the current scope

#AModule#AnObject#AMemberVar

Data member *AMemberVar* accessible in object *AnObject*
accessible in module *AModule*

#AModule#AnObject#AClass::AMemberVar

Data member *AMemberVar* of class *AClass* accessible in
object *AnObject* accessible in module *AModule*



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

Scope override tips

The following tips might help you when overriding scope in C, C++, and Turbo Assembler programs:

1. If you use a file name in a scope override statement, it must be preceded by a module name.
2. If a file name has an extension, such as .ASM, .C, or .CPP, you must specify the extension; Turbo Debugger doesn't try to determine the extension itself.
3. If a function name is the first item in a scope override statement, it must not have a # in front of it. If there's a #, Turbo Debugger interprets the function name as a module name.
4. Any variable you access through scope override syntax must have been initialized already. An automatic variable doesn't have to be in scope, but its function must have run already.
5. If you're trying to access an automatic variable that's no longer in scope, you must use its function name as part of the scope override statement.
 - The scope of a template depends on the current location in the program. Watches and Inspector windows on template

expressions are dependent on the current object the program is in.

- A nested class is in the scope of the class it's nested in. The scope of a nested class isn't global to the program.

Overriding scope in Pascal programs

Use a period (.) to separate the components of the scope.

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

```
[unit.][procedurename.]variablename
```

or

```
[unit.][objecttype.][objectinstance.][method.]fieldname
```

OOP

If you don't specify a unit, the current unit is assumed.

Here are some examples of valid symbol expressions with scope overrides. There is one example for each of the legal combinations of elements that you can use to override a scope.

These examples show various ways of overriding the scope of a variable by using a module or procedure name:

```
MyVar2
```

Variable *MyVar2* in the current scope

```
MyProc.MyVar2
```

Variable *MyVar2* accessible from routine *MyProc*

```
MyUnit.MyVar2
```

Variable *MyVar2* accessible from unit *MyUnit*

```
MyUnit.MyProc.MyVar2
```

Variable *MyVar2* accessible from routine *MyProc* in unit *MyUnit*

OOP

The following examples show how to use scope override syntax with object types, object instances, fields, and methods:

```
AnInstance
```

Instance *AnInstance* accessible in the current scope.

```
AnInstance.AField
```

Field *AField* accessible in instance *AnInstance* accessible in the current scope

```
AnObjectType.AMethod
```

Method *AMethod* accessible in object type *AnObjectType* accessible in the current scope

`AnInstance.AMethod`
Method *AMethod* accessible in instance *AnInstance* accessible in the current scope

`AUnit.AnInstance.AField`
Field *AField* accessible in instance *AnInstance* accessible in unit *AUnit*

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

`AUnit.AnInstance.AMethod.ANestedProc.AVar`
Local variable *AVar* accessible in nested procedure *ANestedProc* accessible in method *AMethod* accessible in instance *AnInstance* accessible in unit *AUnit*

Scope override tips

The following tips might help you when overriding scope in Pascal programs:

1. Any variable you access through scope override syntax must have been initialized already. The procedure or function containing a local variable doesn't have to be in scope, but it must have run already.
2. If you are trying to access a local variable that's no longer in scope, you must use its procedure or function name as part of the scope override statement.
3. You can't use a line number or a file name as part of a Pascal scope override statement. However, you can use **Options | Language** to change the language to C so you *can* use line number syntax.

Scope and DLLs Because TDW simultaneously loads the symbol tables of the current module of your .EXE file and of any DLLs it accesses that have source code and symbol tables, you might not have immediate access to variables in your DLLs (or in your .EXE if you're currently in a DLL).

TDW looks for a variable first in the symbol table of the current module or DLL, and then in any other symbol tables in order of loading. If a variable has the same name in multiple DLLs or in your .EXE and one or more DLLs, TDW sees only the first instance it finds. You can't use scope override syntax to access any

⇒ such variables; instead, you must press *F3* and use the Load Modules and DLLs dialog box to load the appropriate module or DLL.

TDW loads symbol tables for the following:

1. the current module of your .EXE file
2. any DLL you explicitly load using the Symbol Load command in the Load Modules and DLLs dialog box (displayed with *F3* or **View | Module**)
3. any DLL you step into from your program

Implied scope for expression evaluation

Whenever TDW evaluates an expression, it must decide where the *current scope* is for any symbol names without an explicit scope override. Determining scope is important because in many languages you can have symbols inside functions or procedures with the same name as global symbols, and TDW must know which instance of a symbol you mean.

TDW usually uses the current cursor position as the context for determining the scope. Thus, you can set the scope where an expression will be evaluated by moving the cursor to a specific line in a Module window.

One result is that if you've moved the cursor off the current line where your program is stopped, you might get unexpected results from evaluating expressions. If you want to be sure that expressions are evaluated in your program's current scope, use the **Origin local** menu command in the Module window to return to the current location in the source code. You can also set the expression scope by moving around inside the Code pane of a CPU window, by moving the cursor to a routine in the Stack window, or by moving the cursor to a routine name in a Variables window.

Byte lists

Several commands ask you to enter a list of bytes, including the **Search** and **Change local** menu commands in the Data pane of the CPU window, and the **Search** local menu command of the File window when it's displaying a file in hexadecimal format.

A *byte list* can be any mixture of scalar (non-floating-point) numbers and strings in the syntax of the current language, determined by the `Options | Language` command. Both strings and scalars use the same syntax as expressions. Scalars are converted into a corresponding byte sequence. For example, a Longint value of 123456 becomes a 4-byte hex quantity `40 E2 01 00`.

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

C expressions

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

C symbols

A symbol is the name of a data object or routine in your program. A symbol name must start with a letter (*a-z, A-Z*) or underscore (`_`). Subsequent characters can be any of these characters as well as the digits 0 through 9. You can omit the beginning underscore from symbol names; if you enter a symbol name without an underscore and TDW can't find that name, it searches for the name again with an underscore at the beginning. Because the compiler automatically puts an underscore at the start of your symbol names, you don't have to remember to add one.

C register pseudovariables

TDW lets you access the processor registers using the same technique as one of Borland's C or C++ compilers, namely pseudovariables. A *pseudovariable* is a variable name that corresponds to a given processor register.

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

The following pseudovvariables let you access the 80386 processor registers:

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

C constants and number formats

Constants can be either floating point or integer.

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

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

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

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

1.234 4.5e+11

Escape sequences

A string is a sequence of characters enclosed in double quotes (""). You can use the standard C backslash (\) as an escape character.

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

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

C operator precedence

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

```
0x1234::0x1000
_ES::_BX
```

The primary expression operators

```
[] . -> sizeof
```

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

```
* & - ! ~ ++ --
```

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

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

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

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

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

Executing C functions in your program

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

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

```
long power(int x, int y)
{
    long temp = 1;
    while (y-- > 0)
        temp *= x;
    return(temp);
}
```

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

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

C expressions with side effects

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

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

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

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

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

C reserved words and type conversion

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

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

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

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

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

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

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



TDW provides two reserved words, **lh2fp** and **gh2fp**, for dereferencing memory handles used in Windows applications. See page 175 for an explanation of these two type conversions.

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

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

Assembler expressions

TDW supports the complete assembler expression syntax. An assembler expression consists of a mixture of symbols, operators, strings, variables, and constants. Each of these components is described in this section.

Assembler symbols

Symbols are user-defined names for data items and routines in your program. An assembler symbol name starts with a letter (*a-z*, *A-Z*) or one of these symbols: *@ ? _ \$*. Subsequent characters in the symbol can contain the digits *0* to *9*, as well as these characters. The period (*.*) can also be used as the first character of a symbol name, but not within the name.

The special symbol *\$* refers to your current program location as indicated by the CS:IP register pair.

Assembler constants

Constants can be either floating point or integer. A floating-point constant contains a decimal point and may use decimal or scientific notation. For example,

1.234 4.5e+11

Integer constants are hexadecimal unless you use one of the assembler conventions for overriding the radix:

If you want to end a hex number with a D or B, you must append an H to avoid ambiguity.

Format	Radix
digitsH	Hexadecimal
digitsO	Octal
digitsQ	Octal
digitsD	Decimal
digitsB	Binary

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

Assembler operators

TDW supports most of the assembler operators. The first line in the list that follows shows the operators with the lowest priority, and the last line those operators with the highest priority. Within a line, all the operators have the same priority.

xxx PTR (BYTE PTR...)
.
(structure member selector)
:
(segment override)
OR XOR
AND
NOT
EQ NE LT LE GT GE
+ -
* / **MOD SHR SHL**
Unary + Unary -
OFFSET SEG
() []

Variables can be changed using the = assignment operator. For example,

```
a = [BYTE PTR DS:4]
```

Format control

When you supply an expression to be displayed, TDW displays it in a format based on the type of data it is. TDW ignores a format control that is wrong for a particular data type.

If you want to change the default display format for an expression, place a comma at the end of the expression and supply an optional repeat count followed by an optional format letter. You can only supply a repeat count for pointers or arrays.

Character	Format
c	Displays a character or string expression as raw characters. Normally, nonprinting character values are displayed as some type of escape or numeric format. This option forces the characters to be displayed using the full IBM display character set.
d	Displays an integer as a decimal number.
f[#]	Displays as floating-point format with the specified number of digits. If you don't supply a number of digits, as many as necessary are used.
m	Displays a memory-referencing expression as hex bytes.
md	Displays a memory-referencing expression as decimal bytes.
p	Displays a raw pointer value, showing segment as a register name if applicable. Also shows the object pointed to. This is the default if no format control is specified.
s	Displays an array or a pointer to an array of characters as a quoted character string.
x or h	Displays an integer as a hexadecimal number.

Object-oriented debugging

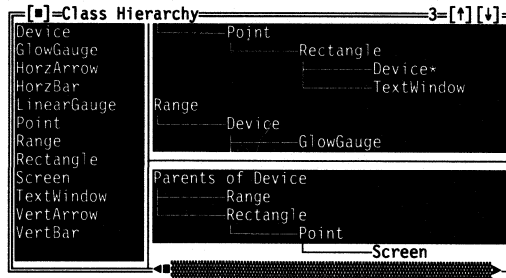
OOP

To meet the needs of the object-oriented programming revolution, TDW supports C++. Besides extensions that let you trace into member functions and examine objects in the Evaluate/Modify dialog box and the Watches window, TDW comes equipped with a special set of windows and local menus specifically designed for object types and classes.

The Hierarchy window

TDW provides a special window for examining class hierarchies. You can bring up the Hierarchy window by choosing **View | Hierarchy**.

Figure 10.1
The Hierarchy window



Use *Tab* to move between the two panes.

The Hierarchy window displays information on classes rather than instances of classes. There are either two or three panes. If the program is a C++ program using multiple inheritance, three panes appear. Otherwise, only two panes appear.

- The left pane, the Class List pane, lists in alphabetical order the classes used by the module being debugged.
- The top right pane, the Hierarchy Tree pane, shows all classes in their hierarchies by using a line graphic that places the original base class at the left margin of the pane and displays derived classes beneath and to the right of the base class, with lines indicating derived class relationships. Any class followed by an asterisk inherits from multiple base classes and shows up in the pane below.
- The bottom right pane, the Parent Tree pane, if it exists, shows all base classes for classes with multiple inheritance.

The Class List pane

The left pane of the Hierarchy window provides an alphabetical list of all classes used by the current module. It supports an incremental matching feature to eliminate the need to scroll through large lists of classes: When the highlight bar is in the left pane, simply start typing the name of the class you're looking for. At each key press, TDW highlights the first class matching all keys pressed up to that point.

Press *Enter* to open a class Inspector window for the highlighted class. Class Inspector windows are described on page 152.

The Class List pane local menu

Press *Alt-F10* to display the local menu for the pane. You can use the control-key shortcuts if you've enabled hot keys with TDWINST. This local menu contains two items: **Inspect** and **Tree**.

Inspect Tree

Inspect _____

Displays a class Inspector window for the highlighted type.

Tree

Moves to the right pane in which the hierarchy tree is displayed and places the highlight bar on the class that was highlighted in the left pane.

The Hierarchy Tree pane

The top right pane displays the hierarchy tree for all classes used by the current module. Base class and derived class relationships are indicated by lines, with derived classes to the right of and below their base classes.

To locate a single class in a complex hierarchy tree, go back to the left pane and use the incremental search feature; then choose the **Tree** command from the local menu to move back into the hierarchy tree. The matched class appears under the highlight bar.

When you press *Enter*, a class Inspector window appears for the highlighted class.

The Hierarchy Tree pane local menu(s)

Inspect

Inspect
Parents Yes

The Hierarchy Tree pane local menu (press *Alt-F10* in the pane) has only one item for C programs or for C++ programs without multiple inheritance: **Inspect**. When you choose it, a Inspector window appears for the highlighted class. However, a faster and easier method is simply to press *Enter* when you want to inspect the highlighted class.

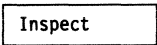
If you have loaded a C++ program that uses classes with multiple inheritance, the Hierarchy Tree pane local menu contains a second command, **Parents**. Use this command to toggle between showing and not showing the base classes of a class with multiple inheritance in the Parent Tree pane. The default for **Parents** is *Yes*.

The Parent Tree pane

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

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

The Parent Tree pane local menu

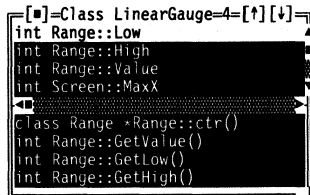


The Parent Tree pane, if it exists, has a local menu of its own with a single command, **Inspect**. It works just the same as the **Inspect** command in the Hierarchy Tree pane local menu: It opens an Inspector window for the highlighted class.

Class Inspector windows

TDW provides a special type of Inspector window to let you inspect the details of a class: the class Inspector window. The window summarizes class information, but does not reference any particular instance. You display this window by bringing up the Hierarchy window (choose **View | Hierarchy**), selecting a class, and pressing **Ctrl-I**.

Figure 10.2
A class Inspector window



The window is divided horizontally into two panes, with the top pane listing the data members of the class and their types, and the bottom pane listing the member function names and the function return types. Use **Tab** to move between the two panes of the class Inspector window.

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 local menu for this pane.) In this way, complex nested structures of classes can be inspected quickly with a minimum of keystrokes.

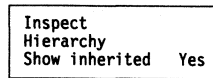
For brevity's sake, member function parameters are not shown in the class Inspector window. To examine parameters, highlight the member function and press **Enter**. A member function Inspector window appears that displays the code address for the object's implementation of the selected member function and the names and types of all its arguments.

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

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

The class Inspector window local menus

Pressing *Alt-F10* brings up the local menu for either pane. If control-key shortcuts are enabled (through TDWINST), you can get to a local menu item by pressing *Ctrl* and the first letter of the item.



The Data Member (top) pane

The Data Member pane local menu contains these items:

Inspect

If the highlighted field is a pointer to a class, a class Inspector window is opened for the highlighted field.

Hierarchy

Opens a Hierarchy window for the class being inspected. The Hierarchy window is described on page 149.

Show Inherited

Yes is the default value of this toggle. When **Show Inherited** is set to *Yes*, TDW shows all data members, whether they are defined within the class of the inspected object or inherited from a base class.

When the toggle is set to *No*, TDW displays only those data members defined within the class being inspected.

The Member Function (bottom) pane

The local menu commands for the bottom Member Function pane are **Inspect**, **Hierarchy**, and **Show Inherited**.

Inspect

A member function Inspector window is opened for the highlighted item. If you press *Ctrl-I* when the cursor is positioned over the address shown in the member function Inspector window, the Module window is brought to the foreground with the cursor at the code that implements what is being inspected.

Hierarchy

Opens a Hierarchy window for the class being inspected. The Hierarchy window is described on page 149.

Show Inherited

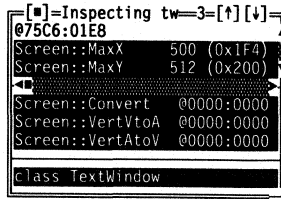
Yes is the default value of this toggle. When it is set to *Yes*, all member functions are shown, whether they are defined within the class being inspected or inherited from a base class. When it is set to *No*, only those member functions are displayed that are defined within the class being inspected.

Object Inspector windows

Class Inspector windows provide information about classes, but say nothing about the data contained in a particular object at a particular time during program execution. TDW provides an extended form of the familiar record Inspector window specifically to inspect objects.

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

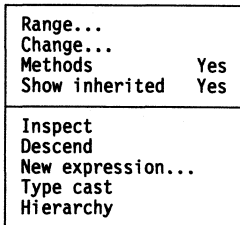
Figure 10.3
An object Inspector window



Most TDW structure Inspector windows have two panes: a top pane summarizing the structure's member names and their current values, and a bottom pane displaying the type of the member highlighted in the top pane. An object Inspector window provides both of those panes, and also a third pane between them. This third pane summarizes the object's member functions, with the code address of each.

The object Inspector window local menus

Each of the top two panes of the object Inspector window has its own local menu, displayed by pressing *Alt-F10* in that pane. Use the control-key shortcuts to get to individual menu items if you've enabled hot keys with TDWINST.




Similar to structure Inspector windows, the bottom pane serves only to display the class of the highlighted data member and doesn't have a local menu.

The local menu commands for the top pane, which summarizes the data members for the selected item, are described here.

Range This command displays the range of array items. If the inspected item is not an array or a pointer, the item cannot be accessed.

Change By choosing this command, you can load a new value into the highlighted data member.

Methods This command is a *Yes/No* toggle, with *Yes* as the default condition. When it's set to *Yes*, member functions are summarized in the middle pane. When it's set to *No*, the middle pane doesn't appear. This toggle is remembered by the next Inspector window to be opened.

- Show Inherited** This command is also a *Yes/No* toggle. When it'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** Choosing this command opens an Inspector window on the highlighted data member. Pressing *Enter* over a highlighted data member does the same thing.
- Descend** The highlighted item takes the place of the item in the current Inspector window. No new Inspector window is opened. However, you can't return to the previously inspected data member, as you could if you had used the **Inspect** option.
-  Use **Descend** to inspect a complex data structure when you don't want to open a separate Inspector window for each item.
- New Expression** This command prompts you for a new data item or expression to inspect. The new item replaces the current one in the window; it doesn't open another window.
- Type Cast** Lets you specify a different data type for the item being inspected. This command is useful if the Inspector window contains a symbol for which there is no type information, as well as for explicitly setting the type for pointers.
- Hierarchy** When you choose this command, a Hierarchy window opens. For a full description of this window, see page 149.

The middle and bottom panes

The middle pane summarizes the member functions of an object. The only difference between the Object Member function pane's local menu and the local menu for the top pane is the absence of the **Change** command. Unlike data members, member functions cannot be changed during execution, so there is no need for this command. The bottom pane displays the type of the item highlighted in the upper two windows.

Using Windows debugging features

This chapter covers the features of TDW that give you access to Windows information and allow you to do the following:

- Log messages received and sent by your application's windows
- List the global heap
- List the local heap
- View the complete list of modules (including dynamic link libraries) loaded by Windows
- Debug dynamic link libraries (DLLs)
- See the contents of any protected-mode selector (in the CPU window)

Windows features

The features that support debugging of Windows programs are

- A view window, the Windows Messages window, which shows messages passed to windows in your program
- Three types of data you can display in the Log window:
 - The data segments in your program's local heap
 - The data segments in the global heap
 - A complete list of modules making up your program, including any dynamic link libraries (DLLs)

- Expression typecasting from memory handles to far pointers
- Support for debugging of DLLs in the Load Module Source or DLL Symbols window (choose **View | Modules**)
- The Selector pane of the CPU window, which allows you to see the contents of any protected-mode selector.

Logging window messages

To track messages being passed to your program's windows, choose the **View | Windows Messages** command to open the Windows Messages window. This window shows you the messages that Windows is passing to one or more windows in your program.

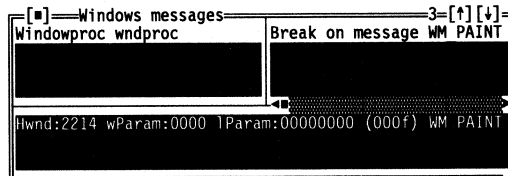
The Windows Messages window is composed of three panes, the Window Selection pane (top left), the Message Class pane (top right), and the Messages pane (bottom). The messages show up in the Messages pane.

The appearance of this window and the way you add application windows to it differ depending on whether you're working with an ObjectWindows application or a standard Windows application.

Selecting a window for a standard Windows application

Figure 11.1
The Windows Messages window for a standard Windows application

If you're debugging a standard Windows application and you select **View | Windows Messages**, you see the following window:



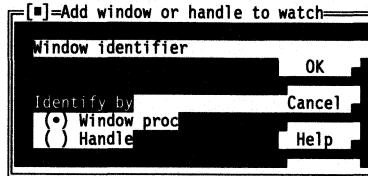
Add...
Remove
Delete all

Before you can log messages, you must first indicate which window you're logging messages for. You do this in the top left pane, the Window Selection pane. This pane's local menu (activated by pressing **Alt-F10**) lets you add a window selection, delete a window selection, or delete all window selections.

Adding a window selection for a standard Windows application

To add a window selection, you can either choose **Add** from the Window Selection pane local menu or begin typing in the pane. Either method brings up the Add Window dialog box.

Figure 11.2
The Add Window dialog box
for a standard Windows
application



Adding the first window proc to this box also sets the message class to "Log all messages."

You can enter either the name of the object that processes messages for the window (select the Window Proc button) or a handle value (select the Handle button). Enter as many routine names or handle values as necessary to track messages for your windows.

It's easier to indicate the window by the name of the routine that processes its messages (for example, *WndProc*) because you can enter a routine name any time after loading your program.

If you prefer to use a handle variable name, you must first step through the program past the line where the handle variable is assigned a handle. (Use the *F7* or *F8* key to single-step through the program.) If you try to enter the variable name before stepping past its assignment statement, TDW will not let you.

Selecting a window for an ObjectWindows application

If you're debugging an ObjectWindows application and you select **View | Windows Messages**, by default you see the standard Windows Messages dialog box in Figure 11.1. This dialog box works the same for ObjectWindows programs as for standard Windows programs, except that you can't use a Windows procedure name. Instead, you must use the handle to the window object for the window whose messages you want to log or break on.

Obtaining a window handle

Before you can use the handle of a window object, you must run your program past the point where the handle is initialized. You can use a number of techniques.

- It's simplest just to run your application and exit back to TDW with *Ctrl-Alt-SysRq*.
- Another possibility is to set a breakpoint in a message-handling routine in your program (such as a routine that handles `WM_MOUSEMOVE` messages), run the program, and then perform the action in the window that triggers the breakpoint (for example, moving the mouse).
- If you're having major problems with the window itself (such as an unrecoverable application error—UAE—that comes up when the window is first displayed), you'll have to go to greater lengths to obtain the window handle.

Because the handle is initialized by the `ObjectWindows` function **CreateWindow** and this function executes after you initialize the window, you have to redeclare this function in the window class and then set a breakpoint on it to get the handle.

For example, the following code redeclares this function for the `TDODEMO` window class **ScribbleWindow**:

```
void ScribbleWindow::SetupWindow()
{
    TWindow::SetupWindow();
}
```

Next, position the cursor on the line after the initialization statement and press *F4* to run the program to the point where the handle of the window, dialog box, or control is initialized. In this example, you'd position the cursor on the closing brace of the function **SetupWindow**.

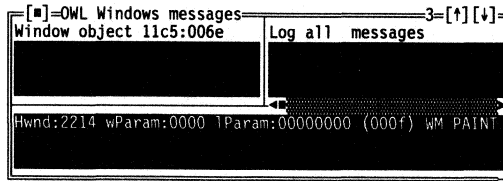
Once the handle is initialized and you've returned to TDW, you can obtain its value by choosing **Data | Inspect** and entering the name of the associated window object (in `TDODEMO`, `WinMain#MyApp.MainWindow`). Look for the data member `HWindow` and copy it into the Clipboard (press *Shift-F3*). You can then paste the *contents* of `HWindow` as a handle into the **Add** dialog box of the Window Messages window's top left pane (press *Shift-F4* in the dialog box's text entry box).

Specifying a window with `ObjectWindows` support enabled

See the file `TDWINST.TDW` for information on setting options in `TDWINST`.

If you run the TDW configuration program `TDWINST`, you can turn on support in TDW for `ObjectWindows` window messages. With this option on, you can use the names of windows objects as they're declared in your application. Choosing **View | Windows Messages** with the `OWL` option on displays the following screen:

Figure 11.3
The Windows Messages
window with ObjectWindows
support enabled



Add...
Remove
Delete all

Before you can log messages, you must first indicate which window, dialog box, or dialog control you're logging messages for. You do this in the top left pane, the Window Selection pane. This pane's local menu (activated by pressing *Alt-F10*) lets you add a window object, delete a window object, or delete all window objects.

Adding a window with ObjectWindows support enabled

Before adding a window object, you must run your program past the point where the window object is initialized. Typically, the object is initialized in a statement like the one in the following function definition from TDODEMO:

```
void CScribbleApplication::InitMainWindow()
{
    MainWindow = new ScribbleWindow(NULL, Name);
}
```

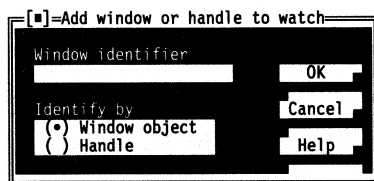
Position the cursor on the line after the initialization statement and press *F4* to run the program to the point where the window, dialog box, or control is initialized. In this example, you'd position the cursor on the closing brace of the function.

Once the window object is initialized, you can add it to the Window Selection pane. To add the object, either choose **Add** from the Window Selection pane local menu or begin typing the object's name in the pane. Either method brings up the Add Window dialog box.



If you're not in the routine where the object is declared, you have to override scope to access it. For example, in TDODEMO, *MainWindow* is a member of *MyApp* (because *MyApp* is of type **CScribbleApplication**, which is derived from **TApplication**, which has a data member called *MainWindow*). However, since *MyApp* is declared in function **WinMain**, unless you're in that function, you can't access *MyApp*, either. Therefore, the scope override that's guaranteed to work in this module is `WinMain#MyApp.MainWindow`.

Figure 11.4
The Add Window dialog box
with ObjectWindows support
enabled



Adding the first object to this pane also sets the message class to "Log all messages."

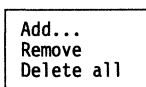
You can enter either the name of the object that processes messages for the window, dialog box, or control (select the Window Object button) or a handle value (select the Handle button). Enter as many object names or handle values as necessary to track messages for your windows.

Deleting a window selection

Deleting a window selection from the Window Selection pane works the same for both types of applications. To delete, move the cursor to the item, then either bring up the local menu and choose **Remove** or press the *Delete*, *Ctrl-Y*, or *Ctrl-R* key.

To delete all selections, choose **Delete All** from the local menu.

Specifying a message class and action



The top right pane is the Message Class pane. Its local menu, identical to that of the Window Selection pane, allows you to add a message class, remove a message class, or delete all classes you have added.

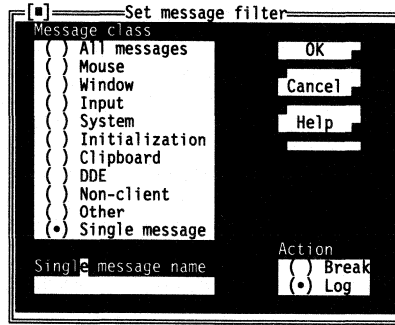
You must specify a window procedure or handle in the Window Selection pane before you can add a message class in this pane.

If you don't indicate a specific message or class of messages to watch, TDW watches all messages sent to the window procedure or handle.

Adding a message class

To add a message class, choose **Add** from the Message Class pane local menu. TDW displays the following dialog box:

Figure 11.5
The Set Message Filter dialog
box



The Set Message Filter dialog box prompts you both for a message class to track and an action to be performed when a message in that class is received.

TDW by default logs all messages starting with `WM_`. Because so many messages come in, you'll probably want to narrow the focus by selecting one of the classes in the Message Class list. You can add only one class at a time, so if you need to track messages from multiple classes, you have to use the **Add** option for each class you want to set.

The following table describes the message classes:

Table 11.1
Windows message classes

Message class	Description
All Messages	All window messages
Mouse	Messages generated by a mouse event (for example, <code>WM_LBUTTONDOWN</code> and <code>WM_MOUSEMOVE</code>)
Window	Messages from the window manager (for example, <code>WM_PAINT</code> and <code>WM_CREATE</code>)
Input	Messages generated by a keyboard event or by the user's accessing a System menu, scroll bar, or size box (for example, <code>WM_KEYDOWN</code>)
System	Messages generated by a system-wide change, (for example, <code>WM_FONTCHANGE</code> and <code>WM_SPOOLERSTATUS</code>)
Initialization	Messages generated when an application creates a dialog box or a window (for example, <code>WM_INITDIALOG</code> and <code>WM_INITMENU</code>)
Clipboard	Messages generated when one application tries to access the Clipboard of a window in another application (for example, <code>WM_DRAWCLIPBOARD</code> and <code>WM_SIZECLIPBOARD</code>)

Table 11.1: Windows message classes (continued)

DDE	Dynamic Data Exchange messages, generated by applications' communicating with one another's windows (for example, WM_DDE_INITIATE and WM_DDE_ACK)
Non-client	Messages generated by Windows to maintain the non-client area of an application window (for example, WM_NCHITTEST and WM_NCCREATE)
Other	Any messages that don't fall into any of the other categories, such as owner draw control messages and multiple document interface messages
Single Message	Any single message you want to log or break on

To track a single message, choose **Single Message** and enter the message name or the message number as a decimal number. If you enter a message name, be sure to use all capital letters.

The default action is to put the messages in the log. The other action you can perform, having the program break when it receives a message, is equivalent to setting a breakpoint for a message.

For example, if you want to track the WM_PAINT message and have the program stop every time this message is sent to a window you chose in the Window Selection pane, do the following:

1. Select the top right pane, the Message Class pane.
2. Bring up the local menu, then choose **Add**.
3. From the dialog box, select **Break** from the Action radio buttons and **Single Message** from the Message Class radio buttons.
4. Enter WM_PAINT in the Message Name input box, then press *Enter*.

Figure 11.1 on page 158 shows how the Windows Messages window looks after you have made these selections and a message has come in.

Deleting a message class

To delete a message class, move the cursor to the item, then either bring up the local menu and choose **Remove** or press one of the following keys: *Delete*, *Ctrl-R*, or *Ctrl-Y*.

To delete all classes, choose **Delete All** from the local menu or press *Ctrl-D*.

The default class, Log all messages, appears after you have deleted all classes. You cannot delete this class using **Remove** or **Delete All** command.

Window message tips

- ⇒ If you're displaying messages for more than one window, do not log all messages. Instead, log specific messages or a specific message class for each window. If you log all messages, the large number of messages being transferred between Windows and TDW might cause your system to hang.
- ⇒ When setting a break on Mouse class messages, be aware that a *mouse down* message must be followed by a *mouse up* message before the keyboard becomes active again. This restriction means that when you return to the application, you might have to press the mouse button several times to get Windows to receive a *mouse up* message. You'll know that Windows has received the message when you see it in the lower pane of the Windows Messages window.
- ⇒ If you enter a handle name but indicate that it's a procedure, TDW accepts your input and doesn't complain. However, when you run your program, TDW does not log any messages. If TDW isn't logging messages after you've set a handle, reenter the handle and be sure to select the Handle button.

Viewing messages

Send to log window No Erase log

Window messages show up in the lower pane of the Windows Messages window. This pane can hold up to 200 messages.

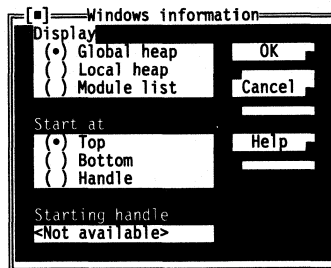
If you want to save the messages to a file, you have to open a log file for the Log window (use **View | Log File**, then choose **Open Log File** from the local menu). Then switch back to the Messages pane and change the **Send To Log Window** entry on the local menu to *Yes*.

If you want to clear the pane of all messages, choose **Erase Log** from the local menu. Any messages written to the Log window will not be affected by this command.

Obtaining memory and module lists

To list the contents of the global or local heap or the modules for your Windows program, first bring up the Log window with **View | Log**, then access the local menu. The last command on the Log window local menu is **Display Windows Info**. Choosing that command displays the Windows Information dialog box, from which you can pick the type of list you want to display and where to start the list.

Figure 11.6
The Windows Information dialog box



If you select the **Global Heap** option, you can choose to display the list from top to bottom, from bottom to top, or from a location indicated by a starting handle.

A starting handle is the name of a global memory handle set in your application by a call to a Windows memory allocation routine like *GlobalAlloc*. Picking a starting handle causes TDW to display the object at that handle as well as the next four objects that follow it in the heap.

Listing the contents of the global heap

The *global heap* is the global memory area Windows makes available to all applications. If you allocate resources like icons, bit maps, dialog boxes, and fonts, or if you allocate memory using the *GlobalAlloc* function, your application is using the global heap.

To see a list of the data objects in the global heap, select the Global Heap radio button in the Windows Information dialog box, then choose OK. The data objects will be listed in the Log window.



Because this list is likely to exceed the number of lines the Log window can write (the default is 50 lines), you should either write the contents to a log file (use the Log window local menu) or increase the number of lines the Log window can use (use TDWINST). The maximum number of lines you can set is 200.

The following table shows two lines of sample global heap output followed by an explanation of each field in the sample output:

Table 11.2
Format of a global heap list

Sample global heap output	
0EC5	00000040b PDB (0F1D)
053E (053D)	00002DC0b GDI DATA MOVEABLE LOCKED=00001 PGLOCKED=0001
Field	Description
0EC5 053E (053D)	Either a handle to the memory object, expressed as a 4-digit hex value, or the word <i>FREE</i> , indicating a free memory block. A memory selector pointing to an entry in the global descriptor table. The selector isn't displayed if it's the same value as the memory handle.
00000040b 00002DC0b	A hexadecimal number representing the length of the segment in bytes.
PDB GDI	The allocator of the segment, usually an application or library module. A PDB is a process descriptor block, also known as a program segment prefix (PSP).
(0F1D)	A handle indicating the owner of a PDB.
DATA	The type of memory object. The types are DATA Data segment of an application or DLL CODE Code segment of an application or DLL PRIV Either a system object or global data for an application or DLL
MOVEABLE	A memory allocation attribute. An object can be <i>FIXED</i> , <i>MOVEABLE</i> , or <i>DISCARDABLE</i> .
LOCKED=00001	For a moveable or moveable-discardable object, the number of locks on the object that have been set using either the <i>GlobalLock</i> or <i>LockData</i> function.
PGLOCKED=0001	For 386 Enhanced mode, the number of page locks on the object that have been set using the <i>GlobalPageLock</i> function. With a page lock set on a memory object, Windows can't swap to disk any of the object's 4-kilobyte pages.

Listing the contents of the local heap

The local heap is a private memory area for the application. It is not accessible to other Windows applications, including other instances of the same application.

A program doesn't necessarily have a local heap. Windows creates a local heap if the application uses the *LocalAlloc* function.

To see a list of the data objects in the local heap, select the Local Heap radio button in the Windows Information dialog box, then choose OK. The data objects will be listed in the Log window.



The list can easily exceed the default length of the log window. See the caution in the previous global heap section (page 166) about using a log file or increasing the number of lines that can be written in the Log window.

The following table shows a typical line of local heap output followed by an explanation of its format:

Table 11.3
Format of a local heap list

Local heap output	
Field	Description
05CD: 0024 BUSY (10AF)	
05CD:	The object's offset in the local data segment
0024	The length of the object in bytes
BUSY	The disposition of the memory object, as follows: FREE An unallocated block of memory BUSY An allocated object
(10AF)	A local memory handle for the object

Obtaining a list of modules

To see a list of the task and DLL modules that have been loaded by Windows, select the Module List radio button in the Windows Information dialog box, then choose OK. The modules will be listed in the Log window.



The list can easily exceed the default length of the log window. See the caution in the global heap section (page 166) about using a log file or increasing the number of lines that can be written in the Log window.

Table 11.4
Format of a Windows module
list

The following table shows three sample lines of a module list followed by an explanation of the last line in the list:

Sample module list output	
0985	TASK TDW C:\TD\TDW.EXE
0E2D	DLL WINDBG C:\WIN3\WINDBG.DLL
0EFD	TASK GENERIC C:\TD\GENERIC.EXE
Field	Description
0EFD	A handle for the memory segment, expressed as a 4-digit hex value.
TASK	The module type. A module can be either a task or a DLL.
GENERIC	The module name.
C:\TD\GENERIC.EXE	The path to the module's executable file.

Debugging dynamic link libraries (DLLs)

TDW can load a DLL that doesn't have a symbol table, but only into a CPU window.

A DLL is a library of routines and resources that is linked to your Windows application at runtime instead of at compile time. This runtime linking allows multiple applications to share a single copy of routines, data, or device drivers, thus saving on memory use. When an application that uses a DLL starts up, if the DLL is not already loaded into memory, Windows loads it in so the application can access the DLL's entry points.

When you load an application into TDW that has DLLs linked into it, TDW determines which of these DLLs, if any, have symbol tables (were compiled with the debugging option turned on) and tracks these DLLs for you. If, during execution of your application, TDW encounters a call to an entry point for one of these DLLs, TDW loads the symbol table and source for that DLL and positions the module line marker at the beginning of the DLL routine called by your application. The DLL is then available in the Module window just as your application source code was.

When the DLL routine exits, if possible, TDW reloads your application's source code and positions the line marker on the next statement after the call to the DLL entry point.

- ➡ If you are tracing through the program using *F7* or *F8*, it might not be possible for TDW to return you to the calling routine in your program because the DLL might return through a Windows function call. In this case, your program just runs as though you had pressed *F9*. This behavior is common in DLL startup code. To

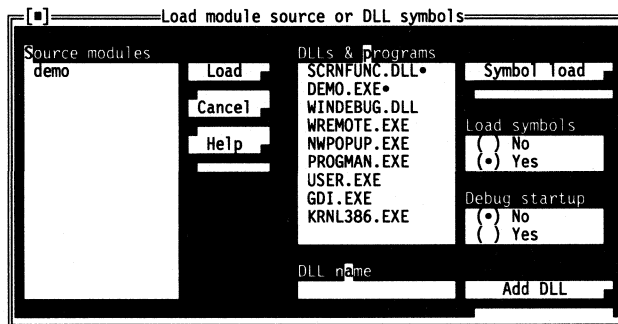
force a return to your application, before tracing in your application to the DLL call, set a breakpoint in your application on the line after the call to the DLL. When debugging DLL startup code, set the breakpoint on the first line of your application.

Because so much of DLL debugging is automatic with TDW, you never have to specify which DLLs to load. However, you might want to perform other tasks, such as:

- Adding a DLL to the list of DLLs
- Setting breakpoints, watches, and so on, in a DLL
- Specifying which DLLs TDW is *not* supposed to load symbols for
- Debugging DLL startup code

To perform any of these tasks, you have to access the Load Modules or DLLs dialog box by using the **View | Modules** command. (Pressing *F3* will also bring up this dialog box.)

Figure 11.7
The Load Modules or DLLs
dialog box



Using the Load Modules or DLLs dialog box

This dialog box enables you to do two things:

- Change to another source module of your application
- Perform operations on DLLs and .EXE files (such as loading in a symbol file and source file)

Changing source modules

If you're debugging an application consisting of multiple source modules linked into one .EXE file and you need access to a module of the application other than the one currently in the Module window, you can bring up the Load Modules or DLLs dialog box and pick one of the modules in the list on the left, the Source Modules list.



To pick a module, highlight it, and then either press *Enter*, click twice with the mouse, or choose the Load button. TDW displays the Module window with the new source module in it.

Working with DLLs and programs

When you're debugging an application that has one or more DLLs associated with it (as does any Windows application) and you bring up the Load Modules or DLLs dialog box, you see in the DLLs & Programs list (the list on the right) a list of DLLs and .EXE files (as well as all the .DRV and .FON files currently loaded into Windows). This list includes all DLL and .EXE files Windows currently has loaded, as well as all DLLs that get started when your application starts up. It does not include any DLLs your application starts by using a **LoadLibrary** call unless one of these DLLs is already loaded by your program or by Windows.

The items at the top of this list, marked on the right with an oval, are your application's .EXE file and the DLLs your application calls. If you make no changes, TDW automatically attempts to load in the symbol table and source for each marked DLL whenever your application makes a call to that DLL. In addition, TDW automatically attempts to load the symbol table and source of any DLL your application starts with a **LoadLibrary** call, even though the DLL might not appear on the list at first. (It will appear there after TDW loads it.)

The buttons to the right of this list perform operations on the DLL or application you have highlighted. The text entry box underneath the list lets you add a DLL to the list. You can use these features as follows:

Table 11.5
DLLs & Programs list dialog
box controls

Button	Description
Symbol load	Load in the symbol table and source files for the DLL or application, regardless of the Load Symbols setting. This command overrides the Load Symbols setting and changes the contents of the Module Window so you can set breakpoints, window messages, and so on for the DLL or application.

Table 11.5: DLLs & Programs list dialog box controls (continued)

Load symbols (No/Yes)	<p>Choose whether to load the DLL symbol table and source when the application makes a call to the DLL. You might use this option to prevent TDW from loading the symbol table and source of a DLL that you don't need to debug. The default setting is <i>Yes</i>.</p> <p>Choosing Yes puts an oval next to the DLL name.</p> <p>When you reload a program, Load Symbols is set to Yes for all DLLs and modules, even for DLLs or modules that were previously set to No.</p>
Debug startup (No/Yes)	<p>Choose whether to debug startup code for the DLL. The default setting is No.</p> <p>Choosing Yes puts double exclamation marks (!!) next to the module or DLL.</p> <p>These buttons are used for DLLs only. To debug application startup code, start TDW with the -I command-line option.</p>
DLL Name	<p>Enter the name of a DLL that isn't on the DLLs & Programs list so you can add it to the list. You can use any file extension you want. Adding a DLL to the list enables you to use one of the previous three commands on it. You can use a full path name if necessary.</p>
Add DLL	<p>Add the DLL in the text entry box to the DLLs & Programs list. Any DLL you add manually has both Load Symbols and Debug Startup set to Yes.</p>

Adding a DLL to the DLLs & Programs list

Before you can set debug options, debug DLL startup code, or prevent TDW from loading a DLL's symbol table and source, the DLL must first be in the DLLs & Programs list. A DLL accessed by your application might not be in this list because, just after your application loads, TDW only knows about DLLs that are linked into the startup code of your application. Your application can also start a DLL explicitly by using the Windows **LoadLibrary** function; TDW won't know about it until your application calls **LoadLibrary**.



There are two different types of startup code mentioned in this section: your application's startup code and DLL startup code.

Some DLLs are started in your application's startup code. When your application starts a DLL, the DLL's startup code is then executed. There are also two types of DLL startup code, explained later on page 173.

If you want to add a DLL to the DLLs & Programs list, bring up the Load Modules or DLLs dialog box (press **F3** or choose **View | Modules**), move to the DLL **Name** text entry box, enter the name of the DLL (enter the full path if necessary), then press the **Add DLL** button to add it to the list.

Setting debug options in a DLL

If you want to set breakpoints or watches or some other debug option for a DLL, bring up the Load Modules or DLLs dialog box (press **F3** or choose **View | Modules**), highlight the DLL on the DLLs & Programs list, then choose **Symbol Load** to bring up the DLL in a Module window. Once you're in the Module window, you can perform your operations on the DLL.

Controlling TDW's loading of DLL symbol tables

By default, TDW loads in the symbol table and source of every DLL that your application accesses, but only if the DLL has a TDW-compatible symbol table. A DLL has a symbol table compatible with TDW if it was compiled with debugging information turned on and the compiler was a Borland language compiler.

Because it takes time to load in a DLL's symbol table and then load in the original application's symbol table once the DLL routine has finished, you might want to disable TDW's default operation for DLLs you don't want to debug. To prevent TDW from loading a DLL's symbol table, bring up the Load Modules or DLLs dialog box (press **F3** or choose **View | Modules**), find the DLL on the DLLs & Programs list, highlight it, and then push the **Load Symbols No** button.

Debugging DLL startup code

By default, TDW does not debug DLL startup code and only loads a DLL's symbol table when your application makes a call to a DLL entry point. TDW then brings up the Module or CPU window with the current line marker at the beginning of the DLL routine called by the application.

TDW debugs DLL startup code if you tell it to. You can use TDW to debug either of two types of DLL startup code:

What kind of startup code are you debugging?

- The initialization code immediately following *LibMain* (the default)

- The assembly-language code linked into the DLL that does initial startup and contains emulated math packages for the size model the DLL is running in (selected by starting TDW with the **-I** command-line option)

After you specify startup debugging for one or more of the DLLs in your application, TDW loads in the symbol table for each DLL either when your application startup code starts the DLL or when your application makes a **LoadLibrary** call.

Is your application already loaded?

If you try to load your application and then set startup debugging, TDW might not behave as you expect, since some or all of the DLLs might already have been loaded. Therefore, you should set startup debugging either

- By setting the DLLs before you load your application
- By loading your application, indicating the DLLs for startup debugging, and then restarting your application (**Ctrl-F2** or **Run | Program Reset**)

Doing startup code debugging

With all these preliminaries in mind, use the following steps to specify startup debugging for one or more DLLs and to debug those DLLs' startup code:

1. Bring up the Load Modules or DLLs dialog box (press **F3** or choose **View | Modules**).
2. If no program is loaded, skip to step 5. Otherwise, find a DLL on the DLLs & Programs list and highlight it.
3. Select the Debug Startup **Yes** button.
4. Repeat steps 2 and 3 until you've set startup debugging for all DLLs you're interested in.
5. If a DLL you want isn't on the list or there are no DLLs on the list (because you haven't loaded your application yet), use the **DLL Name** text entry box to enter each DLL name and add it to the list using the **Add DLL** button.
6. When you've set all the DLLs for which you want to debug startup code, choose either **File | Load** to load in your application (if you haven't loaded it yet) or **Run | Program Reset** (**Ctrl-F2**) to reload your application (if you loaded it before setting startup debugging).
7. Before you run the application, you should set breakpoints to guarantee that the DLLs will return to your application after

the startup code executes. With your application's source code in the Module window,

- a. Set a breakpoint on the first line of your application.
 - b. If you're debugging startup code for any DLLs loaded with **LoadLibrary** calls, set a breakpoint on the first line of code after each of these calls.
8. As your application starts each DLL, TDW puts you either in the Module window at the DLL's *LibMain* (the default) or in the CPU window at the start of the assembly code listing for the startup library (because you ran TDW using the **-I** option).
 9. When you've finished debugging startup code for a DLL, press **F9** to run through the end of the startup code and return to the application. If you've specified any more DLLs for startup code debugging, TDW displays startup code for those DLLs when your application starts them.



Be sure to run to the end of a DLL's startup code before reloading the current application or loading a new one. If you don't, the partially executed DLL startup code might cause Windows to hang, forcing you to reboot.

Converting memory handles to addresses

Windows uses memory handles instead of addresses for memory objects because it performs its own memory management and can change the physical location in memory of an object. If you need the actual address referred to by a memory handle, you can use the TDW built-in typecast symbols **lh2fp** (for a local handle) and **gh2fp** (for a global handle) to dereference the handle.

You use these typecasting symbols in TDW just as you use the regular C++ typecasting symbols for pointers. For example, you could cast the local memory handle **hLocalMemory** using two methods:

- You could use the **Data | Inspect** window to evaluate the expression `(lh2fp)hLocalMemory`.
- You could use the **Type Cast** command in the Inspector local window and enter `lh2fp` as the type.

In either case, the expression evaluates to the first character of the memory block pointed to by `hLocalMemory`.

You could also use either of these techniques to do a more complicated cast—for example, a two-stage cast from a handle into a character pointer into a pointer to the data in memory, as follows:

```
(Mystruct far *) (lh2fp) hLocalMemory
```


Assembler-level debugging

For more information on assembler-level debugging, see the file DOC\ASMDEBUG.TDW in the language compiler directory on your hard disk.

This chapter is for programmers who need to take a lower-level look at their code. It gives a brief introduction to the CPU window and the six panes of that window. You can also get online Help information about any pane of this window and its local menu by positioning the cursor in the pane and pressing *F1*.

When source debugging isn't enough

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

Sometimes, however, you need information you can't get from the source code Module window, such as

- looking at the contents of an area in memory referenced by a protected-mode selector
- looking at the exact instructions that the compiler generated for a line of source code, as well as the contents of the stack and CPU registers
- tracing through Windows code to find where your program stopped

To perform any of these functions, you have to use the CPU window. In addition, it helps to be familiar with Windows' use of

memory and to have knowledge of both the 80x86 family of processors and the machine instructions the compiler generates for your source code. Because many excellent books are available about the internal workings of the CPU, we won't go into that in detail here. You can quickly learn how the compiler turns your source code into machine instructions by looking at the instructions generated for each line of source code.

The CPU window

The CPU window shows you the entire state of the CPU. You can

- examine and change the bits and bytes that make up your program's code and data
- access the contents of any area of memory referenced by a selector
- use the built-in assembler in the Code pane to patch your program temporarily by entering instructions exactly as you would type assembler source statements
- access the underlying bytes of any data structure, display them in a number of formats, and change them

Figure 12.1
The CPU window

The screenshot shows the CPU window for a remote CPU at address 80386. The window is divided into several panes:

- Code Pane:** Displays assembly instructions with their addresses and selectors. The current instruction is at address 015F:57, which is a `push di` instruction. The instruction at 0163:33F6 is highlighted with a mouse cursor.
- Registers Pane:** Shows the state of various registers:

ax	FF15	c=0
bx	08DE	z=1
cx	0015	s=0
dx	08DE	o=0
si	08C0	p=1
di	08DE	a=0
bp	1E2A	i=1
sp	1E1E	d=0
ds	12B5	
es	12B5	
ss	12B5	
cs	12AD	
ip	0158	
- Memory Pane:** Shows memory contents at various addresses:

ds:0000	00 00 00 00 00 00 50 1E	PA
ds:0008	00 00 C2 0A 2C 1E 2C 1E	
ds:0010	00 00 08 00 00 00 95 12	
ds:0018	B6 12 00 00 80 01 00 \$	
- Status Bar:** Shows the current stack pointer: `ss:1E20 0001` and `ss:1E1E 00A2`.

Open a CPU window by choosing **View | CPU** from the menu bar. Depending on what you're viewing in the current window, the new CPU window comes up positioned at the appropriate code, data, or stack location, thus providing a convenient method for taking a low-level look at the code, data, or stack location your cursor is currently on.

The following table shows where your cursor is positioned when you choose the **CPU** command:

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

TDW also automatically puts you in the CPU window if TDW regains control from your application and the current code being executed is Windows code or DLL code that has no debugging information.



The CPU window has six panes. To go from one pane to the next, press *Tab* or *Shift-Tab*, or click the pane with your mouse. The line at the top of the CPU window shows what processor type you have (8086, 80286, 80386, or 80486).

- The top left pane (Code pane) shows the disassembled program code intermixed with the source lines.
- The second top pane (Register pane) shows the contents of the CPU registers.
- The top right pane (Flags pane) shows the state of the eight CPU flags.
- The middle left pane (Selector pane—below the Code pane) shows all Windows selectors and indicates the general contents of each.
- The bottom left pane (Data pane—below the Selector pane) shows a raw hex dump of any area of memory you choose.
- The bottom right pane (Stack pane) shows the contents of the stack.

As with all windows and panes, pressing *Alt-F10* pops up the pane's local menu. If control-key shortcuts are enabled, pressing the *Ctrl* key with the highlighted letter of the desired local menu command executes the command.

In the Code, Data, and Stack panes, you can press *Ctrl* ← and *Ctrl* → to shift the starting display address of the pane by 1 byte up or down. Pressing these keys is easier than using the **Goto** command if you just want to adjust the display slightly.

The Code pane

An arrow (▶) shows the current program location (CS:IP).

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

There are two ways of choosing an address:

- Use the local menu Goto, Origin, Follow, Caller, or Previous command.
- Position on a code selector in the Selector pane, then choose Examine to display the contents of the selector in the Code pane.

The left part of each disassembled line shows the address of the instruction. The address is displayed either as a hex segment and offset, or with the segment value replaced with the CS register name if the segment value is the same as the current CS register. If the window is wide enough (zoomed or resized), the bytes that make up the instruction are displayed. The disassembled instruction appears to the right.

If the highlighted instruction in the Code pane references a memory location, the memory address and its current contents are displayed on the top line of the CPU window. This feature lets you see both where an instruction operand points in memory and the value that is about to be read or written over.

The disassembler

The Code pane automatically disassembles and displays your program instructions. If an address corresponds to either a global symbol, static symbol, or a line number, the line before the disassembled instruction displays the symbol if the **Mixed** display mode is set to *Yes*. Also, if there is a line of source code that corresponds to the symbol address, it is displayed after the symbol.

Global symbols appear simply as the symbol name. Static symbols appear as the module name, followed by a 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.

When an immediate operand is displayed, you can infer its size from the number of digits: A byte immediate has 2 digits, and a word immediate has 4 digits.

The Register and Flags panes

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

The top right pane is the Flags pane, which shows the state of the eight CPU flags. The following table lists the different flags and how they are shown in the Flags pane:

Letter in pane	Flag name
c	Carry
z	Zero
s	Sign
o	Overflow
p	Parity
a	Auxiliary carry
i	Interrupt enable
d	Direction

You can use the local menu of the Register pane to increment or decrement a register by 1, to set the register to 0, to change the register, or to toggle between displaying the register as 16-bit or 32-bit values (requires an 80386 processor or greater).

The local menu of the Flags pane allow you to toggle the flag between 0 and 1.

The Selector pane

This pane shows a list of protected-mode selectors and indicates some information about each one.

A selector can be either valid or invalid. If valid, the selector points to a location in the protected-mode descriptor table corresponding to a memory address. If invalid, the selector is unused.

For a valid selector, the pane shows the following:

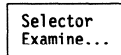
- if the contents are data or code
- if the memory area the selector references is loaded (present in memory) or unloaded (swapped out to disk)
- the length of the referenced memory segment in bytes

If the selector references a data segment, there's additional information on the access rights (Read/Write or Read only) and the direction the segment expands in memory (Up or Down).

The Selector pane local menu

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

You can use the local menu of the Selector pane to go to a new selector (the Selector command) or see the contents of the selector currently highlighted in the Selector pane (the Examine command). The contents display in either the Code pane or the Data pane, depending on their nature.



Selector Prompts you to type a selector to display in the pane. You can use full expression syntax to enter the selector. If you enter a numeric value, TDW assumes it is decimal unless you use the syntax of the current language to indicate that the value is hexadecimal.

For example, if the current language were C, you could type the hexadecimal selector value 7F as `0x7F`. For Pascal, you'd type it as `$7F`. You could also type the decimal value 127 in order to go to selector 7F.

Another method of entering the selector value is to display the CPU window and check the segment register values. If a register holds the selector you're interested in, you can enter the name of the register preceded by an underscore (`_`). For example, you could type the data segment register as `_DS`.

Examine Displays the contents of the memory area referenced by the current selector and switches focus to the pane where the contents are displayed. If the selector points to a code segment, the contents are displayed in the Code pane. If the contents are data, they're displayed in the Data pane.

The Data pane

This pane shows a raw display of an area of memory you've selected. The leftmost part of each line shows the address of the data displayed in that line. The address is displayed either as a hex segment and offset, or with the segment value replaced with one of the register names if the segment value is the same as that register. The Data pane matches registers in the following order: DS, ES, SS, CS.

Next, the raw display of one or more data items is displayed. The format of this area depends on the display format selected with the **Display As** local menu command. If you choose one of the floating-point display formats (**Comp**, **Float**, **Real**, **Double**, **Extended**), a single floating-point number is displayed on each line. **Byte** format displays 8 bytes per line, **Word** format displays 4 words per line, and **Long** format displays 2 long words per line.

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

There are two ways of choosing an address:

- Use the local menu **Goto**, **Follow**, or **Previous** command.
- Position on a data selector in the **Selector** pane, then choose **Examine** to display the contents of the selector in the **Data** pane.

The **Data** pane local menu lets you go to a new address, search for a character string, change bytes at the current cursor location, follow near or far pointer chains, restore a previous address, change how data appears in the window, and move, change, read, and write blocks of memory.

The Stack pane

An arrow (▶) shows the current stack pointer (SS:IP).

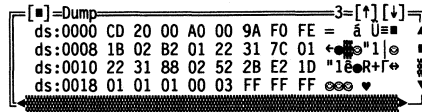
The **Stack** pane, in the lower right corner of the CPU window, shows the contents of the stack.

The commands in the local menu let you change positions in the stack and change values of words on the stack.

The Dump window

The Dump window, opened by choosing **View | Dump**, shows you a raw data dump of any area of memory. The Dump window shows you a raw data dump of any area of memory. It works much like the Data pane in the CPU window (see page 183), except that, when zoomed to full size, the Dump Window show twice as much data on a single line.

Figure 12.2
The Dump window

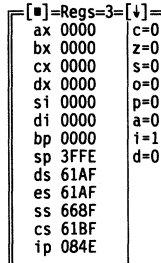


Typically, you use this window if you're in an Inspector window and you want to look at the raw bytes that make up the object you are inspecting. Use **View | Dump** to get a Dump window that's positioned to the data in the Inspector window.

The Registers window

The Registers window shows you the contents of the CPU registers and flags. It works like a combination of the Registers and Flags panes in the CPU window (see page 181).

Figure 12.3
The Registers window



You can perform the same functions from the local menu of the Registers window as you can from the local menus of the Registers pane and the Flags pane.

Command reference

Now that you've read about all the commands, here's a quick summary. This chapter lists and describes

- all the single-keystroke commands available on the function and other keys
- all the menu bar commands and the commands for the local menu of each window type
- keystrokes used in the two types of panes (those in which you enter text and those from which you select an item)
- keystrokes for moving and resizing windows

Hot keys

A hot key is a key that performs its action no matter where you are in the Turbo Debugger environment. The following table lists all the hot keys:

Table 13.1: The function key and hot key commands


Key	Menu command	Function
<i>F1</i>		Brings up context-sensitive help
<i>F2</i>	Breakpoints Toggle	Sets breakpoint at cursor position
<i>F3</i>	View Module	Module pick list
<i>F4</i>	Run Go to Cursor	Runs to cursor position
<i>F5</i>	Window Zoom	Zooms/unzooms current window
<i>F6</i>	Window Next Window	Goes to next window
<i>F7</i>	Run Trace Into	Executes single source line or instruction
<i>F8</i>	Run Step Over	Executes single source line or instruction, skipping calls
<i>F9</i>	Run Run	Runs program
<i>F10</i>		Invokes the menu bar, takes you out of menus
<i>Alt-F1</i>	Help Previous Topic	Brings up last help screen
<i>Alt-F2</i>	Breakpoints At	Sets breakpoint at an address
<i>Alt-F3</i>	Window Close	Closes current window or all Inspector windows
<i>Alt-F4</i>	Run Back Trace	Reverses program execution
<i>Alt-F5</i>	Window User Screen	Shows your program's screen
<i>Alt-F6</i>	Window Undo Close	Reopens the last-closed window
<i>Alt-F7</i>	Run Instruction Trace	Executes a single instruction
<i>Alt-F8</i>	Run Until Return	Runs until return from function
<i>Alt-F9</i>	Run Execute To	Runs to a specified address
<i>Alt-F10</i>		Invokes the window's local menu
<i>Alt-1—9</i>		Switch to numbered window 1 through 9
<i>Alt-Space</i>		Goes to the  (System) menu
<i>Alt-B</i>		Goes to the B reakpoints menu
<i>Alt-D</i>		Goes to the D ata menu
<i>Alt-E</i>		Goes to the E dit menu.
<i>Alt-F</i>		Goes to the F ile menu
<i>Alt-H</i>		Goes to the H elp menu
<i>Alt-O</i>		Goes to the O ptions menu
<i>Alt-R</i>		Goes to the R un menu
<i>Alt-V</i>		Goes to the V iew menu
<i>Alt-W</i>		Goes to the W indow menu
<i>Alt-X</i>	File Quit	Quits Turbo Debugger and returns you to DOS
<i>Alt =</i>	Options Macros Create	Defines a keystroke macro
<i>Alt -</i>	Options Macros Stop	Ends a macro recording
	Recording	
<i>Ctrl-F2</i>	Run Program Reset	Stops debug session and resets the program to start again
<i>Ctrl-F4</i>	Data Evaluate	Evaluates an expression
<i>Ctrl-F5</i>	Window Size/Move	Initiates window moving or resizing
<i>Ctrl-F7</i>	Data Add Watch	Adds a variable to the Watches window
<i>Ctrl →</i>		Shifts 1 byte up the starting address in a Code, Data, or Stack pane in a CPU window
<i>Ctrl ←</i>		Shifts 1 byte down the starting address in a Code, Data, or Stack pane in a CPU window

Table 13.1: The function key and hot key commands (continued)

Key	Menu command	Function
<i>Shift-F1</i>	H elp Index	Goes to the index for online help
<i>Shift-F3</i>	E dit C opy	Copies item at cursor or highlighted item to Clipboard
<i>Shift-F4</i>	E dit P aste	Pastes item from Clipboard to window or dialog box prompt
<i>Shift-Tab</i>		Moves cursor to previous window pane or dialog box item
<i>Shift →</i>		Moves cursor between the panes in a window (the pane in the direction of the arrow becomes the active pane)
<i>Shift ←</i>		
<i>Shift ↑</i>		
<i>Shift ↓</i>		
<i>Esc</i>		Closes an Inspector window, exits menus and dialog boxes
<i>Ins</i>		Starts text block or list selection (highlight); use arrow keys to highlight
<i>Tab</i>	W indow N ext Pane	Moves cursor to next window pane or dialog box item

Commands from the menu bar

You invoke the menu bar by pressing the *F10* key; you can then go directly to one of the individual menus by

- moving the cursor to the menu title and pressing *Enter*
- pressing the highlighted letter of the menu title

You can also open a menu directly (without first moving to the menu bar) by pressing *Alt* in combination with the first letter of the menu name you want.

The ≡ (System) menu

R epaint Desktop	Redisplays entire screen
R estore Standard	Restores standard window layout
A bout	Displays information about Turbo Debugger

The File menu

O pen	Loads a new program to debug
C hange Dir	Changes to new disk or directory
G et Info	Displays program information
S ymbol Load	Loads symbol table independent of .EXE file
Q uit	Returns to DOS

The Edit menu

C opy	Copies an item into the Clipboard
P aste	Pastes an item from the Clipboard into a window or dialog box prompt
C opy to Log	Copies the highlighted item or the item at the cursor to the Log window

The View menu

B reakpoints	Displays breakpoints
S tack	Displays procedure-calling stack
L og	Displays log of events and data
W atches	Displays variables being watched
V ariables	Displays global and local variables
M odule	Displays program source module
F ile	Displays disk file as ASCII or hex
C PU	Displays CPU instructions, data, stack
D ump	Displays raw data dump
R egisters	Displays CPU registers and flags
N umeric Processor	Displays coprocessor or emulator
E xecution History	Displays assembler code saved for backtracking
H ierarchy	Displays class list and hierarchy tree
W indows messages	Displays list of window messages for one or more windows in your application program
C lipboard	Displays the Clipboard window so you can see the items you've copied into the Clipboard.
A nother	
M odule	Makes another Module window
D ump	Makes another Dump window
F ile	Makes another File window

00P

The Run menu

Run	Runs your program without stopping
Go To Cursor	Runs to current cursor location
Trace Into	Executes one source line or instruction
Step Over	Traces, skipping calls
Execute To	Runs to specified address
Until Return	Runs until procedure returns
Animate	Continuously steps your program
Back Trace	Reverses program execution for one source line or instruction
Instruction Trace	Executes a single instruction
Arguments	Sets program command-line arguments
Program Reset	Reloads current program

The Breakpoints menu

Toggle	Toggles breakpoint at cursor
At	Sets breakpoint at specified address
Changed Memory Global	Sets global breakpoint on memory area
Expression True Global	Sets global breakpoint on expression
Hardware Breakpoint	Sets a hardware breakpoint
Delete All	Removes all breakpoints

The Data menu

Inspect	Inspects a data object
Evaluate/Modify	Evaluates an expression
Add Watch	Adds variable to Watches window
Function Return	Inspects current routine's return value

The Options menu

Language	Sets expression language
Macros	
Create	Defines a keystroke macro
Stop Recording	Ends the recording session

Remove	Removes a keystroke macro
Delete All	Removes all keystroke macros
Display Options	Lets you set screen display options (screen swapping, size, tabs)
Path for Source	Directory list for source files
Save Options	Saves options, screen layout, and macros to disk
Restore Options	Restores options from disk

The Window menu

Zoom	Zooms window to full screen size and back
Next	Activates successive windows open onscreen
Next Pane	Goes to the next pane in a window
Size/Move	Moves window or changes its size
Iconize/Restore	Reduces window to a small symbol or restores it
Close	Closes window
Undo Close	Reopens the last window closed
Dump Pane to Log	Writes current pane to Log window
User Screen	Displays your program output
<i>Numbered window list</i>	A numbered list of up to 9 open windows to activate
Window Pick	Displays a menu of open menus if more than 9 are open onscreen

The Help Menu

Index	Goes to the index for online help
Previous Topic	Brings up last help screen
Help on Help	Accesses online help on the help system

The local menu commands

Each type of window and each pane within a window has a different local menu.

You invoke the local menu for the current window by pressing **Alt-F10**. If control-key shortcuts are enabled, you can go directly to one of the individual menu items by pressing the **Ctrl** key in combination with the highlighted letter of the item you desire.

(Use the installation program TDWINST to enable control-key shortcuts, if they've been disabled.)

The menus in this section are arranged in alphabetical order to make lookups easier.

The following sections describe the local menu for each window and pane.

Most panes have shortcuts to commonly used commands on their local menu. In the following sections, these special keys are highlighted in the menu commands. In many panes, the *Enter* key is a shortcut to examining or changing the currently highlighted item. The *Del* key often invokes the local menu command that deletes the highlighted item. Some panes let you start typing letters or numbers without first invoking a local menu command. In these cases, the dialog box for one of the local menu items pops up to accept your input.

Breakpoints window

The Breakpoints window has two panes: the List pane on the left and the Detail pane on the right. Only the List pane has a local menu.

Set Options	Sets breakpoint actions, conditions, pass count, and enable/disable
Add	Adds a new breakpoint
Remove	Removes highlighted breakpoint
Delete All	Deletes all breakpoints
Inspect	Looks at code where breakpoint is set
Group	Work with groups of breakpoints

Del is the shortcut for **Remove** in this window.

The CPU window menus

The CPU window has six panes, each with a local menu: the Code pane, the Data pane, the Selector pane, the Stack pane, the Register pane, and the Flags pane.

Code pane

Goto	Displays code at new address
Origin	Displays code at CS:IP
Follow	Displays code at JMP or CALL target
Caller	Displays code at calling procedure
Previous	Displays code at last address
Search	Searches for instruction or bytes

View Source	Switches to Module window
Mixed	Mixes source code with disassembly: <i>No/Yes/Both</i>
New CS:IP	Sets CS:IP to execute at new address
Assemble	Assembles instruction at cursor
I/O	Brings up I/O menu
In Byte	Reads a byte from an I/O location
Out Byte	Writes a byte to an I/O location
Read Word	Reads a word from an I/O location
Write Word	Writes a word to an I/O location

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

Selector pane

Selector	Lets you enter a new selector to go to
Examine	Displays contents of memory area referenced by selector in code pane or data pane, depending on type of contents

Data pane

Goto	Displays data at new address
Search	Searches for string or data bytes
Next	Searches again for next occurrence
Change	Changes data bytes at cursor address
Follow	
Near Code	Sets Code pane to the near address under the cursor
Far Code	Sets Code pane to the far address under the cursor
Offset to Data	Sets Data pane to the near address under the cursor
Segment:Offset to Data	Sets Data pane to the far address under the cursor
Base Segment:0 to Data	Sets Data pane to start of segment that contains the address under the cursor
Previous	Displays data at last address
Display As	
Byte	Displays hex bytes
Word	Displays hex words
Long	Displays hex 32-bit long words
Comp	Displays 8-byte Pascal comp integers
Float	Displays short (4-byte) floating-point numbers (float)

Real	Displays 6-byte floating-point numbers (Pascal real)
Double	Displays 8-byte floating-point numbers (double)
Extended	Displays 10-byte floating-point numbers (long double)
Block	
Clear	Sets memory block to zero
Move	Moves memory block
Set	Sets memory block to value
Read	Reads from file to memory
Write	Writes from memory to file

Typing any character is a shortcut for the **C**hange local menu command in this pane.

Flags pane

Toggle	Sets or clears highlighted flag
---------------	---------------------------------

Pressing *Enter* or *Spacebar* is a shortcut for the **T**oggle local menu command in this pane.

Register pane

Increment	Adds one to highlighted register
Decrement	Subtracts one from highlighted register
Zero	Clears highlighted register
Change	Sets highlighted register to new value
Registers 32-bit	Toggles 32-bit register display: <i>No/Yes</i>

Typing any character is a shortcut for the **C**hange local menu command in this pane.

Stack pane

Goto	Displays stack at new address
Origin	Displays data at SS:SP
Follow	Displays code pointed to by current item
Previous	Restores display to last address
Change	Lets you edit information

Typing any character is a shortcut for the **C**hange local menu command in this pane.

Dump window

The Dump window is identical to the Data pane of the CPU window. Its local menu is identical to the Data pane local menu.

The Execution History window menus

The Execution History window has two panes, each with a local menu: the Instructions pane and the Keystroke Recording pane.

Instructions pane

The Instructions pane shows instructions already executed that you can examine or undo.

Inspect	Takes you to the highlighted command
Reverse Execute	Reverses program execution to the instruction highlighted in the Instructions pane
Full History	Enables (Yes) or disables (No) reverse execution

File window

The File window shows the contents of the disk file as hex bytes or as an ASCII file.

Goto	Displays line number or hex offset
Search	Searches for string or data bytes
Next	Searches again for next occurrence
Display As	Sets file display mode: ASCII/Hex
File	Switches to view new file

Typing any character is a shortcut for the **S**earch local menu command.

Log window menu

The Log window shows messages sent to the log and allows you to list Windows memory and module information.

Open Log File	Starts logging to a file
Close Log File	Stops logging to a file

Logging	Toggles logging: <i>No/Yes</i>
Add Comment	Writes user comment to log
Erase Log	Clears all log messages
Display Windows Info	Displays the Windows Information dialog box, from which you can pick the type of list (global heap, local heap, or module) you want to display

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

Module window

The Module window shows the source file for the program module.

Inspect	Shows contents of variable under cursor
Watch	Adds variable under cursor to watch list
Module	Changes to display different module
File	Changes to display different file
Previous	Displays last module and position
Line	Displays source at line in module
Search	Searches for text string
Next	Searches for next occurrence of string
Origin	Displays current program location
Goto	Shows source or instructions at address

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

Windows Messages window

The Windows Messages window has three panes: the Window Selection pane, the Message Class pane, and the Messages pane.

Window Selection pane

These are the local menu commands in this pane:

Add	Adds a window name or handle value
Remove	Removes the selected window
Delete all	Deletes all window selections

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

The *Del* key or the *Ctrl-Y* key combination is a shortcut for the **Remove local menu** command.

Message Class pane These are the local menu commands in this pane:

Add	Adds a message class or single message
Remove	Removes the selected message class or single message
Delete all	Deletes all message class or single message selections

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

The *Del* key or the *Ctrl-Y* key combination is a shortcut for the **Remove local menu** command.

Messages pane These are the local menu commands in this pane:

Send to log window	Sends all messages received to the log window so they can be saved in a log file
Erase log	Erases all messages in the pane

Clipboard window

The Clipboard window shows you all the items you've copied into the Clipboard. It has a single pane with the following local menu commands:

Inspect	Puts you in the window an item was copied from so you can inspect the item
Remove	Removes the highlighted item
Delete All	Deletes all Clipboard items
Freeze	Freezes the highlighted item at its current value

Numeric Processor window

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

Register pane These are the local menu commands in this pane:

Zero	Clears the highlighted register
Empty	Sets the highlighted register to empty
Change	Sets the highlighted register to a value

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

Status pane This is the local menu command in this pane:

Toggle	Cycles through valid flag values
---------------	----------------------------------

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

Control pane This is the local menu command in this pane:

Toggle	Cycles through valid flag values
---------------	----------------------------------

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

Hierarchy window

oOp

The Hierarchy window has two panes, the Class pane and the Hierarchy Tree pane.

Class pane

Inspect	Shows contents of highlighted class
Tree	Moves to the Hierarchy Tree pane

Hierarchy Tree pane

Inspect	Shows contents of highlighted object or class
Parents	Toggles whether Parent Tree pane is displayed if you are running a program with multiple inheritance

Parent Tree pane

Inspect	Shows contents of highlighted object or type
----------------	--

Registers window menu

The Registers window is identical to the Register and Flags panes of the CPU window. Its local menus are identical to the Register pane local menu and the Flags pane local menu.

Stack window

The Stack window shows the currently active procedures.

Inspect	Shows source code for highlighted procedure
Locals	Shows local variables for highlighted procedure

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

Variables window

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

Global Symbol pane

Inspect	Shows contents of highlighted symbol
Change	Changes value of highlighted symbol
Watch	Opens Watches window and puts currently selected global symbol in window

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

Local Symbol pane

Inspect	Shows contents of highlighted symbol
Change	Changes value of highlighted symbol
Watch	Opens Watches window and puts currently selected global symbol in window
Show	Displays Show dialog box with following choices:
Static	Show only static variables
Auto	Show only variables local to current block
Both	Show both types of variables (default)
Module	Change current module

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

Watches window

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

Watch	Adds a variable or expression to watch
Edit	Lets you edit a watch variable or expression
Remove	Deletes highlighted variable or expression
Delete All	Deletes all watch variables and expressions
Inspect	Shows contents of highlighted variable or expression
Change	Changes contents of highlighted variable; does not affect expressions

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

any character	Watch
<i>Enter</i>	Edit
<i>Del</i>	Remove

Inspector window

An Inspector window shows the contents of a data item.

Range	Selects array members to inspect
Change	Changes the value of highlighted item
Inspect	Opens new Inspector window for highlighted item
Descend	Expands highlighted item into this Inspector window
New Expression	Inspects a new expression in this Inspector window
Type Cast	Typecasts highlighted item to new type

Class Inspector window

OOP

Class Inspector windows have two panes that show the contents (data members and member functions) of a class. Their local menus, the same for both panes, are quite different from the local menu of regular Inspector windows.

Inspect	Shows the contents of the highlighted class
Hierarchy	Returns to the Hierarchy window
Show Inherited	Toggles between showing contents of all ancestor types of object and contents declared in current type

Object Inspector window

OOP

Object Inspector windows contain three panes, of which only the first two have local menus. (The third displays only the class to which the object belongs). Both local menus are the same, and contain the following commands:

Range	Selects data members to inspect
Change	Changes value of highlighted item
Methods	Toggles whether member functions are summarized in middle pane

Show Inherited	Toggles between showing contents all base classes of object and contents declared in current class
Inspect	Opens new Inspector window for highlighted item
Descend	Expands highlighted item into this Inspector window
New Expression	Inspects a new expression in this Inspector window
Type Cast	Typecasts highlighted data item to new type
Hierarchy	Returns to the Hierarchy window

Text panes

Text pane is the generic name for a pane that displays the contents of a text file. The blinking cursor shows your current position in the file. The following table lists all the commands:

Table 13.2
Text pane key commands

Key	Function
<i>Ins</i>	Marks text block
↑	Moves up one line
↓	Moves down one line
→	Moves right one column
←	Moves left one column
<i>Ctrl</i> →	Moves to next word
<i>Ctrl</i> ←	Moves to previous word
<i>Home</i>	Goes to start of line
<i>End</i>	Goes to last character on line
<i>PgUp</i>	Scrolls up one screen
<i>PgDn</i>	Scrolls down one screen
<i>Ctrl-Home</i>	Goes to top line of pane
<i>Ctrl-End</i>	Goes to bottom line of pane
<i>Ctrl-PgUp</i>	Goes to first line of file
<i>Ctrl-PgDn</i>	Goes to last line of file

If you're not using the control-key shortcuts, you can also use the WordStar-style control keys for moving around a text pane.

List panes

This is the generic name for a pane that lists information you can scroll through. A highlight bar shows your current position in the list. Here's a list of all the commands available to you.

Table 13.3
List pane key commands

Key	Function
↑	Moves up one item
↓	Moves down one item
→	Scroll right
←	Scroll left
<i>Home</i>	Goes to start of line
<i>End</i>	Goes to last character on line
<i>PgUp</i>	Scrolls up one screen
<i>PgDn</i>	Scrolls down one screen
<i>Ctrl-Home</i>	Goes to top line of list pane
<i>Ctrl-End</i>	Goes to bottom line of list pane
<i>Ctrl-PgUp</i>	Goes to first item in list
<i>Ctrl-PgDn</i>	Goes to last item in list
<i>Backspace</i>	Backs up one character in incremental match
<i>Letter</i>	Makes incremental search (select by typing)
<i>Ins</i>	Marks multiple list items for block copy

If you're not using the control-key shortcuts, you can also use the WordStar-style control keys for moving around a list pane.

Commands in input and history list boxes

The following table shows the commands available when you're inside an input or list box:

Table 13.4
Dialog box key commands

Key	Function
↑	Moves up one list item
↓	Moves down one list item
→	Moves right one character
←	Moves left one character
Ctrl →	Moves to next word
Ctrl ←	Moves to previous word
Home	Goes to start of line
End	Goes to last character on line
PgUp	Scrolls up one screen
PgDn	Scrolls down one screen
Ctrl-Home	Goes to top line of list pane
Ctrl-End	Goes to bottom line of list pane
Ctrl-PgUp	Goes to first item in list
Ctrl-PgDn	Goes to last item in list
Backspace	Deletes the character before the cursor
Enter	Accepts your input and proceeds
Del	Deletes the character at the cursor
Esc	Cancel the dialog box and returns to menu
Ctrl-N	Completes partially typed name in input box

Window movement commands

The following table shows the keys you can use to reposition and resize a window:

Table 13.5
Window movement key commands

Key	Function
Ctrl-F5	Toggles window-positioning mode
↑	Moves window up one line
↓	Moves window down one line
→	Moves window right one column
←	Moves window left one column
Shift ↑	Resizes window; moves bottom up
Shift ↓	Resizes window; moves bottom down
Shift →	Resizes window; moves right side away from left
Shift ←	Resizes window; moves right side toward left
Home	Moves to left side of screen
End	Moves to right side of screen
PgUp	Moves to top line of screen
PgDn	Moves to bottom line of screen
Enter	Accepts current position
Esc	Cancel window-positioning command

Wildcard search templates

You can use wildcard search templates in two circumstances:

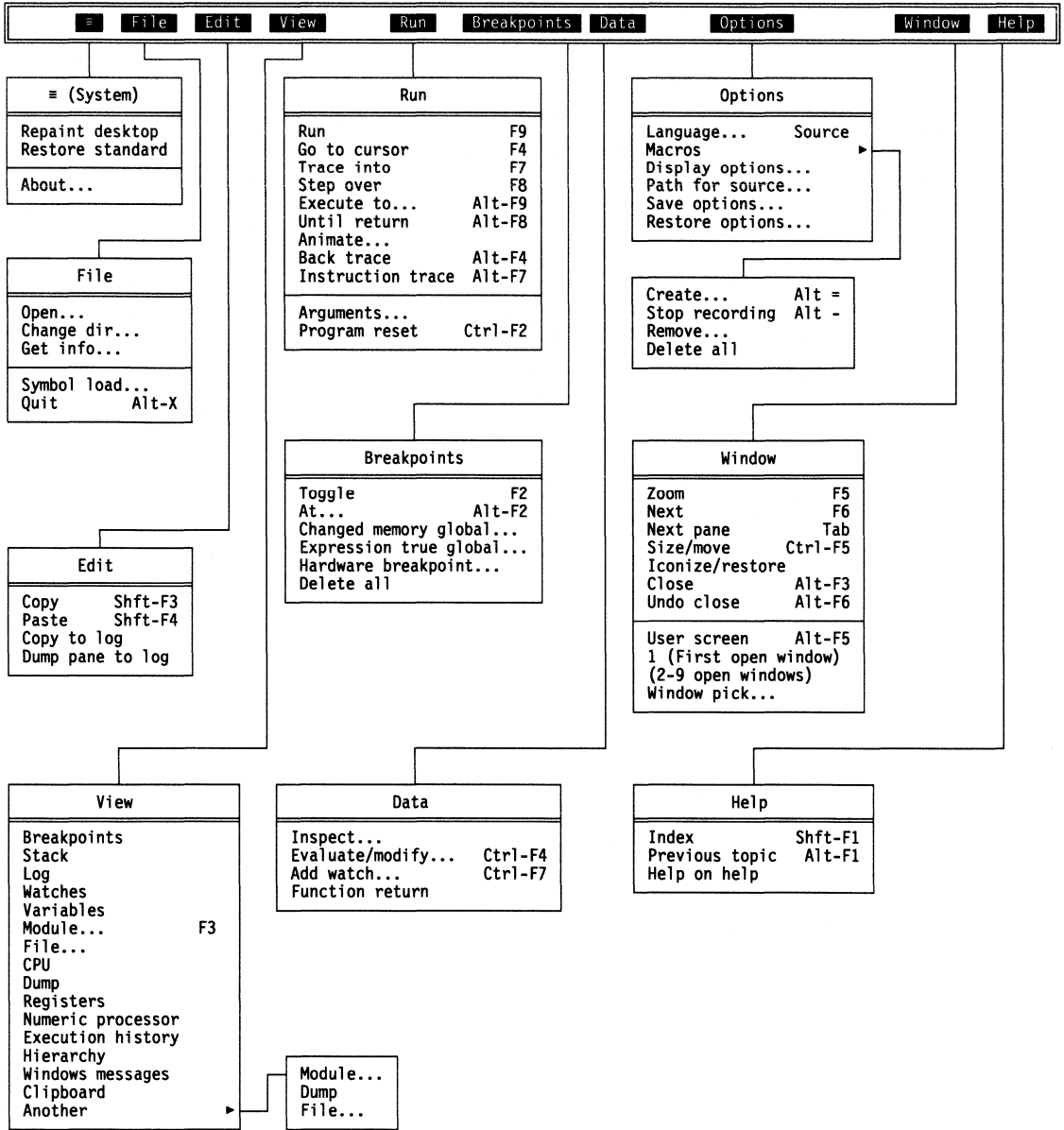
- when you enter a file name to load or examine
- when you enter a text search expression in a text pane

The ? (question mark) matches any single character in the search expression. The * (asterisk) matches 0 or more characters in the search expression.

Complete menu tree

Figure 13.1 shows the complete structure of Turbo Debugger's pull-down menus.

Figure 13.1: The Turbo Debugger menu tree



Debugging a standard C application

Debugging is like the other phases of designing and implementing a program—part science and part art. There are specific procedures that you can use to track down a problem, but at the same time, a little intuition goes a long way toward making a long job shorter.

The more programs you debug, the better you get at rapidly locating the source of problems in your code. You learn techniques that suit you well, and you unlearn methods that have caused you problems.

In this chapter, we discuss some different approaches to debugging, talk over the different types of bugs you may find in your programs, and suggest some ways to test your program to make sure that it works—and keeps on working.

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

When things don't work

First and foremost, don't panic! Even the most expert programmer seldom writes a program that works the first time.

To avoid wasting a lot of time on fruitless searches, try to resist the temptation to randomly guess where a bug might be. It is

better to use a universally tried-and-true approach: divide and conquer.

Make a series of assumptions, testing each one in turn. For example, you can say, "The bug must be occurring before function *xyz* is called," and then test your assumption by stopping your program at the call to *xyz* to see if there's a problem. If you do discover a problem at this point, you can make a new assumption that the problem occurs even earlier in your program.

If, on the other hand, everything looks fine at function *xyz*, your initial assumption was wrong. You must now modify that assumption to "The bug is occurring sometime *after* function *xyz* is called." By performing a series of tests like this, you can soon find the area of code that is causing the problem.

That's all very well, you say, but how do I determine whether my program is behaving correctly when I stop it to take a look? One of the best ways of checking your program's behavior is to examine the values of program variables and data structures. For example, if you have a routine that clears an array, you can check its operation by stopping the program after the function has executed, and then examining each member of the array to make sure that it's cleared.

Debugging style

Everyone has their own style of writing a program, and everyone develops their own style of debugging. The debugging suggestions we give here are just starting points that you can build on to mold your own personal approach.

Many times, the intended use of a program influences the approach you take to debug it. If a program is for your own use or will only be used once or twice to perform a specific task, a full-scale testing of all its components is probably a waste of time, particularly if you can determine that it is working correctly by inspecting its output. If a program is to be distributed to other people or performs a task of which the accuracy is hard to determine by inspection, your testing must be far more rigorous.

Run the whole thing

For a simple or throwaway program, the best approach is often just to run it and see what happens. If your test case has problems, run the program with the simplest possible input and check the output. You can then move on to testing more complicated input cases until the output is wrong. This testing procedure will give you a good feeling for just how much or how little of the program is working.

Incremental testing

When you want to be very sure that a program is healthy, you must test the individual routines, as well as checking that the program works as expected for some test input data. You can do this in a couple of ways: You can test each routine as you write it by making it part of a test program that calls it with test data. Or you can use TDW to step through the execution of each routine when the whole program is finished.

Types of bugs

Bugs fall into two broad categories: those peculiar to the language you're working in and those that are common to any programming language or environment.

By making mental notes as you debug your programs, you learn both the language-specific constructs you have trouble with, and also the more general programming errors you make. You can then use this knowledge to avoid making the same mistakes in the future, and to give you a good starting point for debugging future programs.

Understanding that each bug is an instance of a general family of bugs or misunderstandings will improve your ability to write errorless code. After all, it's better to write bug-free code than to be really good at finding bugs.

General bugs

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

Hidden effects If you are careless about using global variables in functions, a call to a function can leave unexpected contents in a variable or data structure:

```
char workbuf[20];
strcpy(workbuf, "all done\n");
convert("xyz");
printf(workbuf);
...
convert(char *p)
{
    strcpy(workbuf, p);
    while (*p)
        ...
}
```

Here, the correct thing to do would be to have the function use its own private work buffer.

Assuming initialized data Don't assume that another routine has already set a variable for you:

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

You should code a routine of this sort defensively by adding the statement

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

Not cleaning up This sort of bug can crash your program by exhausting heap space:

```

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

```

Fencepost errors These bugs are named after the old brain teaser that goes “If I want to put up a 100-foot fence with posts every 10 feet, how many fenceposts do I need?” A quick but wrong answer is ten (what about the final post at the far end?). Here’s a simple example:

```

for (n = 1; n < 10; n++)
{
    ...          /* oops--only 9 times */
}

```

Here you can easily see the numbers 1 and 10, and you think that your loop goes from one to ten. (Better make that `<` into a `<=`.)

C-specific bugs

Turbo C++ for Windows is very good at finding C-specific bugs that other compilers don’t warn you about. You can save yourself some debugging time by turning on all the warnings that the compiler is capable of generating. (See the Turbo C++ for Windows manuals for information on setting these warnings.)

What follows is by no means an exhaustive list of ways you can get in trouble with C. For some of these errors, Turbo C++ for Windows issues a warning message. Remember to examine the cause of any warning messages; they may be telling you about a bug in the making.

Using uninitialized automatic variables In C, an automatic variable declared inside a function is undefined until you assign a value to it:

```

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

```

```
}  
}
```

This function executes the **while** loop an unpredictable number of times because *n* is not initialized to 0 before being used as a counter.

Confusing = and == C lets you both assign a value (=) and test for equality (==) within an expression; for example,

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

This expression inadvertently loads *y* into *x* and performs the statements in the **if** expression if the value of *y* is not 0. You almost certainly meant to say

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

Confusing operator precedence C has so many operators that it is hard to remember which ones are applied first when an expression is evaluated. One combination that often causes grief is the mixture of shift operators with addition or subtraction. For example,

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

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

Bad pointer arithmetic When you use a pointer to step through an array, be careful how you increment and decrement it. For example,

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

does not increment *intp* to point to the next element of an integer array. Instead, *intp* is advanced by two array elements because in adding to or subtracting from a pointer, C takes into account the size of the item the pointer is pointing to. All you have to do to move the pointer to the next element is

```
intp++
```

Unexpected sign extension

Be careful about assigning between integers of different sizes:

```
int i = 0XFFFE;
long l;
l = i;
if (l & 0X80000000) {
    ...                /* this DOES get executed */
}
```

One of C's strong points can cause you trouble if you are not aware of how it operates. C lets you assign freely between scalar values (**char**, **int**, and so on). When you copy an integer scalar into a larger scalar, the sign (positive or negative) is preserved in the larger scalar by propagating the sign (highest) bit throughout the high portion of the larger scalar. For example, an **int** value of -2 (0xffff) becomes a **long** value of -2 (0xffffffe).

Unexpected truncation

This problem is the opposite of the previous one:

```
int i;
long l = 0X10000;
i = l;
while (i > 0) {
    ...                /* this does NOT get executed */
}
```

Here, the assignment of l to i resulted in the top 16 bits of l being truncated, leaving a value of zero in i .

Misplaced semicolons

The following code fragment may appear to be fine at first glance:

```
for (x = 0; x < 10; x++);
{
    ...                /* only executed once */
}
```

Why does the code between the braces execute only once? Closer inspection reveals a semicolon (;) at the end of the **for** expression. This hard-to-find bug causes the loop to execute ten times, but does nothing. The subsequent block is then executed once. This is a nasty problem because you can't find it with the usual technique of examining the formatting and indenting of code blocks in your program.

Macros with side effects

The following problem is enough to make you swear off **#define** macros for life:

```
#define toupper(c) 'a' <= (c) && (c) <= 'z' ? (c) - 'a' - 'A' : (c)
char c, *p;
c = toupper(*p++);
```

Here, *p* is incremented two or three times, depending on whether the character is uppercase. This type of problem is very hard to find, because the side effect is hidden within the macro definition.

Repeated autovvariable names

Another hard one to find:

```
myfunc()
{
    int n;
    for (n = 5; n >= 0; n--)
    {
        int n = 10;
        ...
        if (n == 0)
        {
            ... /* never gets executed */
        }
    }
}
```

Here, the automatic variable name *n* is reused in an inner block, hiding access to the one declared in the outer block. You must be careful about reusing variable names in this manner. You can get into trouble more easily than you might think, especially if you use a limited number of variable names for local loop counters (for example, *i*, *n*, and so forth).

Misuse of autovvariables

This function means to return a pointer to the result:

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

The trouble is that by the time the function returns, the automatic variable is no longer valid and is likely to have been overwritten by other stack data.

Undefined function return value

If you don't end a function with the **return** keyword followed by an expression, it returns an indeterminate value; for example,

```
char *first_capital_letter(char *p)
{
    while (*p)
    {
        if ('A' <= *p && *p <= 'Z')
            return(p);
        p++;
    }
    /* Oops--nothing returned here */
}
```

If there are no capital letters in the string, a garbage value is returned. You should put a `return(0)` as the last line of this function.

Misuse of break keyword

The **break** keyword exits from only a single level of **do**, **for**, **switch**, or **while** loops:

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

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

Code has no effect

Sometimes a typo results in source code that compiles, but doesn't do what you want it to. It may not do anything at all.

```
a + b;
```

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

Accuracy testing

Making a program work with valid input is only part of the job of testing. The following sections discuss some important test cases

that any program or routine should be subjected to before being given a clean bill of health.

Testing boundary conditions

Once you think a routine works with a range of data values, you should subject it to data at the limits of the range of valid input. For example, if you have a routine to display a list from 1 to 20 items long, you should make sure it behaves correctly both when there is exactly 1 item and exactly 20 items in the list. This can flush out the one-too-few and one-too-many “fencepost” errors (described on page 211).

Invalid data input

Once you are sure that a routine works with a full range of valid input, check that it behaves correctly when it's given invalid input. Check that erroneous input is rejected, even when it's very close to valid data. For example, the previous routine that accepted values from 1 to 20 should make sure that 0 and 21 are rejected.

Empty data input

Empty data input is a frequently overlooked area, both in testing and in designing a program. If you write a program to have reasonable default behavior when some input is omitted, you greatly enhance its ease of use.

Debugging as part of program design

When you first start designing your program, you can plan for the debugging phase. One of the most basic tradeoffs in program design involves the degree to which the different parts of your program check that they are getting valid input and that their output is reasonable.

If you do a lot of checking, you end up with a very resilient program that can often tell you about an error condition but continues to run after performing some reasonable recovery. You also end up with a larger and slower program. This type of program can be fairly easy to debug because the routines themselves inform you of invalid data before the dangers can be propagated.

You can also implement a program whose routines do little or no validation of input or output data. Your program will be smaller and faster, but bad input data or a small bug can bring things to a grinding halt. This type of program can be the most difficult to debug, since a small problem can end up manifesting itself much later during execution. This makes it hard to track down the original error.

Most programs end up being a mixture of these two techniques. You should treat input from external sources (such as the user or a disk file) with greater suspicion than data from one internal routine calling another.

The sample debugging session

This sample session uses some of the techniques we talked about in the previous sections. The program you are debugging, `TDDEMOB`, is a version of the demonstration program used in Chapter 3 (`TDDEMO.C`), except this one has some deliberate bugs in it. As with `TDDEMO`, `TDDEMOB` was compiled using the Turbo C++ for Windows EasyWin feature to display its output through Windows.

Make sure your working directory contains the two files needed for the debugging demonstration, `TDDEMOB.C` and `TDDEMOB.EXE`. (The *B* in these file names stands for “buggy.”)

Looking for errors

Before we start the debugging session, let’s run the buggy demo program to see what’s wrong with it. To start the program, type

```
TDDEMOB
```

You are prompted for lines of text. Enter two lines of text

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

A final empty line ends your input. `TDDEMOB` then prints out its analysis of your input:

```
Arguments:
Enter a line (empty line to end): one two three
Enter a line (empty line to end): four five six
Enter a line (empty line to end):
```

```
Total number of letters = 7
Total number of lines = 6
Total word count = 2
Average number of words per line = 0.3333333
'E' occurs 1 times, 0 times at start of a word
'F' occurs 1 times, 1 times at start of a word
'N' occurs 1 times, 0 times at start of a word
'O' occurs 2 times, 1 times at start of a word
'R' occurs 1 times, 0 times at start of a word
'U' occurs 1 times, 0 times at start of a word
There is 1 word 3 characters long
There is 1 word 4 characters long
```

Notice that there are erroneous numbers for the total number of words, letters, and word count. Later on, the letter and word frequency tables seem to be based on an erroneous letter and word count. This situation is all-too-typical—the program must have more than one bug. This happens frequently in the early stages of debugging a program.

Deciding your plan of attack

Your first task is to decide which problem to attack first. A good rule of thumb is to start with the problem that appears to be happening first. In this program, each input line is broken down into words, then analyzed, and finally, after all the lines have been entered, the tables are displayed. Since the word and letter counts are off as well as the tables, it's a good bet that something is wrong during the initial breaking down and counting phase.

Now is the time to start debugging, *after* you've thought about the problem for a moment and decided on a rough plan of attack. Here, the strategy is to examine the routine *makeintowords*, to see if it is correctly chopping the line into null-terminated words, and then see if *analyzewords* is correctly counting the analyzed line.

Starting Turbo Debugger

To start the sample debugging session, make sure the Turbo C++ edit window with TDDEMOB is current, then use the Run | Debugger command.

TDW loads the buggy demo program and displays the startup screen. If you want to exit from the tutorial session and return to Turbo C++, press *Alt-X* at any time. If you get hopelessly lost, you can always reload the demonstration program and start from the

beginning again by pressing *Ctrl-F2*. (Note that reloading doesn't clear breakpoints or watches.)

Since the first thing you want to do is to check that *makeintowords* is working correctly, run the program up to that routine and then check it. There are two approaches you can use: Either *step* through *makeintowords* as it executes, making sure that it does the right thing, or stop the program *after makeintowords* has done its stuff and see if it did the right thing.

Since *makeintowords* has a clearly defined task and it's easy to determine whether it's working correctly by inspecting the output buffer it produces, let's opt for the second approach. To do this, move down to line 42 and press *F4* to run to this line. When the program screen appears, type

```
one two three
```

and press the *Enter* key.

Inspecting

You are now stopped at the source line after the call to *makeintowords*. Look at the contents of *buffer* to see if the right thing happened. Move the cursor up a line, place it under the word *buffer*, and press *Alt-F10 I* (for Inspector) to open an Inspector window to show the contents of *buffer*. Use the arrow keys to scroll through the elements in the array. Notice that *makeintowords* has indeed put a single null character (0) at the end of each word as it is meant to. This means that you should execute more of the program and see if *analyzewords* is doing the right thing. First, remove the Inspector window by pressing *Esc*. Then, press *F7* twice to execute to the start of *analyzewords*.

Breakpoints

Check that *analyzewords* has been called with the correct pointer to the buffer by moving the cursor under *bufp* and pressing *Alt-F10 I*. You can see that *bufp* indeed points to the null-terminated string 'one'. Press *Esc* to remove the Inspector window. Since there seems to be a problem with counting characters and words, let's put a breakpoint at the places where a character and a word are counted:

1. Move to line 93 and press *F2* to set a breakpoint.
2. Move to line 97 and set another breakpoint.

3. Finally, set a breakpoint on line 99 so you can look at the character count this function returns.

Setting multiple breakpoints like this is a typical way to learn about whether things are happening in the right order in a program, and lets you check on important data values each time the program stops at a breakpoint.

The Watches window

Run the program by pressing *F9*. The program stops when it reaches the breakpoint on line 93. Now you want to look at the value of *charcount*. Since you'll want to check it each time you hit a breakpoint, this is an ideal time to use the **W**atch command to place it in the Watches window. Move the cursor under *charcount* and press *Alt-F10 W*. The Watches window at the bottom of the screen now displays the current value of 0. To make sure that the character is being counted properly, execute a single line by pressing *F7*. The Watches window now shows that *charcount* is 1.

The Evaluate/Modify dialog box

Run the program again by pressing *F9*. You are now back at line 93 for another character. Press *F9* again twice to read the last letter on the word and the terminating null. *charcount* now correctly shows 3, and the *wordcounts* array is about to be updated to count a word. Everything is fine so far. Press *F9* again to start processing the next word in the buffer. AHA! Something is wrong.

You expected the program to stop again on line 93 as it processed the next word, but it didn't. It went straight to the statement that returns from the function. The only way to end up on line 99 is if the **while** loop that started on line 83 no longer has a true test value. This means that **bufp != 0* must evaluate to false (that is, 0).

To check this, move back to line 83 and mark the entire expression **bufp != 0* by putting the cursor under the ***, pressing *Ins*, and moving the cursor to the final *'0'* before the *')'*. Now evaluate this expression by opening the **Data | Evaluate Modify** dialog box and either pressing *Enter* or choosing the Eval button to accept the marked expression. The value is indeed 0. Press *Esc* to return to the Module window.

Eureka!

Now here comes the analytical leap that enables you to “solve” the bug. The reason `bufp` points to a 0 is because that is where the inner **while** loop starting on line 86 left it at the end of a word. To continue to the next word, you must increment `bufp` past the 0 that ended the previous word. To do this, you need to add a “`bufp++`” statement before line 97. You could recompile your program with this statement added, but Turbo Debugger lets you “splice” in expressions by using a fancy sort of breakpoint.

To do this, first reload the program by pressing `Ctrl-F2` so you can test with a clean slate. Now remove all the breakpoints you set in the previous session by typing `Alt-B D`. Go back to line 97 and set a breakpoint again by pressing `F2`. Now, open a Breakpoints window by pressing `Alt-V B`. Do the following to set this breakpoint to execute the expression `bufp++` each time it is encountered:

1. With the Breakpoints window open, press `Ctrl-S` to open the Breakpoint Options dialog box.
2. Press the **Change** button to display the Conditions and Actions dialog box.
3. Set the Action radio buttons to Execute.
4. Press `Tab` to get to the Action Expression prompt.
5. Enter `bufp++`.
6. Press `Enter` twice to return to the Breakpoints window. Displayed in the right pane of this window is the new action *Execute “bufp++”*.
7. Press `Alt-F3` to return to the Module window.

Now run the program. Enter the usual two input lines.

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

Press `Enter` at the third prompt, and when the program terminates, look at your output on the User screen.

You’ll notice that things have improved considerably. The total number of words and lines seem to be wrong, but the tables are correct. Close the application window to get back to the Module window.

The next thing you'll do is stop at the beginning of the *printstatistics* routine and see if it is given the correct values to print. First reload the program by pressing *Ctrl-F2* to reset. Then go to line 104 and press *F4* to execute to there.

Enter the usual two input lines,

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

then press *Enter* on the third prompt.

Now that you're back in the Module window on line 104, move the cursor to the *nlines* argument and press *Ctrl-I* to look at its value. Note that the value is 6, the wrong value. It should be 2.

Next, go back to where *nlines* is called from in **main** and look at its value there. Move the cursor to line 36, place it under *nlines*, and press *Ctrl-I* to look at the value. The value of *nlines* in **main** is 2, which is correct. If you go down to line 46, you will notice that the two arguments *nwords* and *nlines* have been reversed. There is no way that the compiler could have known that you meant to have them the other way around.

If you correct these two bugs, the program will run correctly. The file TDDEMO.EXE is a corrected version you can run if you're curious.

Debugging an ObjectWindows application

The sample Windows programs in this chapter were written using the ObjectWindows application framework that makes Windows programming so much easier.

The programs are TDODEMO and TDODEMOB (the *B* stands for buggy). TDODEMOB has several bugs in it that you'll discover by working through this chapter.

If there are no .EXE files for TDODEMO and TDODEMOB, you'll have to open their project files and compile them with debug information included.

Before continuing, it might be helpful if you start TDODEMO from Windows and play with it a bit to get an idea of how it works. You can either use the Program Manager File | Run command to start TDODEMO.EXE or add it to a program group as an icon.

About the program

TDODEMO is an ObjectWindows program that lets you use a mouse to scribble in various colors on the screen. When you click the left mouse button and drag the mouse, the program draws on the screen. You can clear the window by clicking the right-hand mouse button. TDODEMO has a menu bar that lets you pick any of four pen colors: Red, Green, Blue, or Black.

You draw by pressing the mouse button, moving the mouse, and releasing the mouse button. The program accomplishes this task

easily by using the ObjectWindows library and *dynamic virtual member functions*. A dynamic virtual member function is a virtual member function with a numeric identifier (called a *dispatch index*) attached to it.

Because Turbo C++ for Windows defines Windows message names as numeric constants, you can use a Windows message name as the dispatch of a dynamic virtual member function. ObjectWindows can then call the member function whenever the window for which the member function is declared receives a message that matches the member function's dispatch index. If there is no member function with an identifier matching the Windows message, ObjectWindows calls the default window function.

For example, in order to create a function that responds to WM_MOUSEMOVE messages, you can declare a function within a window object that looks like this:

```
virtual void WMMouseMove(RTMessage Msg) = [WM_FIRST+WM_MOUSEMOVE];
```

As you can see, immediately after the function declaration, you use an equal (=) sign to attach an expression in brackets ([]) to the function, in this case WM_FIRST+WM_MOUSEMOVE. You add ObjectWindows constant WM_FIRST to the message constant WM_MOUSEMOVE to indicate that the message constant represents a WM-type message.

The type **RTMessage** contains the Windows procedure parameters *wParam* and *lParam*. These parameters often hold additional information about the message, such as where the cursor is positioned.

The next few sections explain how the TDODEMOB program works. They purposely gloss over the bugs so you can discover them later. It might be helpful to start Turbo C++ for Windows and open TDODEMOB.CPP so you can follow along in the code.

The Color Scribble window type definition

The Color Scribble window class is defined as follows:


```

class ScribbleWindow : public TWindow
{
public:
    HDC HandleDC;           // Display context for drawing.
    BOOL ButtonDown;       // left-button-down flag
    HPEN ThePen;           // Pen that is used for drawing in color
    ScribbleWindow(PWindowsObject AParent, LPSTR ATitle);
    ~ScribbleWindow();
    void GetWindowClass(WNDCLASS &AWndClass);

    virtual void WMLButtonDown(RTMessage Msg)=[WM_FIRST+WM_LBUTTONDOWN];
    virtual void WMLButtonUp(RTMessage Msg)=[WM_FIRST+WM_LBUTTONUP];
    virtual void WMMouseMove(RTMessage Msg)=[WM_FIRST+WM_MOUSEMOVE];
    virtual void WMRButtonDown(RTMessage Msg)=[WM_FIRST+WM_RBUTTONDOWN];
    virtual void SelectRedPen(RTMessage Msg)=[CM_FIRST+CM_RED];
    virtual void SelectGreenPen(RTMessage Msg)=[CM_FIRST+CM_GREEN];
    virtual void SelectBluePen(RTMessage Msg)=[CM_FIRST+CM_BLUE];
    virtual void SelectBlackPen(RTMessage Msg)=[CM_FIRST+CM_BLACK];
    virtual void SetupWindow();
};

```

The **ScribbleWindow** class defines a window object that responds to the following user input:

- Mouse movement
- Left mouse button press and release
- Right mouse button press
- Pen color and position

There are three data members, *HandleDC*, *ButtonDown*, and *ThePen* that hold a device context, the state of the mouse button, and the current pen the user draws with, respectively.

ScribbleWindow The **ScribbleWindow** constructor attaches the menu to the program and initializes the *ButtonDown* data member to FALSE and *ThePen* to CM_BLACK.

GetWindowClass The **GetWindowClass** member function calls the standard **TWindow GetWindowClass** function to set up the window so it behaves like any other **TWindow**, and then initializes the program's custom icon.

WMLButtonDown When the user presses the left mouse button in the Color Scribble window and is about to draw, the window receives a `WM_LBUTTONDOWN` message, which causes `ObjectWindows` to call **WMLButtonDown** (since it has an identifier of `WM_FIRST+WM_LBUTTONDOWN`). **WMLButtonDown** moves the pen to the current position of the mouse and sets `ButtonDown` to indicate that the button is down, and then selects `ThePen` into the current device context. There are additional Windows calls this function should be making that will be discussed later.

WMLButtonUp When the user finishes scribbling and releases the mouse button, the window receives a `WM_LBUTTONUP` message, which in turn causes `ObjectWindows` to call **WMLButtonUp**. The program marks the `ButtonDown` variable `FALSE` and releases the device context associated with the window.

WMRButtonDown When the user presses the right button to clear the screen, `ObjectWindows` calls the function **WMRButtonDown**, which calls the Windows function **UpdateWindow**. Calling this function is supposed to clear the window.

WMMouseMove Once the user starts moving the mouse over the window, the window begins receiving `WM_MOUSEMOVE` messages, which cause `ObjectWindows` to call the function **WMMouseMove**. If the user has pressed the left mouse button, the program draws a line each time the mouse is moved. If the user hasn't pressed a mouse button, nothing at all happens.

The pen-color routines There are four functions that set the pen color by deleting the current pen and creating a new one of the correct color. These functions differ only in the color each one sets.

Creating the application

To create an application that uses the Color Scribble window, it's necessary to derive a class based on the `ObjectWindows` class **TApplication**. The purposes of this class, **CScribbleApplication**, are

- to redeclare the **InitMainWindow** function so the application can create a main window with the properties of **ScribbleWindow**

MainWindow, used to set up window message breakpoints later in this chapter, is a member of MyApp.

■ to provide a type for the object *MyApp*, which is used to set up the window and run the program

Now that you know how the program works, you can begin to debug it.

Debugging the program

If you haven't done so already, start Turbo C++ for Windows and load the project file, TDODEMOB.PRJ, then load TDODEMOB.CPP. Next, choose Run | Debugger to run TDW and load the demo program, then press *F9* to run the demo program under TDW.

"Exception 13" is the message TDW displays when your program causes an unrecoverable application error (UAE).

You can move the mouse around and even choose menu selections, but when you press the left mouse button to start drawing, the program fails and returns to TDW, which displays an "Exception 13" error message.

Finding the first bug

When you press *Esc* to clear the message box, TDW leaves you in the CPU window. This window is displayed because your program was executing Windows code when the failure occurred. Since you didn't return to the Module window, you don't have a convenient marker to tell you where you were in your program when it made the call to Windows that caused the UAE.

Before continuing, press *Alt-F3* to close the CPU window (you'll be working primarily in the Module window).

Finding the function that called Windows

Since the program failed when you pressed the left mouse button, it's likely that the problem is in the **WMLButtonDown** function. However, there's another technique you can use to see where your program was, a stack trace.

To do a stack trace, display the Stack window (choose View | Stack), then scroll down the list of hexadecimal instructions until you come to a line indicating a routine in your program (you'll see the name of the routine in ASCII characters). That line is the one that called the Windows kernel.

As you can see in the Stack window, the routine you need to look at is, indeed, **WMLButtonDown**. To go to this routine in the Module

window, first click in that window. Next, press *Ctrl-S*, type `WMLButtonDown`, and press *Enter* to find this routine. If you see the message "Search expression not found," go to the top of the file and press *Ctrl-N* to search again (in TDW, you can search only from the current cursor position to the end of the file). You might have to press *Ctrl-N* several times before you get to the function itself.

Debugging WMLButtonDown

`WMLButtonDown` takes a variable of type `RTMessage` as a parameter and extracts from this message the location of the mouse. It then calls the Windows functions `MoveTo` and `SelectObject` to position the pen in the window and select it as the current drawing tool.

Since you saw the name of this routine in the Stack window, the bad call to Windows has to be one of these two Windows calls. To see which it is, you can run the program to the beginning of this function and then single-step through it to see which call causes the UAE.

With the cursor on the first line of `WMLButtonDown`, reload TDODEMOB by pressing *Ctrl-F2*, then press *F4* to run the program to that point. When you see the Color Scribble window, press the left mouse button to get the program to return to TDW. (You might have to press a key to get Windows to release mouse messages.) This time, there is no UAE (at least not yet), because all that has executed so far is the Windows call to TDODEMOB's `WMLButtonDown` function. TDW returns you to the first line of that function.

Begin pressing *F7* to single-step through the program. When you press *F7* on the call to `MoveTo`, you see the error box displaying "Exception 13." The call to `MoveTo` must be the problem.

Debugging MoveTo

The parameters of `MoveTo` are the handle of the pen and the X and Y coordinates of the current cursor location. The coordinates come from the `Msg` parameter, which gets them from Windows. Unless the program picks up the wrong part of this message (it doesn't), these two parameters should be OK.

The culprit must be `HandleDC`, the pen's device context handle.

At this point, since you've had two UAEs, the safest course of action is to exit TDW and close Windows before continuing.

Fixing the bug If *HandleDC* was the cause of the UAE, the handle either was set improperly or was never set in the first place. In fact, the handle wasn't set. The program should have set the handle by initializing a display context with the following Windows function call:

```
HandleDC = GetDC(HWindow);
```

The following code shows **WMLButtonDown** with the context initialization statement added:

```
void ScribbleWindow::WMLButtonDown(RTMessage Msg)
{
    if ( !ButtonDown )
    {
        ButtonDown = TRUE;           // Mark mouse button as being
                                     // pressed so when mouse movement
                                     // occurs, a line will be drawn.

        HandleDC = GetDC(HWindow); // Create display context for drawing.

        MoveTo(HandleDC, Msg.LP.Lo, Msg.LP.Hi);
                                     // Move drawing point to location
                                     // where mouse was pressed.

        SelectObject(HandleDC, ThePen);
                                     // Select pen into display context.
    }
}
```

Testing the fix Open Windows and Turbo C++ for Windows and reload the TDODEMO project and source file. Add the context initialization statement to **ScribbleWindow::WMLButtonDown**, then compile the project with debug information included (choose Compile | Build All).

Just in case there are more bugs, run the program program under TDW again (choose Run | Debugger, then press *F9* when the Module window comes up).

Now if you draw with the pen, you see the drawing appear in the default color, black. Try different colors by selecting pen colors from the menu. Red, green, and blue all work fine; however, when you try to change the pen color back to black, the pen won't change color. It looks like you've found another bug.

Finding the pen color bug

The most likely culprit for this bug is the **ScribbleWindow** function that creates a black pen, **SelectBlackPen**. Exit Color Scribble, then press *Ctrl-F2* to reset the program. Set a breakpoint at the opening brace of **ScribbleWindow::SelectBlackPen**, then run the program and choose Pen | Black. (You might have to press a key to get Windows to release mouse messages.) TDW should have stopped execution at the breakpoint. Since it didn't, something else must be wrong.

It appears that **SelectBlackPen** is never being called. Because this routine relies on the dynamically dispatched virtual table (DDVT) to get called, it's possible that there's something wrong with the identifier for the function.

Setting a window message breakpoint

When a user chooses a menu item, Windows sends a WM_COMMAND message to the window that owns the menu. The *wParam* parameter of the message contains the identifier of the menu item that was selected. When an ObjectWindows window receives a WM_COMMAND message, it scans through the dispatch indexes of the window object looking for the value CM_FIRST+*wParam*. **SelectBlackPen** has an index of CM_FIRST+CM_BLACK, where CM_BLACK has the value 104.

In order to find out what the *wParam* parameter of the Pen | Black command message is, you need to tell TDW to stop execution when it receives a WM_COMMAND message. You can then run the program, make the menu selection, and then check the *wParam* part of WM_COMMAND to see if it matches the constant CM_BLACK.

Before you can set the breakpoint, you have to get back to TDW. Close the Color Scribble application window, then, when you're back in the Module window, use *Ctrl-F2* to reload TDODEMOB. When the module window comes up, the next step is to set a window message breakpoint using one of two methods, depending on whether you have ObjectWindows message support enabled or disabled.

See the file TDWINST.TDW for information on TDWINST.

By default, there is no special support for ObjectWindows window message breakpoints. You can't use a window object you've declared in your program to set a window message breakpoint; you have to use the window handle instead. If you want to use the

window object (which is easier, but might slow down debugging when you have a window message breakpoint set), you have to run TDWINST, choose Options | Source Debugging, and check the OWL Window Messages checkbox.

Setting a window message breakpoint with a handle

If you don't have ObjectWindows support enabled, you must set the window message breakpoint by using the window handle. Because most of the window setup is done in ObjectWindows, you have to go to some lengths to get access to the handle.

Initial setup of the window is done in the **InitMainWindow** function, but the handle isn't set until later. To return control back to TDW, you could set a breakpoint in one of the mouse-handling functions (such as **WMLButtonDown**), run the program, then use the mouse and cause the program to hit the breakpoint. (If your breakpoint is in **WMLButtonDown**, you could press the left mouse button.)

Another technique is to redeclare the ObjectWindows function that initializes the handle, **SetupWindow**, so you can get control immediately after the handle is initialized. This function is redeclared in TDODEMOB as a virtual function and is defined as follows:

```
void ScribbleWindow::SetupWindow()  
{  
    TWindow::SetupWindow();  
}
```

To use it, position the cursor on the closing brace of the **SetupWindow** function, then press *F4* to run the program to that point.

Whichever method you use, when TDW regains control, do the following to set a window message breakpoint on the message **WM_COMMAND**:

1. Choose Data | Inspect and inspect the window object *MainWindow*. Because it is now out of scope, you have to use the following scope override syntax:

```
WinMain#MyApp.MainWindow
```

MainWindow is a member of *MyApp* because *MyApp* is of type **CScrubbleApplication**, which is derived from the class **TApplication**, of which *MainWindow* is a data member.

2. Zoom the Inspector window so you can see the data members in the top pane. *HWindow* is the data member that holds the window's handle.
3. Position the cursor on *HWindow*, then press *Shift-F3* to copy it into the Clipboard.
4. Choose View | Windows Messages to bring up the Windows Messages dialog box.
5. Press *Ctrl-A* in the top left pane to display the Add dialog box. Select the Handle button, then position the cursor in the text entry box.
6. Press *Shift-F4* to display the Clipboard. Put the cursor on *HWindow*, select the Contents button (to copy the contents of *HWindow*, the handle value), then choose OK to paste the handle into the text entry box.
7. In the text entry box, append `0x` to the front of the handle value to indicate that it's a hexadecimal number, then press *Enter*.
8. Move to the top right pane and type `WM_COMMAND`. You'll see the Set Message Filter dialog box appear as you begin typing.
9. Set the Action for this message to Break, then press *Enter* to set a breakpoint on this message.

The program will now return control to TDW whenever you make a menu selection, because doing so generates a `WM_COMMAND` message.

Setting a window message breakpoint with a window object

If you've used `TDWINST` to enable ObjectWindows window message breakpoint support, you can use the window object *MainWindow* to set the window message breakpoint.

1. Move the cursor to the closing brace of the *InitMainWindow* function and press *F4* to run the program to that point.
2. When you see the Module window again, choose View | Windows Messages to display the OWL Windows Messages dialog box.
3. In the top left pane, type `MainWindow` and press *Enter*.

4. In the top right pane, type `WM_COMMAND`, choose the Action button Break, then press *Enter* to set a breakpoint on this message.

The program will now return control to TDW whenever you make a menu selection, because doing so generates a `WM_COMMAND` message.

Inspecting `wParam`

You can now resume execution of the program by pressing *F9*.

Choose Pen | Black Pen from the menu. Once you have selected a black pen, TDW stops execution and displays the CPU Window, indicating that the program was executing Windows kernel code at the time the break occurred. Close the CPU window by pressing *Alt-F3*.

If necessary, bring up the Windows Messages window again. Zoom the window to full size so you can see the entire message in the lower pane. You can see that the window received a `WM_COMMAND` message with 204 (0xCC) in the `wParam` parameter. But the constant `CM_BLACK` is 104, not 204. The reason the virtual function was not getting called was that the application was looking for an identifier of `CM_FIRST+204`, but its value was actually `CM_FIRST+104`.

The value 204 was specified in the menu definition in the `TDODEMO.RC` file. This error could have been avoided by using the same identifiers for the menu as are in the header file and putting an **#INCLUDE** statement for that header file at the start of the `.RC` file. Instead, integers were used in this menu definition, which left the responsibility for cross-checking the values to the programmer.

If you edit `TDODEMOB.H` and change `CM_BLACK` to 204, selecting a black pen should work correctly. When you've made this change, the constant declarations in `TDODEMOB.H` will be as follows:

```
#define PenWidth 1
#define MenuID 100
#define IconID 100
#define CM_RED 101
#define CM_GREEN 102
#define CM_BLUE 103
#define CM_BLACK 204
```

Testing the fix Run Color Scribble and exit it, then exit TDW. When you are back in Turbo C++ for Windows, load the header file TDODEMOB.H, change the CM_BLACK constant definition, then recompile the project and run the program under TDW.

Now when you draw in the window, you might notice another problem. If, as you're drawing, you move the mouse off the window, then back onto the window at another location, you'll see that the program has drawn a straight line connecting the point where you left the window and the point where you came back on.

What the program should do is just stop drawing when you leave the window and start drawing when you come back. You've discovered yet another bug.

Finding the off-screen drawing bug

A place to start looking for this bug is in the window messages the window receives. Get out of the Color Scribble program and load TDODEMOB.CPP into TDW's Module window.

Logging the window messages

Depending on whether or not you have ObjectWindows window message support set, use one of the two methods described starting on page 231 to initialize the window. Then indicate in the Windows Messages window's top left pane (either by window object or handle) which window to track messages for.

Next, move the cursor to the top right pane and add WM_LBUTTONDOWN as a message breakpoint, to allow TDW to regain control after you finish drawing.

You also want to look at all the messages that come back, but setting WM_LBUTTONDOWN erased the Log All Messages setting. To restore this setting, press *Ctrl-A* to display the Set Message Filter dialog box, then select the Log All Messages class.

Discovering the bug

Resume execution of TDODEMOB by pressing *F9*. Begin drawing, then move the mouse off the client area, move it around a bit, and then move it back on again at another place and release the left mouse button so control returns to TDW.

Before looking at the Window Messages window, make sure to zoom it to full size (press *F5*) so you can see more messages. When

you look in the lower pane of the Windows Messages window, you see a lot of WM_NCHITEST and WM_SETCURSOR messages.

Scroll to the WM_LBUTTONDOWN message, which is where you started drawing. After this message, you see a series of WM_NCHITEST, WM_SETCURSOR, and WM_MOUSEMOVE messages, followed a series of WM_NCHITEST, WM_SETCURSOR, and WM_NCMOUSEMOVE messages, followed by another series of WM_MOUSEMOVE messages, and a final WM_LBUTTONUP message.

A WM_MOUSEMOVE message happens only in the program's client area. WM_NCMOUSEMOVE messages occur when the mouse is moved off the program's client area.

Now it becomes clear what the bug is. The program draws from the location of the last WM_MOUSEMOVE message to the location of the current WM_MOUSEMOVE message. When the mouse exits the client area, the program doesn't receive any WM_MOUSEMOVE messages. Therefore, when the mouse returns to the client area, the last location is where it left the screen, and the program obediently draws a line from that location to the current location.

Fixing the bug

One possible solution would be to determine when the mouse is off the client area, so the program can ignore the last mouse position and begin drawing again when the mouse reenters the client area. That would require some fancy logic to determine when the mouse was leaving the client area of the window and when it moves back over the client area. Fortunately, there is an easier way.

The Windows function **SetCapture** does exactly what's needed. This function tells Windows to send all mouse-related messages to the specified window until the program calls **ReleaseCapture**, thus causing the window to receive WM_MOUSEMOVE messages when the mouse is off the client area instead of WM_NCMOUSEMOVE messages.

If you put **SetCapture** in **ScribbleWindow::WMLButtonDown** and **ReleaseCapture** in **WMLButtonUp**, **WMMouseMove** will actually draw outside the window when the mouse is scribbling outside the window. However, Windows will clip all the lines drawn outside the client area, thus producing the desired effect.

These changes are shown in the following code listing:

```
void ScribbleWindow::WMLButtonDown(RTMessage Msg)
{
    if ( !ButtonDown )
    {
        ButtonDown = TRUE;           // Mark mouse button as being
                                     // pressed so when mouse movement
                                     // occurs, a line will be drawn.

        SetCapture(HWindow);        // Tell Windows to send all mouse
                                     // messages to window. WMLButtonUp
                                     // function will release the capture.

        HandleDC = GetDC(HWindow);   // Create display context for drawing.

        MoveTo(HandleDC, Msg.LP.Lo, Msg.LP.Hi);
                                     // Move drawing point to location
                                     // where mouse was pressed.

        SelectObject(HandleDC, ThePen);
                                     // Select pen into display context.
    }
}

void ScribbleWindow::WMLButtonUp(RTMessage)
{
    if ( ButtonDown )
    {
        ReleaseCapture();
        ReleaseDC(HWindow, HandleDC);
        ButtonDown = FALSE;
    }
}
```

Testing the fix

Exit Color Scribble, then exit TDW. When you're back in Turbo C++ for Windows, enter the changes to the two routines, then recompile the program and run it. Now when you draw on the window, everything works fine, but when you try to erase the screen by using the right-hand mouse button, nothing happens. You've found another bug.

Finding the erase-screen bug

Since **WMRButtonDown** is the function that handles the right-hand mouse button, the bug probably has something to do with that routine. Either **WMRButtonDown** isn't getting called, or there's a bug in it.

Exit Color Scribble, then load TDODEMOB into TDW. To execute to **WMRButtonDown**, the function where the bug probably is, press **Alt-F9** and type `ScribbleWindow::WMRButtonDown`. Scribble a little in the window, then press the right-hand mouse button. TDW stops the program at the beginning of **WMRButtonDown**, so you know this routine is getting called.

Using the *F7* key, step into **WMRButtonDown** and stop at the call to **UpdateWindow**. (Don't press *F7* yet on this line.) The only parameter is *HWindow*. You can assume that *HWindow* has been set correctly because other functions are using it successfully. Since there's nothing obviously wrong, one thing you can do is test to see if the `WM_PAINT` message that should be sent to the window by the call to **UpdateWindow** is actually being received by the window.

By now you probably know how to set a message breakpoint on `WM_PAINT`. If not, review the description of how to set a message breakpoint on page 231. In addition to setting a message breakpoint on `WM_PAINT`, set the program to log `WM_PAINT` messages. (In the Set Message Filter dialog box, add `WM_PAINT` again, but this time select the Log button.)

After setting the message breakpoint, press *F7* to execute the **UpdateWindow** call. Since the program doesn't break and return, `WM_PAINT` is never getting sent to the window.

You can verify that no `WM_PAINT` messages were received by pressing the right-hand mouse button to return control to TDW at **WMRButtonDown**, then checking the lower pane of the View | Windows Messages dialog box. There are no `WM_PAINT` messages there. For some reason, calling **UpdateWindow** isn't working as expected.

Analyzing the cause of the bug

This bug requires a little understanding of how Windows handles the **UpdateWindow** function. When a program calls this function, Windows checks to see if any part of the window is invalid and needs repainting. If so, Windows sends a `WM_PAINT` message to the window. If not, there's no reason to waste system resources with an unnecessary message, so Windows does nothing. But how does Windows know that the window needs updating?

An application notifies Windows that at least part of the Window is invalid by calling either **InvalidateRect** or **InvalidateRgn**. These two functions put an update area in the window and notify

Windows that it should update the window with a WM_PAINT message.

You could just replace the call to **UpdateWindow** with a call to **InvalidateRect**. However, Windows assigns a low priority to the WM_PAINT message it sends in response to either of these function calls, so if you want to ensure that the window gets updated immediately, you should retain the call to **UpdateWindow**.

Fixing the bug

Adding a call to **InvalidateRect** in **WMRButtonDown** will fix the problem. **InvalidateRect** takes three parameters, a window handle that identifies the window, a pointer to a rectangle that marks the rectangle to be added to the update rectangle, and a Boolean parameter that specifies if the rectangle should be erased. You can pass in NULL for the pointer to the rectangle, telling Windows that the entire window should be added to the update rectangle. The following code listing shows how **WMRButtonDown** should look with the new function call added:

```
void ScribbleWindow::WMRButtonDown (RTMessage)
{
    InvalidateRect (HWindow, NULL, TRUE);
    UpdateWindow (HWindow);
}
```

Testing the fix

Run Color Scribble and exit it, then exit TDW. When you are back in Turbo C++ for Windows, enter the changes to **WMRButtonDown**, then recompile the program and run it. Now when you draw on the window, then press the right-hand mouse button to erase it, the window does get erased. You've found all the bugs, and the program now works perfectly.

Summary of command-line options

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

```
tdw [options] [program_name [program_args]]
```

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

Table A.1
TDW command-line options

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

Error and information messages

TDW displays error messages and dialog boxes at the current cursor location. This chapter describes the dialog boxes and error and information messages TDW generates.

We tell you how to respond to both dialog box and error messages. All the dialog box messages and error messages (including the startup fatal error messages) are listed in alphabetical order, with a description provided for each one.

Dialog box messages

TDW displays a dialog box when you must supply additional information to complete a command. The title of the dialog box describes the information that's needed. The contents may show a history list (previous responses) that you have given.

You can respond to a dialog box in one of two ways:

- Enter a response and accept it by pressing *Enter*.
- Press *Esc* to cancel the dialog box and return to the menu command that preceded the dialog box.

Some dialog boxes only present a choice between two items (like Yes/No). You can use *Tab* to select the choice you want and then press *Enter*, or press *Y* or *N* directly. Cancel the command by pressing *Esc*.

For a more complete discussion of the keystroke commands to use when a dialog box is active, refer to Chapter 2.

Here's an alphabetical list of all the messages generated by dialog boxes:

Already recording, do you want to abort?

You are already recording a keystroke macro. You can't start recording another keystroke macro until you finish the current one. Press *Y* to stop recording the macro; *N* to continue recording the macro.

Device error – Retry?

An error has occurred while writing to a character device, such as the printer. This could be caused by the printer being unplugged, offline, or out of paper. Correct the condition and then press *Y* to retry or *N* to cancel the operation.

Disk error on drive __ – Retry?

A hardware error has occurred while accessing the indicated drive. This may mean you don't have a floppy disk in the drive or, in the case of a hard disk, it may indicate an unreadable or unwriteable portion of the disk. You can press *Y* to see if a retry will help; otherwise, press *N* to cancel the operation.

Edit watch expression

Modify or replace the watch expression. The dialog box is initialized to the currently highlighted watch expression.

Enter address, count, byte value

Enter the address of the block of memory you want to set to a particular byte value, then a comma, then the number of bytes you want to set, then another comma followed by the value to fill the block with.

Enter address to position to

Enter the address you want to view in your program. You can enter a function name, a line number, an absolute address, or a memory pointer expression. See Chapter 9 for more on entering addresses.

Enter animate delay (10ths of sec)

Specify how fast you want the *Animate* command to proceed. The higher the number, the longer between successive steps during animation.

Enter code address to execute to

Enter the address in your program where you want execution to stop. See Chapter 9 for more information on entering addresses.

Enter command-line arguments

Enter the command-line arguments for the program you're debugging.

Enter comment to add to end of log

Enter an arbitrary line of text to add to the messages displayed by the Log window. You can enter any text you want; it will be placed in the log exactly as you type it.

Enter destination address for marked block

Enter the segment:offset or segment that you want to move the marked block to.

Enter expression for conditional breakpoint

Enter an expression that must be true (nonzero) in order for the breakpoint to be triggered. This expression will be evaluated each time the breakpoint is encountered as your program executes. Be careful about any side effects it may have.

Enter expression to watch

Enter a variable name or expression whose value you want to watch in the Watches window. If you want, you can enter an expression that does not refer to a memory location, such as $x * y + 4$). If the dialog box is initialized from a text pane, you can accept the entry by pressing *Enter*, or change it and enter something else entirely.

Enter inspect start index, range

Enter the index of the first item in the array you want to view, followed by the number of items you want to view. Separate the two scalars by a space or a comma (,).

Enter instruction to assemble

Enter an assembler instruction to replace the one at the current address in the Code pane. The file ASMDEBUG.TDW has a condensed listing of all assembler keywords and discusses assembly language in more detail.

Enter log file name

Enter the name of the file you want to write the log to. Until you issue a **Close Log File** command, all lines sent to the log will be written to the file, as well as displayed in the window.

The default file name has the extension .LOG and is the same file name as the program you are debugging. You can accept this name by pressing *Enter*, or type a new name instead.

Enter memory address, count

Enter a memory address, followed by an optional comma and the number of items you want to clear. You can use a symbol name or a complete expression for the address.

Enter name of file to view

You can use DOS-style wildcards to get a list of file choices, or you can type a specific file name to load.

Enter new bytes

Enter a byte list that will replace the bytes at the position in the file marked by the cursor. See Chapter 9 for a complete description of byte lists.

Enter new coprocessor register value

Enter a new value for the currently highlighted numeric coprocessor register. You can enter a full expression to generate the new value. The expression will be converted to the correct floating-point format before being loaded into the register.

Enter new data bytes

Enter a byte list to replace the bytes at the position in the segment marked by the cursor. See Chapter 9 for a complete description of byte lists.

Enter new directory

Enter the new drive or directory name that you want to become the current drive and directory.

Enter new file offset

You are viewing a disk file as hexadecimal data bytes. Enter the offset from the start of the file where you want to view the data bytes. The file will be positioned at the line that contains the offset you specified.

Enter new line number

Enter the line number you want to see in the current module. If you enter a line number that is past the end of the file, you'll see the last line in the file. Line numbers start at 1 for the first line in the file. The current line number that the cursor is on is shown as the first line of the Module window.

Enter new relocation segment value

Enter an expression in the current language. This value will be used to set the base segment address of a symbol table that you loaded with the **File | Symbol Load** command. The expression that you enter should evaluate to the segment number of the start of the code for which the symbol table applies.

Enter new selector

Enter the selector value that you want to become current. You can enter an actual sector hex value, or you can enter a segment register value, such as CS, DS, or ES.

Enter new value

Enter a new value for the currently highlighted CPU register. You can enter a full expression to form the new value.

Enter port number

Enter the I/O port number you want to read from; valid port numbers are from 0 to 65,535.

Enter port number, value to output

Enter the I/O port number you want to write to, and the value to write; separate the two expressions with a comma. Valid port numbers are from 0 to 65,535.

Enter program name to load

Enter the name of a program to debug. You can use DOS wildcards to get a list of file choices, or you can type a specific file name to load. If you do not supply an extension to the file name, .EXE will be appended.

Enter read file name

Enter a file name or a wildcard specification for the file you want to read into memory. If you supply a wildcard specification or accept the default *.* , a list of matching files will be displayed for you to select from.

Enter search bytes

Enter a byte list to search for starting at the position in memory marked by the cursor. See Chapter 9 for a complete description of byte lists.

Enter search instruction or bytes

Enter an instruction, as you would for the **Assemble local** menu command, or enter a byte list as you would for a **Search** command in a Data pane.

Enter search string

Enter a character string to search for. You can use a simple wildcard matching facility to specify an inexact search string; for example, use * to match zero or more of any characters, and ? to match any single character.

Enter source address, destination, count

Enter the address of the block you want to move, the number of bytes to move, and the address you want to move them to. Separate the three expressions with commas.

Enter source directory path

Enter a list of directories, separated by spaces or semicolons (;). These directories will be searched, in the order that they appear in this list, for your source files.

Enter symbol table name

Enter the name of a symbol table to load from disk. Usually these files have an extension of .TDS. You must explicitly supply the filename extension.

Enter value to fill marked block

Enter a byte value to be filled into the marked block.

Enter variable to inspect

Enter the name of a variable or expression whose contents you want to examine. If the dialog box is initialized from a text pane, you can accept the entry by pressing *Enter* or change it and enter something else.

Enter write file name

Enter the name of the file you want to write the block of memory to.

Overwrite ___?

You have specified a file name to write to that already exists. You can choose to overwrite the file, replacing its previous contents, or you can cancel the command and leave the previous file intact.

Overwrite existing macro on selected key

You have pressed a key to record a macro, and that key already has a macro assigned to it. If you want to overwrite the existing macro, press *Y*; otherwise, press *N* to cancel the command.

Pick a method name

You have specified a routine name that can refer to more than one method in an object. Pick the correct one from the list presented.

Pick a module

Select a module name to view in the Module window. You are presented with a list of all the modules in your program. If you want to view a file that is not a program module, use **View | File**.

Pick a name

Pick a name from the list of displayed symbols. You can start to type a name, and you will be positioned to the first symbol, starting with what you have typed so far.

Pick a source file

Select a source file from the list displayed; only the source files that make up the current module are shown.

Pick a window

Pick a window from the list of active window titles.

Pick macro to delete

Pick the key or key combination for the macro you want to delete. The key will be returned to its original pre-macro functionality.

Press key to assign macro to

Press the key that you want to assign the macro to. Then, press the keys to do the command sequence that you want to assign to the macro key. The command sequence will actually be performed as you type it. To end the macro recording sequence, press the key you assigned the macro to, or press *Alt*.

Program already terminated, Reload?

You have attempted to run or step your program after it has already terminated. If you choose *Y*, your program will be reloaded. If you choose *N*, your program will not be reloaded, and your run or step command will not be executed.

Reload program so arguments take effect?

You have just changed the command-line arguments for the program you're debugging. If you type *Y*, your program will be reloaded and set back to the start. You usually want to do this after changing the arguments because programs written in many Borland languages only look at their arguments once—just as the program is loaded. Any subsequent changes to the

program arguments won't be noticed until the program is restarted.

Error messages

TDW uses error messages to tell you about things you haven't quite expected. Sometimes the command you have issued cannot be processed. At other times the message warns that things didn't go exactly as you wanted.

You can easily tell an error message from a prompt if you turn on Error Message Beeps in TDINST.

Fatal errors

All fatal errors cause TDW to quit and return to Windows. Some fatal errors are the result of trying to start TDW from the command line. A few others occur if something fatal happens while you are using the debugger. In either case, after having solved the problem, your only remedy is to restart TDW.

Bad or missing configuration file

The configuration file is either corrupted or not a TDW configuration file.

Invalid switch: __

You supplied an invalid option switch on the command line. Appendix A has an abbreviated list of all command-line switches, and Chapter 4 discusses each one in detail.

Not enough memory

TDW ran out of working memory while loading.

Old configuration file

You have attempted to start TDW with a configuration file for a previous version. You must create new configuration files for this version of TDW.

Unsupported video adapter

TDW can't determine which display adapter you are using. TDW supports EGA, VGA, Hercules, and Super-VGA.

Other error messages

'**)**' expected

While evaluating an expression, a right parenthesis was found to be missing. This happens if a correctly formed expression starts with a left parenthesis and does not end with a matching right one. For example,

```
3 * (7 + 4
```

should have been

```
3 * (7 + 4)
```

'**:**' expected

While evaluating a C expression, a question mark (?) separating the first two expressions of the ternary ? operator was encountered; however, no matching : (colon) to separate the second and third expressions was found. For example,

```
x < 0 ? 4 6
```

should have been

```
x < 0 ? 4 : 6
```

'**]**' expected

While evaluating an expression, a left bracket ([) starting an array index expression was encountered without a matching right bracket (]) to end the index expression. For example,

```
table[4
```

should have been

```
table[4]
```

This error can also occur when entering an assembler instruction using the built-in assembler. In this case, a left bracket was encountered that introduced a base or index register memory access, and there was no corresponding right bracket. For example,

```
mov ax,4[si
```

should have been

```
mov ax,4[si]
```

Already logging to a file

You issued an **Open Log File** command after having already issued the same command without an intervening **Close Log File** command. If you want to log to a different file, first close the current log by issuing the **Close Log File** command.

Ambiguous symbol name

You have entered a symbol name in an expression that does not uniquely identify a member function, and you have chosen not to pick the correct symbol from a list. You must pick the proper symbol from the list presented before your expression can be evaluated.

Bad or missing configuration file name

You have specified a nonexistent file name with the **-c** command-line option.

Cannot access an inactive scope

You entered an expression or pointed to a variable in a Module window that is not in an active function. Variables in inactive functions do not have a defined value, so you can't use them in expressions or look at their values.

Cannot be changed

You tried to change a symbol that can't be changed. The only symbols that can be changed directly are scalars (**int**, **long**, and so forth) and pointers. If you want to change a structure or array, you must change individual elements one at a time.

Can't have more than one segment override

You attempted to assemble an instruction where both operands have a segment override. Only one operand can have a segment override. For example,

```
mov es:[bx],ds:1
```

should have been one of the following:

```
mov es:[bx],1
```

or

```
mov ax,[1]
mov es:[bx],ax
```

Can't set a breakpoint at this location

You tried to set a breakpoint in ROM, nonexistent memory, or in segment 0. The only way to view a program executing in

ROM is to use the **Run | Trace Into** command to watch it one instruction at a time.

Can't set any more hardware breakpoints

You can't set another hardware breakpoint without first deleting one you have already set. Different hardware debuggers support different numbers and types of hardware breakpoints.

Can't set hardware condition on this breakpoint

You've attempted to set a hardware condition on a breakpoint that isn't a global breakpoint. Hardware conditions can only be set on global breakpoints.

Can't set that sort of hardware breakpoint

The hardware device driver that you have installed in your CONFIG.SYS file can't do a hardware breakpoint with the combination of cycle type, address match, and data match that you have specified.

Constructors and destructors cannot be called

This error message appears only if you are debugging a program that uses objects. You probably tried to evaluate a member function that's either a constructor or a destructor. This is not allowed.

Count value too large

In the Data pane of the CPU window, you've entered too large a block length to one of the local menu Block commands. The block length can't exceed FFFFh.

Ctrl-Alt-SysRq interrupt. System crash possible. Continue?

You attempted either to exit TDW or to reload your application program while the program was suspended as a result of your having pressed *Ctrl-Alt-SysRq*. Because Windows kernel code was executing at the time you suspended the application, exiting TDW or reloading the application will have unpredictable results (most likely hanging the system and forcing a reboot).

If possible, set a breakpoint in your code that will cause your program to exit to TDW, and then run your program again. When your program encounters the breakpoint and exits to TDW, you can terminate TDW or reload your program.

Destination too far away

You attempted to assemble a conditional jump instruction where the target address is too far from the current address.

The target for a conditional jump instruction must be within -128 and 127 bytes of the instruction itself.

Divide by zero

You entered an expression using the divide (/, **div**) or modulus operators (**mod**, **%**) that had on its right side an expression that evaluated to zero. Since the divide and modulus operators do not have defined values in this case, an error message is issued.

DLL already in list

In the **View | Modules** dialog box, you tried to add a DLL to the DLLs & Programs list, but the DLL was already in the list.

Error opening file ____

TDW couldn't open the file that you want to look at in the File window. The file might not exist or might be in another directory.

Error opening log file ____

The file name you supplied for the **Open Log File** local menu command can't be opened. Either there is not enough room to create the file, or the disk, directory path, or file name you specified is invalid. Either make room for the file by deleting some files from your disk, or supply a correct disk, path, and file name.

Error reading block into memory

The block you specified could not be read from the file into memory. You probably specified a byte count that exceeded the number of bytes in the file.

Error saving configuration

TDW could not write your configuration to disk. Make sure that there is some free space on your disk.

Error writing block to disk

The block that you specified could not be written to the file that you specified. You probably specified a count that exceeded the amount of free file space available on the disk.

Error writing log file ____

An error occurred while writing to the log file collecting the output from the log window. Your disk is probably full.

Error writing to file

TDW could not write your changes back to the file. The file might be marked as read-only, or a hard error may have occurred while writing to disk.

Expression too complex

The expression you supplied is too complicated; you must supply an expression that has fewer operators and operands. You can have up to 64 operators and operands in an expression. Examples of operands are constants and variable names. Examples of operators are plus (+), assignment (=), and structure member selection (->).

Expression with side effects not permitted

You have entered an expression that modifies a memory location when it gets evaluated. You can't enter this type of expression whenever TDW might need to repeatedly evaluate an expression, such as when it is in an Inspector window or Watches window.

Extra input after expression

You entered an expression that was valid, but there was more text after the valid expression. This sometimes indicates that you omitted an operator in your expression. For example,

```
3 * 4 + 5 2
```

should have been

```
3 * 4 + 5 / 2
```

Another example,

```
add ax,4 5
```

should have been

```
add ax,45
```

You could also have entered a number in the wrong syntax for the language you are using, for example, 0xF000 instead of 0F000h when you are in assembler mode.

Help file ____ not found

You asked for help but the disk file that contains the help screens could not be found. Make sure that the help file is in the same directory as the debugger program.

Immediate operand out of range

You entered an instruction that had a byte-sized operand combined with an immediate operand that is too large to fit in a byte. For example,

```
add BYTE PTR[bx],300
```

should have been

```
add WORD PTR[bx],300
```

Initialization not complete

You have attempted to access a variable in your program before the data segment has been set up properly by the compiler's initialization code. You must let the compiler initialization code execute to the start of your source code before you can access most program variables.

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. For example,

```
'A = 'a'
```

should have been

```
'A' = 'a'
```

Invalid format string

You have entered a format control string after an expression, but it is not a valid format control string. See Chapter 9 for a description of format strings.

Invalid function parameter(s)

You have attempted to call a routine in an expression, but you have not supplied the proper parameters to the call.

Invalid instruction

You entered an instruction to assemble that had a valid instruction mnemonic, but the operand you supplied is not

allowed. This usually happens if you attempt to assemble a **POP CS** instruction.

Invalid instruction mnemonic

When entering an instruction to be assembled, you failed to supply an instruction mnemonic. An instruction consists of an instruction mnemonic followed by optional arguments. For example,

```
AX,123
```

should have been

```
MOV ax,123
```

Invalid number entered

In a File window or a Module window, you typed an invalid number to go to (using the Goto command). Numbers must be greater than zero and in decimal format.

Invalid operand(s)

The instruction you're trying to assemble has one or more operands that aren't allowed. For example, a **MOV** instruction cannot have two operands that reference memory, and some instructions only work on word-sized operands. For example,

```
POP al
```

should have been

```
POP ax
```

Invalid operator/data combination

You've entered an expression where an operator has been given an operand that can't have the selected operation performed on it. For example, you attempt to multiply a constant by the address of a function in your program.

Invalid pass count entered

You have entered a breakpoint pass count that is not between 1 and 65,535. You can't set a pass count of 0. While your code is running, a pass count of 1 means that the breakpoint is eligible to be triggered the first time it is encountered.

Invalid register

You entered an invalid floating-point register as part of an instruction being assembled. A floating-point register consists of the letters *ST*, optionally followed by a number between 0 and 7 within parentheses; for example, *ST* or *ST(4)*.

Invalid register combination in address expression

When entering an instruction to assemble, you supplied an operand that did not contain one of the permitted combinations of base and index registers. An address expression can contain a base register, an index register, or one of each. The base registers are BX and BP, and the index registers are SI and DI. Here are the valid address register combinations:

```
BX  BX+SI
BP  BP+SI
DI  BX+DI
SI  BP+DI
```

Invalid register in address expression

You entered an instruction to assemble that tried to use an invalid register as part of a memory address expression between brackets ([]). You can only use the BX, BP, SI, and DI registers in address expressions.

Invalid symbol in operand

When entering an instruction to assemble, you started an operand with a character that can never be used to start an operand: for example, the colon (:).

Invalid type cast

A correct C cast starts with a left parenthesis, contains a possibly complex data type declaration (excluding the variable name), and ends with a right parenthesis. For example,

```
(x *)p
```

should have been

```
(struct x *)p
```

Invalid value entered

When prompted to enter a memory address, you supplied a floating-point value instead of an integer value.

Keyword not a symbol

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 ->). This symbol, however, was not preceded by a structure name, nor was it preceded by a pointer to a structure.

No coprocessor or emulator installed

You tried to create a Numeric Processor window using the **View | Numeric Processor** command, but there is no numeric processor chip installed on your system, and the program you're debugging either doesn't use the software emulator or the emulator has not been initialized.

No hardware debugging available

You have tried to set a breakpoint that requires hardware debugging support, but you don't have a hardware debugging device driver installed. You can also get this error if your hardware debugging device driver does not find the hardware it needs.

No help for this context

You pressed *F1* to get help, but TDW could not find a relevant help screen. Please report this to Borland technical support.

No modules have line number information

You have used the **View | Module** command, but TDW can't find any modules with enough debug information in them to let you look at any source modules. This message usually happens when you're debugging a program without a symbol table. See the "Program has no symbol table" error message entry on page 260 for more information on symbol tables.

No previous search expression

You attempted to perform a **Next** command from the local menu of a text pane, but you had not previously issued a **Search** command to specify what to search for. You can only use **Next** after issuing a **Search** command in a pane.

No program loaded

You attempted to issue a command that requires a program to be loaded. There are many commands that can only be issued when a program is loaded. For example, none of the commands in the **Run** menu can be performed without having a

program loaded. Use the **File | Open** command to load a program before issuing these commands.

No type information for this symbol

You entered an expression that contains a program variable name without debug information attached to it. This can happen when the variable is in a module compiled without the correct debug information being generated. You can supply type information by preceding the variable name with a typecast expression to indicate its data type.

Not a function name

You entered an expression that contains a call to a routine, but the name preceding the left parenthesis introducing the call is not the name of a routine. Any time a parenthesis immediately follows a name, the expression parser presumes that you intend it to be a call to a routine.

Not a record, structure, or union member

You entered an expression that used one of the C structure member selectors (`.` or `->`). This symbol, however, was not preceded by a structure name, nor was it preceded by a pointer to a structure.

Not enough memory for selected operation

You issued a command that needed to create a window, but there is not enough memory left for the new window. You must first remove or reduce the size of some of your windows before you can reissue the command.

Not enough memory to load program

Your program's symbol table has been successfully loaded into memory, but there is not enough memory left to load your program.

Not enough memory to load symbol table

There is not enough room to load your program's symbol table into memory. The symbol table contains the information that TDW uses when showing you your source code and program variables. If you have any resident utilities consuming memory, you might want to remove them and then restart TDW. You can also try making the symbol table smaller by having the compiler only generate debug information for those modules you are interested in debugging.

When this message is issued, you must free enough memory to load both your program and its symbol table. If you're

debugging a TSR program that's already loaded, then you must start Turbo Debugger using the **-sm** command-line option to reserve memory for the program's symbol table.

Only one operand size allowed

You entered an instruction to assemble that had more than one size indicator. Once you have set the size of an operand, you can't change it. For example,

```
mov WORD PTR BYTE PTR[bx],1
```

should have been

```
mov BYTE PTR[bx],1
```

Operand must be memory location

You entered an expression that contained a subexpression that should have referenced a memory location but did not. Some things that must reference memory include the assignment operator and the increment and decrement (**++** and **--**) operators.

Operand size unknown

You entered an instruction to assemble, but did not specify the size of the operand. Some instructions that can act on bytes or words require you to specify which size to use if it cannot be deduced from the operands. For example,

```
add [bx],1
```

should have been

```
add BYTE PTR[bx],1
```

Path not found

You entered a drive and directory combination that does not exist. Check that you have specified the correct drive and that the directory path is spelled correctly.

Path or file not found

You specified a nonexistent or invalid file name or path when prompted for a file name to load. If you do not know the exact name of the file you want to load, you can pick the file name from a list by pressing *Enter* when the dialog box first appears. The names in the list that end with a backslash (\) are directories, letting you move up and down the directory tree through the lists.

Program has invalid symbol table

The symbol table attached to the end of your program has become corrupted. Re-create an .EXE file and reload it.

Program has no objects or classes

You've attempted to open a **View | Hierarchy** window on a program that isn't object-oriented.

Program has no symbol table

The program you want to debug has been successfully loaded, but it doesn't contain any debug symbol information. You'll still be able to step through the program using a CPU window to examine raw data, but you won't be able to refer to any code or data by name.

Program linked with wrong linker version

You are attempting to debug a program with out-of-date debug information. Relink your program using the latest version of the 9linker or recompile it with the latest version of the compiler.

See page 57 for a description of how to compile your program with debugging information.

Program not found

The program name you specified does not exist. Either supply the correct name or pick the program name from the file list.

Register cannot be used with this operator

You have entered an instruction to assemble that attempts to use a base or index register as a negative displacement. You can only use base and index registers as positive offsets. For example,

```
INC WORD PTR[12-BX]
```

should have been

```
INC WORD PTR[12+BX]
```

Register or displacement expected

You have entered an instruction to assemble that has a badly formed expression between brackets ([]). You can only put register names or constant displacement values between the brackets that form a base-indexed operand.

Run out of space for keystroke macros

The macro you are recording has run out of space. You can record up to 256 keystrokes for all macros.

Search expression not found

The text or bytes that you specified could not be found. The search starts at the current location in the file, as indicated by the cursor, and proceeds forward. If you want to search the entire file, press *Ctrl-PgUp* before issuing the search command.

Source file ___ not found

TDW can't find the source file for the module you want to examine. Before issuing this message, it has looked in several places:

- where the compiler found it
- in the directories specified by the **-sd** command-line option and the **Options | Path for Source** command
- in the current directory
- in the directory where TDW found the program you're debugging

You should add the directory that contains the source file to the directory search list by choosing **Options | Path for Source**.

Symbol not found

You entered an expression that contains an invalid variable name. You might have mistyped the variable name, or it might be in some procedure or function other than the active one or out of scope in a different module.

Symbol table file not found

The symbol table file that you have specified does not exist. You can specify either a .TDS or .EXE file for the symbol file.

Syntax error

You entered an expression in the wrong format. This is a general error message when a more specific message is not applicable.

Too many files match wildcard mask

You specified a wildcard file mask that included more than 100 files. Only the first 100 file names will be displayed.

Unexpected end of line

While evaluating an expression, the end of your expression was encountered before a valid expression was recognized.

For example,

```
99 - 22 *
```

should have been

```
99 - 22 * 4
```

And this example,

```
SUB AX,
```

should have been

```
SUB AX, 4
```

Unknown character

You have entered an expression that contains a character that can never be used in an expression, such as a reverse single quote (').

Unknown record, structure, or union name

You have entered an expression that contains a typecast with an unknown record or enum name. (Note that assembler structures have their own name space different from variables.)

Unknown symbol

You entered an expression that contained an invalid local variable name. Either the module name is invalid, or the local symbol name or line number is incorrect.

Unterminated string

You entered a string that did not end with a closing quote ("). 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.

Variable not available

Your program's code has been optimized, and the variable you're looking for can no longer be accessed.

Video mode not available

You have attempted to switch to 43/50-line mode, but your display adapter does not support this mode; you can only use 43/50-line mode on an EGA or VGA.

????

- in Variables window 71
- in Watches window 93

:: (double colon) operator 143, 145

-? option (help) 61

≡ (System) menu 187

- activating 19

80x87 coprocessors *See* numeric coprocessors

80x86 processors

- debugging, breakpoints 112
- type, in CPU window 179

80386 processor

- debugging, Windows applications 13
- hardware debugging registers 13
- registers 141

A

accuracy testing 215

action

- breakpoints 104
- sets of and breakpoints 115

active window 31

- returning to 20

activity indicators 40

adapters *See* graphics adapters; video adapters

Add command

- breakpoints 106, 116
- window messages
 - message classes 162
 - window object 161
 - window selection 159

Add Comment command (log) 121, 122

Add Watch command 91

Add Window dialog box

- ObjectWindows application 162
- standard Windows application 159

addresses 133

- instructions, disassembled 180

memory *See* memory, addresses

running to specified 127

problems with 79

scope override for, C and C++ 134

symbol tables, base segment 245

Alt-key shortcuts *See* hot keys

Always option

breakpoints condition 112

display swapping 65

ancestor and descendant relationships 151

ancestor classes 156

Animate command 80, 242

Another command 30

arguments 3, *See also* parameters

calling function 27

command-line options 59, 247

changing 86

setting 81, 86

Arguments command 86

arrays

changing 250

indexes 243

inspecting 22, 31, *See also* Inspector windows
C tutorial 51

subranges of 97, 99, 101

quoted character strings and 148

watching 92, *See also* Watches window

arrow keys *See also* keys

history lists and 24

Inspector windows and 52

menu commands and 19

radio buttons and 21

resizing windows with 35

ASCII

files 201

display option for 130

searching 129

text, viewing files as 128, 129

ASMDEBUG.TDW file 10

assembler *See also* inline assembler
built-in 178, *See also* Code pane
problems with 249
code 29
tracking 30
data, formatting 183
inline, keywords
problems with 256
instructions *See also* instructions
back tracing and unexpected side effects 83
breakpoints and 107
executing single 78, 79
execution history and 83
multiple, treated as single 79
recording 83
watching 28, *See also* CPU window
memory dumps 183, 184
mode, starting TDW in 62
registers *See* CPU, registers
stack *See* Stack window
symbols 180

Assembler option (language convention) 132

assembly code, debugging 10

assignment operators *See also* operators

language-specific 72, 94

Turbo C++ for Windows 143

expressions with side effects and 90, 144

At command (breakpoints) 105, 116

B

Back Trace command 80

backward trace 17, 83, *See also* Back Trace

command; reversing program execution

assembler instructions 83

interrupts and 82

binary operators *See also* operators

Turbo C++ for Windows 143

bits 178

blinking cursor 34

blocks

memory *See* memory, blocks

moving 246

reading from, problems with 252

writing to files, problems with 252

books, reference 7

Borland, contacting 5

Both option (integer display) 66

bottom line *See* reference line

boundary errors 211

testing for 216

Break option (breakpoints) 113

Breakpoint Detail pane 106

Breakpoint List pane 106

breakpoints 26, 103-113, *See also* Breakpoints

window

actions 104, 113

sets of 115

At command 105

Boolean 112

Changed Memory 117

Changed Memory Global command 105

condition sets 114

conditional 118

conditions for triggering 103, 112

adding actions 111

customizing 111, 116

defined 103

Delete All command 105

disabling/enabling 110

Expression True Global command 105

Get Info message about 76

global 110, 116

memory variables and 117

where occurred in program 77

groups

Add command 108

defined 107

delete 109

disable 109, 113

enable 109, 113

Group ID text box 110

List dialog box 108

hardware 112, 118

hardware-assisted

device drivers and 117, 251

problems with 251, 257

Hardware Breakpoint command 105

inspecting 107

local variables 118

location 103

logging values 119

pass counts 104, 118, *See also* pass counts

setting 115

reloading programs and 85

- removing 104, 107
- running programs to 50
- saving 110
- scope 118
- setting 104
 - in module files 118
 - problems with 250, 251
 - tutorial 49
- simple 116
- templates and 119
- Toggle command 105
- using 221
 - with demo programs 219
- viewing 106
- window messages
 - Get Info message about 76
 - setting 164
 - TDODEMO 230
- Breakpoints command 105
- Breakpoints menu 104, 189
- Breakpoints window 26, 105-107
 - local menu 106, 191
 - opening 105
 - panes 106
- bugs 15-16, 207, 209-211
 - accuracy testing 215
 - boundary errors 211
 - testing for 216
 - C-specific 211-215
 - finding 16, 81, 207-208
 - backward trace and 80
 - demo programs
 - TDODEMOB 223-238
 - execution history and 82
 - history lists and 120
 - interrupting programs and 83
 - in subroutines 210
 - TDODEMO
 - SelectBlackPen function 230
 - stack trace 227
 - WMLButtonDown function 234
 - WMMouseMove function 227
 - WMRButtonDown function 236
 - fixing
 - TDODEMOB
 - line drawing 229
 - off-screen drawing 235
 - pen color 233
 - screen clearing 238
 - SelectBlackPen function 233
 - WMLButtonDown function 229, 235
 - WMRButtonDown function 238
 - incremental testing 209
 - returning information on 75
 - built-in assembler 178
 - built-in syntax checker 16
 - bullets (•)
 - Result box and 89
 - Watches window and 92
 - buttons 21, 38, *See also* dialog boxes
 - Help 21
 - radio *See* radio buttons
 - byte lists
 - entering 129, 139
 - language syntax 140
 - bytes 178, 180
 - formatting 183
 - hexadecimal, viewing files as 128
 - memory blocks
 - setting, responding to prompt 242
 - raw data 244
 - examining 94
 - searching for 261
 - watching 28

C

C++

 - arrays 53
 - code, tracing into 48
 - compound data types 53
 - data, types 51
 - demo programs 48
 - EasyWin module 43
 - expressions, entering in dialog boxes 54
 - functions 48, 49
 - returning from 48
 - tracing into 48
 - variables
 - inspecting 51
 - return values 53
 - watching 50
 - C++ expressions, problems with 250
 - C++ programs
 - class instances, formatting 89

- class member functions 74
- debugging, this parameter and 89
- multiple inheritance 151
- stepping through 79
- tracing into 78
- c option (load configuration file) 61
- problems with 250
- C programming language *See* Turbo C
- calculator 91
- calculator, using Watches window as 131
- capturing WM_MOUSEMOVE messages 235
- case sensitivity, overriding 62
- casting *See* type conversion
- central processing unit *See* CPU
- CGA *See* graphics adapters; video adapters
- Change command
 - Data Member pane local menu 155
 - Global pane local menu 72
 - Inspector window local menu 101
 - Static pane local menu 72
 - Watches window local menu 94
- Change dialog box
 - global symbols and 72
 - local symbols and 72
- Changed Memory Global command (breakpoints) 105
- Changed Memory option (breakpoints) 112
- character constants 254
- character devices, problems with 242
- character strings
 - null-terminated 95, 99
 - quoted 129
 - arrays as 148
 - problems with 262
 - searching for
 - File window, in 129
 - Module window, in 126
 - responding to prompt 246
 - searching for next
 - File window, in 130
 - Module window, in 127
 - Turbo C++ for Windows 142
- characters
 - display (ASCII vs. hex) 130
 - escape (Turbo C++ for Windows) 142
 - invalid 262
 - problems with scalar variables and 95
 - raw 148
 - value of 95
- check boxes 21, *See also* dialog boxes
 - Breakpoint Disabled 110
 - Save Configuration 67
- Class List pane 150
 - local menu 150
- Class pane, local menu 197
- classes, nested, scope of 137
- clearing a window, TDODEMO program 237
- Clipboard 36
 - local menu commands 39
 - tips for using 40
 - watching expressions 39
- Clipboard window
 - local menu 196
- clipping items from windows 36
- close box 33
- Close command 31, 36, 101
- Close Log File command 122
- code *See also* specific language application
 - breakpoints and 107, 110, 118
 - checking onscreen 30
 - current segment *See* programs, current location
 - debugging *See* debugging
 - disassembled, problems with 71
 - editing 123-124
 - exit, returned to Windows 77
 - inspecting 82, *See also* Inspector windows
 - splicing in (breakpoints) 113
 - stepping through 79, *See also* Step Over command
 - tracing into 78, *See also* Trace Into command
 - execution history and 81
 - viewing 178
 - execution history and 29
 - in multiple files 126, 130
 - watching *See also* Watches window
 - in slow motion 80
- Code pane
 - current program location 180
 - disassembler and 180
 - immediate operands and 180
 - instruction addresses 180
 - local menu 191
- color graphics adapters *See* graphics adapters

- color monitors *See* monitors
- .COM files, debugging 11
- command-line options *See also* specific switch
 - arguments 247
 - changing 86
 - setting 81, 86
 - disabling 60
 - entering 59
 - setting in TCW 59
 - summary of 239
 - syntax 58
 - help with 61
 - TDW utilities 11
- commands 22, *See also* specific menu command
 - assigning as macros 64
 - choosing 19
 - active windows and 31
 - dialog boxes and 241
 - escaping out of 20
 - hot keys and menu 20
 - local menu 24
 - summary of 185-205
 - onscreen 41, 42
- comments
 - adding to history lists 121
 - adding to log 243
- compiler directives, files and 123
- compiling demo programs, TDODEMO 229
- complex data objects 92
- complex data types 87
- compound data objects 91
 - inspecting 93
- condition sets (breakpoints) 114
- conditional breakpoints *See* breakpoints
- conditions *See also* breakpoints
 - controlling (breakpoints) 118
- CONFIG.SYS *See* configuration files
- configuration files 63
 - changing default name 67
 - directory paths 62
 - loading 61
 - overriding 61, 63
 - problems with 248, 250
 - saving
 - options to 66
 - problems with 252
 - TDCONFIG.TDW 36, 61, 63
- constants
 - Inspector windows and 94
 - problems with 254
 - Turbo Assembler 146
 - Turbo C++ for Windows 142
- constructors 90
 - problems with 251
- context-sensitivity 21, 22
 - help 40-42
- continuous trace 80
- control-key shortcuts *See* hot keys; keys
- Control pane, local menu 197
- conversion *See* type conversion
- coprocessors *See* 80x87 coprocessors; numeric coprocessors
- copying and pasting 36
- CPU *See also* CPU window
 - flags
 - state of 181
 - viewing 29, 184
 - memory, displaying 181
 - memory dump 183
 - registers 140
 - 80386 processor 141
 - compound data types and 91
 - optimization with 52
 - viewing 29, 184
 - state, examining 28, 178
- CPU command 94, 178
- CPU window 28
 - cursor in 179
 - disassembled code and 71
 - opening 178
 - automatic 28
 - panes 28
 - processor type in 179
 - program execution and 78-83
- Create command 64
- CScribbleApplication object (TDODEMO) 226
- Ctrl-Alt-SysRq (program interrupt key) 83
 - Program Reset command, and 84, 234
 - TDDEBUG.386, need for 12
- current activity, help with 40
- current code segment *See* programs, current location
- cursor 34
 - CPU window 179

running programs to 78
tutorial 48
cursor-movement keys *See* keys
customer assistance 5
customizing TDW 63, 64

D

data 88-91, *See also* Data pane
accessing 132
bashing, global breakpoints and 116
formatting 89
input 216, 217
inspecting 87-102, *See also* Inspector
windows
in recursive functions 74
manipulating 29
modifying 54
objects
complex 92
compound 91, 93
inspecting 88, 184
pointing at 91
watching 92
raw
displaying 183
examining 94
inspecting 184
viewing 28, 184, 244
structures, inspecting 22, 156
testing, invalid input and 216
truncated 89
types 87
complex 87
converting *See* type conversion
inspecting 31, 94-100
problems with 71, 89, 147
tracking 117
variables and 258
values 216
setting breakpoints for 117
viewing 178
in recursive routines 71
incorrect values shown 77
watching *See* Watches window
Data Member pane 152
local menu 153, 155
Data menu 88-91, 189

Data pane
local menu 192
memory addresses in 183
Debug Information command 57
debugging 15-19, 177, *See also* programs,
debugging
assembly code 10
.COM files 11
continuous trace 80
control 69-86, 133
memory use and 75
returning to TDW 78, 84
defined 15
demo programs *See* demo programs
dynamic link libraries 169
startup code 173
features 1, 18
functions 144
recursive 74
message logs and 27
multi-language programs 10
multiple components 53
object-oriented programs *See* object-oriented
programs, debugging
ObjectWindows programs 223
required files 2
restrictions 16
routines 123, 210
recursive 71
sessions 69
preparing programs for 57-67, 216
starting 85
simple programs 209
source files and 2
steps 15
strategies 218
terminology 3
tips
DLL Startup buttons 172
Load Symbols reset 172
window messages 165
tools 17
tutorial 43
Help with 45
TDODEMOB 227-238
variables 210
uninitialized 210

- Windows programs
 - features list 157
 - user interface 157
- decimal numbers 66
 - integers displayed as 148
- Decimal option (integer display) 66
- default settings
 - overriding 63
 - restoring 67
- Delete All command
 - Breakpoints window local menu 107
 - Macros menu 65
 - Watches window local menu 93
 - window messages
 - message classes 164
 - Windows Messages window
 - window proc 162
- Delete All command (breakpoints) 105
- demo programs 43
 - C++ 48
 - compiling and linking
 - TDODEMO 229
 - Help with 45
 - reloading 44
 - source file 43
 - starting 218
 - TDDEMO 44
 - TDODEMO 223
 - TDODEMOB 227-238
 - Turbo C++ for Windows 217
- derived class relationships 150
- Descend command
 - Data Member pane local menu 156
 - Inspector window local menu 101
- descendant relationships 150, 151
- destructors 90
 - problems with 251
- device drivers
 - breakpoints and 117
 - problems with 257
- dialog boxes 20-21
 - Add Group 108
 - bottom line in 42
 - Breakpoint Options 106, 109
 - Change 72
 - closing 67
 - commands and 241
 - Conditions and Actions (breakpoints) 111
 - Display Options 65
 - Edit Breakpoint Groups 108
 - escaping out of 241
 - Evaluate/Modify 89, 131, 220
 - Expression Language 132
 - Hardware Breakpoint Options 112
 - icons 19
 - Load Program 85
 - messages 241-248
 - moving around in 21
 - responding to 241
 - Save Options 67
 - search 126, 129
 - Watch 72
- directories
 - paths
 - for source 3
 - multiple 62
 - problems with 259
 - setting 62, 66, 246
 - starting directory, changing 63
- disassembled instructions 180
- disassembler 180
- disk drives, accessing, problems with 242
- disks
 - distribution 9
 - files on *See* files, disk
 - writing to, problems with 253
- display
 - formats
 - expressions 147
 - integers 66
 - modes
 - defaults, setting 65
 - problems with 262
 - options, saving 36
 - output 65
 - problems with 36
 - swapping 65
- Display As command
 - Data pane local menu 183
 - File window local menu 130
- Display Options command 65
- Display Options dialog box 65
- Display Swapping radio buttons 65
- Display Windows Info command 122, 166, 195

- distribution disks 9
 - copying 9
- DLL *See* dynamic link libraries
- do option (run on secondary display) 61
- DOS
 - versions 75
 - wildcards, choosing files and 127
- double colon (::) operator 143, 145
- drawing, cursor off-screen, with TDODEMOB 235
- drives *See* disk drives
- ds option (swap screens) 61
- Dump Pane to Log command 121
- Dump window 28, 184
 - local menu 194
- duplicate windows, opening 30
- Dynamic Link Libraries
 - expressions, accessing 138
 - scope considerations 138
- dynamic link libraries
 - debugging 169
 - startup code 173
 - reverse execution, and 80
- dynamic virtual member functions 223

E

- EasyWin module 43
- Edit command, Watches window local menu 93
- Edit menu 188
- editing
 - expressions 93
 - history lists 24
- EGA *See also* graphics adapters; video adapters
- line display 66
- EMS
 - execution history and 82
 - information about 75
- end of lines, problems with 261
- Enhanced Graphics Adapters *See* EGA
- Enter Program Name to Load dialog box 85
- Erase Log command
 - Log window local menu 122
 - Windows Messages window 165
- Erase Log File command 122
- erasing a window, TDODEMO program 237

- errors
 - boundary *See* boundary errors
 - Exception 13 77
 - fatal 248
 - messages 248-262
- escape sequences, Turbo C++ for Windows 142
- Evaluate input box 89
- Evaluate/Modify command 88-91, 131
- Evaluate/Modify dialog box 89, 131
 - using 220
- Exception 13 error message 77, 227
- exception codes 77
 - 13 77
- executable program files *See* files
- Execute option (breakpoints) 113
- Execute To command 79
- execution history 81
 - backward trace and 82
 - deleting 82
 - losing 83
 - recovering 83
- Execution History command 82
- Execution History window 29
 - opening 82
- exit code, returned to Windows 77
- exiting, TDW 67
- exiting, TDW, tutorial 44
- expanded memory specification *See* EMS
- Expression Language dialog box 132
- Expression True Global command (breakpoints) 105
- Expression True option (breakpoints) 112
- expressions 131-148
 - Clipboard, watching in 39
 - complex 88
 - editing 93
 - entering, problems with
 - character constants 254
 - inactive scope 250
 - invalid characters 262
 - invalid variables 261, 262
 - memory areas 259
 - no record name for field 257, 258
 - not routine name 258
 - operators 252, 253, 255
 - too complex 253
 - evaluating 88-91, 220

- implied scope 139
- language conventions 132
- problems with
 - end of line 261
 - no right bracket 249
 - no right parenthesis 249
 - scope 139
 - side effects 253
- formatting 147
 - problems with 254
- inspecting 31, 88, 102, 246, *See also* Inspector windows
- language options 132
- pointing at 91
- return values 92, 131
- scope override
 - C and C++ 134
 - Pascal 137
- syntax
 - Turbo Assembler 146-147
 - Turbo C++ for Windows 140-146
- undefined 93
- updating 93
- watching 91, 243, *See also* Watches window
 - format specifiers and 89
- with side effects (C programs) 90, 144

F

- fatal errors 248
- features, version 3.0 1
- File command
 - File window local menu 130
 - Module window local menu 126
 - View menu 127
- File menu 187
- File window 28
 - local menu 128, 194
 - opening 126
- FILELIST.DOC file 9
- files *See also* File menu; File window
 - ASMDEBUG.TDW 10
 - compiler directives and 123
 - configuration *See* configuration files
 - demo program 43
 - disk 28, 123, 127
 - history lists and 121
 - executable program 123, 245

- required for debugging 2
- FILELIST.DOC 9
- HELPME!.TDW 10
- include 123
- INSTALL.EXE 9
- list boxes and 26
- loading *See* files, opening
- log 243
 - problems with 250, 252
- MANUAL.TDW 10
- modifying, byte lists and 139
- moving to specific line number in 126, 129
- multiple, viewing 126, 130
- opening 85, 127, 244
 - problems with 252
 - nonexistent drive and directory 259
 - nonexistent or invalid name 259
 - wildcard masks and 261
- overriding 132
- overwriting 246
- program module
 - loading a new module 126
 - setting breakpoints in other 118
 - viewing 123
- README 10
- searching for 204
- searching through
 - File window, in 129
 - Module window, in 126
- source *See* source files
- TDCONFIG.TDW 36, 61, 63
- TDDEBUG.386 12
- TDDEMO.C 43
- TDDEMO.EXE 222
- TDDEMOB.C 217
- TDDEMOB.CPP 223
- TDDEMOB.CPP 223
- text 128, 201
- tracking 30
- UTILS.TDW 11
- viewing 28, 124, 130
 - as ASCII text 128
 - as hex data 128, 130
 - offset address 244
 - multiple 126, 130
 - source code 124
- writing to, problems with 253

- filled arrow 48
- Flags pane 181
- floating point
 - constants
 - Turbo Assembler 146
 - Turbo C++ for Windows 142
 - numbers
 - formatting 148, 183
 - problems with 29
 - registers 244
 - problems with 255
- format specifiers 89, 148
 - problems with
 - invalid string 254
- Full History command 83
- function keys 42, *See also* hot keys; keys
 - summary of 185-187
- Function Return command 91
- functions
 - calling 91
 - debugging 144
 - inspecting *See* Inspector windows
 - recursive, local data and 74
 - return values and current 91
 - returning from 79
 - stepping through 80
 - variables and inactive 250
 - Windows *See* Windows, functions
- functions, member, dynamic virtual 223

G

- Get Info command 75
- GetWindowClass function (TDODEMO)
 - Scribble Window 225
- gh2fp (type-cast symbol) 175
- global breakpoints *See also* breakpoints, global
 - where occurred in program 77
- global memory, Windows
 - information about 75
 - listing 166
- global menus 19, *See also* menus
 - local vs. 23
 - reference 187-190
- Global pane 70
 - local menu 71
- Global Symbol pane local menu 198
- global symbols 198

- disassembler and 180
- global variables *See also* variables
 - changing 72
 - debugging, in subroutines 210
 - inspecting 71, *See also* Inspector windows
 - same name as local 71
 - viewing 27, 70
 - in stack 27
 - Watches window, adding to 72
- GlobalAlloc function 166
- GlobalLock function 167
- GlobalPageLock function 167
- Go to Cursor command 78
- Goto command
 - File window local menu 129
 - Module window local menu 127
- graphics adapters *See also* hardware
 - EGA 66
 - problems with 262
 - supported 248
 - VGA 66
- Group command (breakpoints) 107

H

- h option (help) 61
- handle
 - memory
 - casting to far pointer 175
 - listing global memory, and 166
 - window
 - accessing in ObjectWindows programs 231
 - HWindow in MainWindow 232
 - messages
 - and 159, 162
- hardware
 - adapters *See* graphics adapters; video
 - adapters
 - breakpoints 118
 - memory variables and 117
 - debugging 13
 - problems with 251, 257
 - setting breakpoints 105
 - TDDEBUG.386 file required for 12
 - primary and secondary displays 61
 - requirements 2
 - Hardware Breakpoint command 105
 - Hardware option (breakpoints) 112

- HDWDEBUG.TD 112
- heap
 - allocation 210
 - global, Windows 166
 - local, Windows 168
- Help
 - demo programs 45
- help 40-42
 - accessing 41
 - problems with 253, 257
 - additional topics for 41
 - command-line options 61
 - TDW utilities 11
 - context-sensitive 40-42
 - current activity 40
 - dialog boxes 21
- Help button 21
- Help Index 41
- Help menu 41, 190
- Help on Help command 41
- Help screen
 - activating 41
 - highlighted keywords in 41
- HELPME!.TDW 10
- HELPME!.TDW file 10
- Hex display option (files) 130
- Hex option (integer display) 66
- hexadecimal bytes 129
 - viewing
 - data as 183
 - files as 128, 130
- hexadecimal constants, Turbo Assembler 146
- hexadecimal numbers 66
 - integers displayed as 148
- hierarchies, class 149
- Hierarchy command
 - Data Member pane local menu 153, 156
 - Member Functions pane local menu 154
 - View menu 149
- Hierarchy Tree pane 150, 151
 - local menu 151, 198
- Hierarchy window 149, 197
 - opening 149
 - panes 150-152
- highlight bar in windows 34
- history lists 24-25, *See also* execution history
 - breakpoints 120
 - editing 24
 - logging to 121
 - moving around in 202
- hot keys 20, *See also* keys
 - Alt = (Create Macros) 64
 - Alt-B (Breakpoints) 104
 - Alt-F4 (Back Trace) 80
 - Alt-F3 (Close) 36
 - Alt-F9 (Execute To) 79
 - Alt-F7 (Instruction Trace) 80
 - Alt-F6 (Undo Close) 36
 - Alt-F5 (User screen) 30
 - Ctrl-F2 (Program Reset) 81
 - Ctrl-F5 (Size/Move) 35
 - Ctrl-I (Inspect) 22
 - Ctrl-N (text entry) 25
 - dialog boxes 21
 - F2 (Breakpoints) 49
 - F4 (Go to Cursor) 78
 - F3 (Module window) 28
 - F6 (Next Window) 34
 - F9 (Run) 78
 - F8 (Step Over) 79
 - F7 (Trace Into) 78
 - F8 (Until Return) 79
 - F5 (Zoom) 35
 - help with 42
 - local menus 23, 42
 - macros as 26, 64
 - summary of 185-187
 - Tab/Shift-Tab (Next Pane) 34
- HWindow data member
 - window handle in MainWindow 232

I

- IBM display character set 148
- Iconize/Restore command 35
- icons
 - dialog boxes 19
 - menu 19
 - reducing windows to 33, 35
 - zoom 33
- identifiers, referencing in other modules 133
- include files 123
- incremental matching 25
- Index command 41
- indicators, activity 40

- inline assembler *See also* assembler
 - arrays, inspecting 99
 - constants 146
 - data, inspecting 98-100
 - expressions 146-147
 - INCLUDE compiler directive 123
 - instructions *See also* instructions
 - keywords, problems with 256
 - operators, precedence 147
 - pointers, inspecting 98
 - scalars, inspecting 98
 - structures, inspecting 100
 - symbols 146
 - unions, inspecting 100
- input *See* I/O
- input boxes 21, *See also* dialog boxes
 - Action Expression 113
 - Address (breakpoints) 106, 110
 - Condition Expression 112
 - entering text in 24, 25
 - Evaluate 89
 - Group ID (breakpoints) 110
 - history lists and 24-25
 - moving around in 202
 - New Value 89
 - Pass Count 115
 - Result 89
 - Save To 67
 - Tab Size, TDW 66
- Inspect command 51
 - Breakpoints window local menu 107
 - Class List pane local menu 150
 - Data Member pane local menu 152, 153, 156
 - Data menu 31, 88
 - Global pane local menu 71
 - Hierarchy Tree pane local menu 151
 - Inspector window local menu 101
 - Instructions pane local menu 82
 - Member Functions pane local menu 154
 - Module window local menu 125
 - Parent Tree pane local menu 152
 - Stack window local menu 74
 - Static pane local menu 72
 - Watches window local menu 93
- Inspector windows 17, 22, 31, 94-102
 - arrays 97, 99
 - classes 154
 - closing 31
 - compound data objects and 88, 102
 - functions 97
 - global symbols and 71
 - language-specific programs and 94
 - local menus 100-102
 - class 200
 - object 200
 - local symbols and 72
 - member functions 152
 - objects 156
 - opening 26
 - additional 31
 - panes
 - class 152
 - object 155
 - pointers 95, 98
 - problems with
 - character values in 95
 - multiple lines and 96, 99
 - pointers to arrays 96
 - reducing number onscreen 102
 - scalars 95, 98
 - structures 96, 100
 - unions 96, 100
 - using
 - C tutorial 51
 - in demo programs 219
 - variables in 71
 - viewing contents as raw data bytes 94
- INSTALL.EXE 9
- installation 12
 - TDDEBUG.386 12
 - TDW 9
- instruction opcodes, illegal 77
- Instruction Trace command 80
 - execution history and 82
- instructions *See also* Instructions pane
 - assembling 178
 - problems with 254, 255, 256
 - base and index registers 256, 260
 - instruction mnemonics 255
 - invalid registers 256
 - size indicators 259
 - target addresses 251
 - back tracing into 83
 - breakpoints and 116

- built-in assembler and 178
- divide, information about 77
- execution history and 82-83
- inspecting 82, *See also* Inspector windows
- machine 178
 - executing 78, 79, 80
- multiple assembly treated as single 79
- referencing memory 180
- viewing history of 82
- watching *See also* CPU window; Watches window

Instructions pane, local menu 82, 194

Integer Format radio buttons 66

integers

constants

Turbo Assembler 146

Turbo C++ for Windows 142

formatting 66

viewing

decimal 148

hexadecimal 148

watching 92, *See also* Watches window

international sort order 2

interrupting programs

using Ctrl-Alt-SysRq 83

using message breakpoints, in TDODEMO 230

interrupts

back tracing into 82

program *See also* Ctrl-Alt-SysRq

tracing into 80

Windows program, messages about 77

InvalidateRect function, in TDODEMO 237

InvalidateRgn function 237

I/O

ports

reading from 245

writing to 245

video 65

K

keys *See also* arrow keys; function keys; hot keys

assigning as macros 64

cursor-movement 34, 203

CPU window 179

dialog boxes 21, 203

Help window 41

menu commands 20

text boxes 202

text files 201

keystrokes

assigning as macros 64

displayed 29

recording 260

problems with 242

restoring to previous 65

keywords, inline assembler

problems with 256

keywords in Help window 41

L

-l option (assembler mode) 62, 173

labels, running programs to 79

tutorial 49

Language command 132

language-specific applications

assignment operators and 72

conventions 132

expressions and 131

Inspector windows and 94

scope override and 134

using 16, 131

Layout option (save configuration) 67

layouts, restoring 36

lh2fp (type-cast symbol) 175

Line command 126

line numbers 244

Code pane 180

displaying current 48

generating scope override 134

moving to specific 126, 129

problems with, source files and current 125

lines

end of, problems with 261

lines, multiple, problems with 96, 99

linked lists 102

list boxes 21, *See also* dialog boxes

Conditions and Actions (breakpoints) 111

incremental matching in 25

moving around in 202

list panes, Pick a Module 123

lists

choosing items from 34

- global memory 166
- local heap 168
- modules, Windows 168
- Load Modules or DLLs dialog box 170
- LoadLibrary function 172
- local and static variables
 - selecting for Variables window 73
 - watching 73
- local heap 168
- local memory, Windows, listing 168
- local menus 22-24, *See also* menus
 - accessing 23
 - Breakpoints window 106-107, 191
 - Class List pane 150
 - Class pane 197
 - Code pane 191
 - Control pane 197
 - Data Member pane 153, 155
 - Data pane 192
 - Dump window 194
 - File window 194
 - Global pane 71
 - Global Symbol pane 198
 - Hierarchy Tree pane 151, 198
 - Inspector windows 100-102, 200
 - Instructions pane 82, 194
 - Local Symbol pane 199
 - Log window 121, 194
 - Member Function pane 154
 - Message Class pane 196
 - Messages pane 196
 - Module window 125-127, 195
 - Object Member function pane 156
 - Parent Tree pane 152, 198
 - Register pane
 - CPU window 193
 - Numeric Processor window 197
 - Registers window 198
 - Selector pane 192
 - Stack pane 193
 - Stack window 74, 198
 - Static pane 72
 - Status pane 197
 - Variables window 198
 - viewing hot keys in 42
 - Watches window 93, 199
 - Window Selection pane 195

- Local Symbol pane local menu 199
- local variables *See also* variables
 - breakpoints and 118
 - changing 72
 - global values and 71
 - inspecting 72, *See also* Inspector windows
 - problems with 262
 - viewing 27
 - in stack 27
 - specific instances of 71, 74
- LocalAlloc function 168
- Locals command 71, 74
- location, breakpoints 103
- LockData function 167
- Log command (breakpoints) 120
- log files 243
 - opening, problems with 250, 252
 - writing to, problems with 252
- Log option (breakpoints) 113
- Log To File command 252
- Log window 27, 120-122
 - adding comments to 243
 - local menu 121, 194
 - opening 120
 - window messages, sending to 165
- Logging command 122

M

- machine instructions 178
 - executing 78, 79, 80
- macros 26
 - recording
 - keystrokes as 64
 - problems with 246, 260
 - removing 65
 - restoring to previous 65
 - saving 67
- Macros command 64
- Macros option (save configuration) 67
- MainWindow object, inspecting 231
- MANUAL.TDW 10
- math coprocessor *See* numeric coprocessors
- Member Function pane 152
 - local menu 154
- memory
 - addresses 131
 - disassembler and 180

- displaying 181
- dump 183
- entering 244
- problems with 256
- allocation
 - inspecting 75
 - problems with 210, 258
- blocks 242
 - problems with 252
- dump 28, 184
 - problems with 183
- global Windows
 - information about 75
 - listing 166
- handle
 - casting to far pointer 175
 - listing global memory 166
- local, Windows, listing 168
- locations, problems with 253, 259
- read-only 250
- references, formatting 148
- running out of 258
- watching 112
- Windows global, selectors (accessing) 181
- menu bar 19, 45
 - activating 19
 - commands 187
- menu trees 204-205
- menus 19-20
 - ≡ (System) 19, 187
 - activating 19
 - Breakpoints 104, 189
 - commands *See* commands
 - Data 88-91, 189
 - Edit 188
 - exiting 20
 - File 187
 - global 19
 - local vs. 23
 - reference 187-190
 - Help 41, 190
 - hot keys and 20
 - local *See* local menus
 - Options 64-67, 189
 - pop-up 19
 - pull-down 19
 - Run 69, 78-80, 189
 - program termination and 84
 - tutorial 45
 - View 26, 188
 - Window 33, 47, 190
- message breakpoints
 - Get Info message about 76
- Message Class pane
 - local menu 196
- message classes 163
 - Windows Messages window
 - adding to 162
 - deleting from 164
- message log 27, *See also* log files
- messages *See also* error messages
 - dialog boxes 241-248
 - program termination 76
- window
 - debugging tips 165
 - setting breakpoints, TDODEMO 230
- Windows
 - logging
 - to a file 165
 - to the TDW window 158
 - setting breakpoints 164
- Messages pane, local menu 196
- Methods command 155
- Microsoft Windows *See* Windows
- Mixed command 180
- modes *See* display modes
- Module command 257
 - Module window local menu 126
 - View menu 124
- Module window 28, 124-127
 - filled arrow and 48
 - local menu 125, 195
 - opening 124
 - duplicate 126
 - source files and 124
- modules 3, 123, *See also* Module window
 - compiling 57
 - current
 - changing 170
 - overriding 132
 - loading 124, 247
 - new 126
 - problems with 127
 - referencing identifiers in other 133

- scope override and 93
 - C and C++ 134
 - Pascal 137
- setting breakpoints in other 118
- tracing into 80
- tracking 30
- viewing 28, 124-127
 - duplicate 126
 - problems with 257, 261
 - source code in 244
- Windows, listing 168
- modulus operator, problems with 252
- monitors *See also* hardware; screens
 - display swapping 61
 - monochrome 61
- mouse
 - choosing menu commands 19-20
 - moving around in dialog boxes 21
 - setting breakpoints 49, 104
 - support
 - disabling/enabling 62
 - online help 41
 - windows and 32-33
- multi-language programs 10
- multiple inheritance 151

N

- nested classes, scope of 137
- New Expression command
 - Data Member pane local menu 156
 - Inspector window local menu 102
- new features for version 3.0 1
- New Value input box 89
- Next command *See also* Search command
 - File window local menu 130
 - Module window local menu 127
 - problems with 257
- Next Pane command 34
- Next Window command 34
- nonprinting characters 95
 - return value 148
- null-terminated character string 95, 99
- numbering system, windows 33
- numbers 91
 - decimal 66
 - floating-point *See* floating point, numbers
 - formatting 148

- problems with 253
 - Turbo Assembler 147
 - Turbo C++ for Windows 142
- hexadecimal 66
- scalar 139
- numeric coprocessors *See also* 80x87
 - coprocessors
 - current state, viewing 29
 - registers, entering new values for 244
- numeric exit code 77
- Numeric Processor window 29
 - opening, problems with 257
 - panes 197

O

- Object Member function pane
 - local menu 156
- object modules 123
- object-oriented programs
 - classes, inspecting 154
 - compatibility with Turbo Debugger 149
 - debugging 17, 156
 - nested object structures 152
 - this parameter and 93
 - expressions, problems with 250
 - member functions
 - problems with viewing 251
 - member functions, inspecting 152
- objects
 - formatting 89
 - hierarchy tree 150
 - inspecting 156
 - scope override 137
- objects, data *See* data, objects
- ObjectWindows
 - applications, debugging 223
 - TApplication object 226
- online help *See also* help
 - dialog boxes 21
- OOP *See* object-oriented programs
- opcodes, illegal instruction 77
- Open command 85
- Open Log File command 121
- operands 91, 253
 - instruction, memory pointers and 180
 - problems with 259
 - invalid 255

- out of range 254
- segment overrides and 250
- size 180
 - problems with 259
- operators 253
 - assignment 72, 94, *See* assignment operators
 - binary 143
 - C programs and 90
 - invalid 255
 - modulus, problems with 252
 - precedence
 - Turbo Assembler 147
 - Turbo C++ for Windows 143
- options 64, *See also* Options menu
 - command-line *See* command-line options
 - display swapping 65
 - program execution 78
 - restoring defaults 67
 - saving 66
- Options menu 64-67, 189
- Options option (save configuration) 67
- Origin command 75
 - Module window local menu 127, 139
- output *See also* I/O
 - display onscreen 65

P

- p option (mouse support) 62
- panes
 - blinking cursor in 34
 - Breakpoints window 26, 106
 - Code 180, *See* Code pane
 - CPU window 28
 - cycling through 179
 - Data 183, *See* Data pane
 - Execution History window 29, 82
 - Flags 181
 - Hierarchy window 150-152
 - highlight bar in 34
 - Inspector windows 31
 - class 152
 - object 155
 - list boxes 202
 - local menus and 22
 - moving between window 34
 - Numeric Processor window 29, 197
 - recording current contents of 121
- Register 181
- Registers window 29
- Selector 181
- Stack 183
 - text *See* text panes
- Variables window 27, 70
- Windows Messages window 195
- parameters 3, *See also* arguments
 - logging (breakpoints) 119
 - this 89, 93
 - viewing, program-calling 73
- Parent Tree pane 151
 - local menu 152, 198
- Parents command 151
- parsing, TDW vs. Turbo C++ 10
- Pascal option (language convention) 132
- pass counts 104, 115
 - problems with 255
- pasting and copying 36
- Path for Source command 3, 66
- paths, directory *See* directories
- Pick a Module list pane 123
- pointers 148
 - compound data objects 91
 - memory 132, 180
 - stack, current location 183
- pointing at data objects 91
- pop-up menus 19
- ports, I/O 245
- precedence, operators *See* operators
- Previous command 41
 - Module window local menu 126
- primary display 61, *See also* screens, swapping
- printers, problems with 242
- processors *See* 80x86 processors; CPU
- program execution, interrupting 83
- Program Reset command 81, 85
 - Ctrl-Alt-SysRq, and 84
 - TDODEMOB demo program 234
- programs 216
 - accuracy testing 215
 - compiling 18
 - current location 48, 146
 - CPU window 180
 - Inspector windows 71
 - Module window 74
 - problems with 81, 125

- returning to 75, 127, 139
- scope 139
 - overriding mechanism and 92
 - verifying 30
 - watching 72, 80, 123, *See also* Watches window
- current state 70
 - inspecting 70-77, *See also* Inspector windows
- debugging 16, 17, 57-60, 207-208, *See also* debugging
 - current scope and 139
 - dynamic link libraries 169
 - planning for 67, 216
 - returning information on 75-77
 - starting TDW 58
 - with no debug information 80, 260
 - with out-of-date debug information 260
- demo *See* demo programs
- execution *See also* programs, running
 - controlling 69-86
 - menu options 78
 - reversing 80, 82
 - problems with 83
 - terminating *See* programs, stopping
- fatal errors and 248
- full output screen 30
- incremental testing 209
- inspecting 22, *See also* Inspector windows
- language options, overriding 132
- loading 245, *See also* files, opening
 - dynamic link libraries 169
 - new 85
 - problems with 257, 260
 - symbol tables and 258
- message logs and 27
- modifying *See* programs, altering
- multi-language 10
- opening *See* programs, loading
- patching, temporarily 178
- recompiling 18
- recovering 83
- reloading 81, 85
 - Windows and Ctrl-Alt-SysRq 84
- restarting a debugging session 85
- returning from 48
- returning to 75, 126

- running 29, 69, 86, *See also* programs, execution
 - to breakpoints 50
 - command-line options and 86
 - to cursor 48, 78
 - at full speed 78
 - to labels 49, 79
 - returning information on 75
 - in slow motion 80
 - Windows, from 60
- scope *See* scope
- source code *See* code
- source files and 124
- stepping through
 - problems with 76
 - tutorial 48
- stopping 84, *See also* breakpoints
- stopping (breakpoints) 118
- watching *See* Watches window
- Windows
 - debugging 157
 - not loaded, problems with debugging 76
 - stopping, messages about 76
 - unexecuted, problems with examining values 77
- protected mode, selectors, accessing 181
- pseudovariables (Turbo C++ for Windows) 140
- pull-down menus 19

Q

- Quit command, TDW 67

R

- radio buttons 21, *See also* dialog boxes
 - Action 113
 - Changed Memory 117
 - changing settings 21
 - Condition 112
 - Display Swapping 65
 - Expression Language 132
 - Integer Format 66
 - Log 119
 - Screen Lines 66
- Range command
 - Data Member pane local menu 155
 - Inspector window local menu 101

- read-only memory *See* ROM
 - README file 10
 - READY indicator 25
 - RECORDING indicator 64
 - records, problems with 258, 262
 - recursive functions 74
 - recursive procedures 71
 - reference books 7
 - reference line, dialog boxes 42
 - Register pane 181
 - CPU window, local menu 193
 - Numeric Processor window, local menu 197
 - registers 94, *See also* Registers window
 - 80386 hardware debugging 13
 - CPU *See* CPU, registers
 - floating-point 244
 - problems with 260
 - invalid 255, 256
 - segment 84, 148
 - valid address combinations 256
 - values, accessing 29
 - Registers window 29, 184
 - local menu 198
 - panes 29
 - ReleaseCapture function 235
 - reloading programs 81
 - Windows and Ctrl-Alt-SysRq 84
 - Remove command
 - Breakpoints window local menu 107
 - Macros menu 65
 - Watches window local menu 93
 - Windows Messages window
 - message classes 164
 - window selection 162
 - Repaint Desktop command 36
 - repeat counts 147
 - resize box 33
 - restarting a debugging session 85
 - Restore Options command 36, 67
 - Restore Standard command 36
 - Result input box 89
 - return values 131
 - changing 94, 101
 - expressions 243
 - inspecting 91, *See also* Inspector windows
 - nonprinting characters 148
 - problems with 77, 256
 - tracking 92
 - variables *See* variables
 - return values (breakpoints) 119
 - Reverse Execute command 82
 - reversing program execution 80, 82, *See also*
 - backward trace
 - problems with 83
 - Windows code 80
 - ROM, programs executing in 250
 - routines
 - accessing 132
 - problems with 247
 - calling
 - problems with
 - invalid name 258
 - invalid parameters 254
 - debugging 123, 210
 - inspecting 74
 - variable with same name as 71
 - names, finding 27
 - recursive, local data and 71
 - stepping over 17
 - testing 216
 - viewing in stack 27, 73
 - Run command 78
 - execution history and 82
 - Run menu 69, 78-80, 189
 - program termination and 84
 - running
 - programs *See also* programs, running
 - TDW 60
- ## S
- sample programs *See* demo programs
 - Save Configuration check box 67
 - Save Options command 36, 66
 - Save Options dialog box 67
 - Save To input box 67
 - sc option (ignore case), Turbo Debugger 62
 - scalar numbers 139, 243
 - scalar variables 95
 - scientific notation 142, 146
 - scope 92, 133-139
 - breakpoint expressions 118
 - current 132, 139
 - accessing symbols outside 133
 - DLLs, and 138

- implied, evaluating expressions and 139
- nested classes 137
- overriding 134-138
 - tips
 - C, C++, Turbo Assembler 136
 - Pascal 138
 - problems with, inactive 250
 - templates 136
 - this parameter 89
- Screen Lines radio buttons 66
- screens *See also* hardware; monitors
 - display modes *See* display, modes
 - layouts, restoring 36
 - lines per, setting 66
 - problems with
 - graphics display 36
 - writing to 65
 - startup, TDDEMO 44
 - swapping 65
 - User *See* User screen
- ScribbleWindow type (TDDEMO) 225
- scroll bars 32
- scrolling 31
 - dialog boxes 203
 - Help screens 41
 - Inspector windows 52
 - menus 20
 - text boxes 202
 - text panes 201
- sd option (set source directories) 62
- Search command *See also* Next command
 - File window local menu 129
 - history lists and 24
 - Module window local menu 126
- search paths, source files, for 2
- search templates 204
- secondary display 61, *See also* display, swapping
- segment
 - overrides, problems with 250
 - pointers to register 148
 - registers, program termination and 84
- select by typing 25
- SelectBlackPen function (TDDEMO)
 - bugs
 - finding 230
 - fixing 233
- Selector command
 - Selector pane local menu 182
- Selector pane 181
 - local menu 182, 192
 - memory segments in 181
- Send to Log Window command (Windows Messages window) 165
- Set Message Filter dialog box 163
- Set Options command (breakpoints) 106
- SetCapture function 235
- SetupWindow function, redeclaring 231
- shortcuts *See* hot keys
- Show command 73
- Show Inherited command
 - Data Member pane local menu 153, 156
 - Member Functions pane local menu 154
- side effects
 - changing variable values with 90
 - splicing code (breakpoints) 113
- Size/Move command 35
- Smart option (display swapping) 65
- software, requirements 2
- sort order, international 2
- source code *See* code
- source files 3, *See also* files
 - language conventions and 132
 - loading 124, 247
 - problems with 261
 - search paths for 2
- Source option (language convention) 132
- stack 84, 178, *See also* Stack pane; Stack window
 - current state 27, 73-74
 - pointer, current location 183
- Stack command 71, 73
- Stack pane
 - current stack pointer 183
 - local menu 193
- stack trace 227
- Stack window 27, 73-74
 - local menu 74, 198
 - opening 71, 73
 - stack trace 227
- Standalone Debugging command 57
- starting directory, changing 63
- starting TDW 58
 - in assembler mode 62

- command-line options and 239
- startup code
 - dynamic link libraries 173
 - program, debugging 62
 - types of 172
- startup screens, TDDEMO 44
- Static pane 71
 - local menu 72
- static symbols, disassembler and 180
- status line 41, 42
- Status pane, local menu 197
- Step Over command 79
 - execution history and 82
- stepping over routines 17
- stepping through
 - functions 80
 - programs
 - problems with 76
- Stop Recording command 64
- strings 148
 - byte lists and 139
 - character
 - null-terminated 95, 99
 - quoted 129
 - problems with 262
 - searching for
 - File window, in 129
 - Module window, in 126
 - responding to prompt 246
 - searching for next
 - File window, in 130
 - Module window, in 127
 - Turbo C++ for Windows 142
 - format control *See* format specifiers
 - text, searching for 24
 - truncated 89, 92
- structures
 - changing 250
 - inspecting complicated data 88, 102
 - problems with 258, 262
- Super VGA support 2
- support, technical 5
- switches *See* command-line options
- switching Windows applications, TDW and 60
- symbol names, problems with 250
- Symbol pane 27
- symbol tables 132

- base segment address 245
 - creating 57, 260
 - dynamic link libraries, and 169
 - invalid 260
 - loading 246
 - problems with 258, 261
- symbols 71, 131
 - accessing 133-139, 247
 - disassembler and 180
 - global 198
 - problems with 250, 261, 262
 - invalid 256
 - type information and 258
 - scope 133
 - Turbo C++ for Windows 140
- syntax
 - checkers, built-in 16
 - errors 16, 261
- System Information box 75
- System menu *See* (System) menu

T

- t option (starting directory) 63
- Tab Size input box 66
- Table Load command 245
- tabs, setting 66
 - problems with 262
- TApplication, object 226
- Task List, Windows, TDW and 60
- TD386 virtual debugger
 - setting breakpoints 112
- TDCONFIG.TDW 36, 63
 - loading 61
 - overriding 63
- TDDEBUG.386 file 12
- TDDEMO 43
 - source files 44
- TDDEMO.EXE 222
- TDODEMO
 - CScrubbleApplication object 226
 - GetWindowClass function, ScribbleWindow 225
 - InvalidateRect function 237
 - InvalidateRgn function 237
 - MainWindow object 231
 - ScribbleWindow type 225

- SelectBlackPen function
 - bug, finding 230
 - bug, fixing 233
- SetupWindow function 231
- stack trace 227
- UpdateWindow function 237
- WMLButtonDown function
 - bug, finding 234
 - bug, fixing 229, 235
 - ScribbleWindow 226
- WMLButtonUp function 226
- WMMouseMove function 226
 - bug, finding 227
- WMRButtonDown function 226
 - bug, finding 236
 - bug, fixing 238
- .TDS files, creating 11
- TDW
 - logging window messages 158
 - window object, adding 161
 - window selection
 - adding 159
 - deleting 162
- technical support 5
- templates
 - breakpoint behavior 119
 - scope of 136
- ternary operators 143
- text 66
 - entering
 - active windows and 31
 - in input boxes 24, 25
 - incremental matching 25
 - in log 243
 - searching for 204, 261
 - strings, searching for 24
- text files 128, 201
- text modes *See* display, modes
- text panes 201, 243, 246
 - moving around in 201
- this parameter 89
 - watching 93
- tiled windows 46
- time delays, setting 80, 242
- Toggle command
 - Breakpoints menu 116
- Toggle command 105
- Trace Into command 78
 - continuous tracing 80
 - execution history and 82
 - programs executing in ROM and 251
- tracepoints 103, *See* breakpoints
- tracing 17, *See also* Trace Into command
 - backward *See* backward trace
 - continuous (animation) 80, 242
 - execution history and 81
 - into interrupts 80
 - into functions 48
 - program termination, and, information about 76
 - this parameter and 89, 93
- Tree command 151
- Turbo C++, versions compatible with TDW 2
- Turbo C++ for Windows
 - arrays 212
 - inspecting 97
 - problems with 96
 - automatic variables (autovariables) 214
 - scope 214
 - uninitialized 211
 - bugs specific to 211-215
 - character strings 142
 - compiler 211
 - constants 142
 - data
 - inspecting 95-98
 - types, converting 145
 - demo programs, debugging 217
 - dynamic virtual member functions 223
 - escape sequences 142
 - expressions 140-146, 212
 - with side effects 90, 144
 - #define macros and 214
 - functions 144
 - inspecting 97
 - problems with 90
 - returning from 214
 - integer assignment 213
 - keywords 145
 - keywords, problems with 256
 - loops, exiting 215
 - operators 212
 - expressions with side effects and 90, 144
 - precedence 143, 212

- pointers
 - incrementing and decrementing 212
 - inspecting 95
- pseudovariables 140, 141
- scalars, inspecting 95
- source code 213, 215
- structures, inspecting 96
- symbols 140
- unions, inspecting 96
- variables, return values 92
- Turbo C, expressions, problems with 249
- Turbo Pascal
 - units, override syntax 137
- type conversion 72, 94
 - memory handle to far pointer 175
 - problems with 256, 262
 - Turbo C++ for Windows reserved words and 145
- typecasting *See* type conversion
- types
 - data *See* data, types
 - object *See* objects, types

U

- UAE *See* unrecoverable application error
- unary operators
 - Turbo C++ for Windows 143
- Undo Close command 36
- union members, problems with 258
- units, scope override and 137
- unrecoverable application error (UAE) 227
- Until Return command 79
- UpdateWindow function 237
- User screen 30, 65
- User Screen command 30
- utilities, disk-based documentation for 11
- UTILS.TDW file 11

V

- variables 27, *See also* Variables window
 - accessing 132
 - problems with 254
 - with no type information 145
 - debugging 210
 - DLLs, accessing in 138

- evaluating 88-91
- global *See* global variables
- inactive functions and 250
- inspecting 31, 88, 94-100, 102, 246, *See also*
 - Inspector windows
 - function with same name as 71
 - in recursive functions 74
 - language conventions and 132
 - local *See* local variables
 - logging (breakpoints) 119
 - names 92
 - finding 27
 - problems with 258
 - pointing at 91
 - private 93
 - program termination and 84
 - return values 17, 90
 - inspecting 31
 - problems with 71, 95
 - scalar, character values and 95
 - scope override 135, 137
 - uninitialized 210
 - updating 93
 - viewing 70-73
 - in recursive routines 71
 - watching 27, 91, 92, 243, *See also* Watches
 - window
 - Clipboard, and 39
- Variables command 70
- Variables window 27, 70-73
 - local menu 198
 - opening 70
- VGA *See also* graphics adapters; video adapters
 - line display 66
- video adapters *See also* graphics adapters, hardware
 - display options 66
 - problems with 262
 - Super VGA support 2
 - supported 248
- Video Graphics Array Adapter *See* VGA
- videos *See* monitors; screens
- View menu 26, 188
- virtual member functions
 - dynamic 223

W

Watch command

- Global pane local menu 72
- Module window local menu 125
- Static/Local pane local menu 73
- Watches window local menu 93

Watch dialog box, global symbols and 72

Watches command 92

Watches window 27, 92-94

- local and static symbols and 73
- local menu 93, 199
- opening 92
- using 220
 - C tutorial 50

watchpoints 17, 103, *See also* breakpoints

- C tutorial 50
- Clipboard, and 39
- reloading programs and 85

wildcards

- DOS 127, 261
- searching with 126, 204

window handle

- accessing in ObjectWindows programs 231
- HWindow in MainWindow 232

Window menu 47, 190

- opening 33
- window management and 33

window messages

- breakpoints
 - Get Info message about 76
 - setting 164
 - setting in TDODEMO 230

logging

- to a file 165
- to the TDW window 158

window object

- adding to TDW Windows Messages window 161

Window Pick command 34

window selection

- adding to TDW Windows Messages window 159
- deleting from TDW Windows Messages window 162

Window Selection pane, local menu 195

Windows

- Display Windows Info command 166

exit code returned to 77

functions

- GlobalAlloc 166
- GlobalLock 167
- GlobalPageLock 167
- InvalidateRect, in TDODEMO 237
- InvalidateRgn 237
- LoadLibrary 172
- LocalAlloc 168
- LockData 167
- ReleaseCapture 235
- SetCapture 235
- UpdateWindow 237

Languages setting 2

reference books 7

returning to 67

running programs from 60

selectors, accessing 181

switching applications, TDW and 60

Task List, TDW and 60

TDW, and 157

tips, debugging tips 83

window messages

TDODEMO

- WM_LBUTTONDOWN 234
- WM_LBUTTONUP 234
- WM_MOUSEMOVE 234
- WM_NCMOUSEMOVE 234
- WM_PAINT 237

Windows Messages command 158

windows 17, 26-36

active 31

returning to 20

bottom line in 42

Breakpoints 26, 105-107

Clipboard 30, 196

closing 36

temporarily 35

CPU *See* CPU window

Dump 28, 184, 194

Execution History 29

opening 82

File 28, 194

opening 126

Hierarchy 149, 197

Inspector *See* Inspector windows

layout, saving 36, 67

- local menus and 22
- Log 27, 120, 120-122, 194
- Module *See* Module window
- mouse support 32-33
- moving 35
- moving around in 203
- multiple 33-34, 126, 130
 - moving among 33
- Numeric Processor 29, 197
 - problems with 257
- opening
 - duplicate 30
 - new 26
- panes *See* panes
- problems with 30, 258
 - current program location and 81
 - recovering last closed 36
 - reducing to icon 33, 35
- Registers 29, 184, 198
- repainting 36, *See also* display updating
- resizing 33, 35
- single-line borders and 35
- Stack 27, 73-74, 198
 - opening 71
- tiled 46
- tutorial 46
- Variables 27, 70-73, 198
 - opening 70
- Watches *See* Watches windows
- Windows Messages 30, 195
- Windows Information dialog box 166
- Windows Messages window
 - for an ObjectWindows program 160

- panes 195
 - standard Windows application 158
- WM_LBUTTONDOWN message 234
- WM_LBUTTONUP message 234
- WM_MOUSEMOVE message 234
 - capturing 235
- WM_NCMOUSEMOVE message 234
- WM_PAINT message
 - erase-screen bug 237
- WMLButtonDown function (TDODEMO)
 - bug
 - finding 234
 - fixing 229, 235
 - Scribble Window 226
- WMLButtonUp function (TDODEMO) 226
- WMMouseMove function (TDODEMO) 226
 - bug, finding 227
 - dynamic virtual member function, as a 224
- WMRButtonDown function (TDODEMO) 226
 - bug
 - finding 236
 - fixing 238
- word 180
 - formatting 183
- WordStar-style cursor-movement commands
 - 201, 202

Z

- zoom box 33
- Zoom command 35
- zoom icon 33