

TMS320C6x C Source Debugger User's Guide

Literature Number: SPRU188D
January 1998



Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Read This First

About This Manual

This book tells you how to use the TMS320C6x C source debugger with the following debugging tools to test and refine your code:

- ☐ Emulator
- ☐ Simulator
- ☐ Fast simulator

Each tool has its own version of the debugger. These versions operate almost identically; however, the executable files that invoke them are very different. Separate commands are provided for invoking each version of the debugger.

There are two debugger environments: the basic debugger environment and the profiling environment.

- ☐ The basic debugger environment is a general-purpose debugging environment. You can use standard data-management commands and run-type commands to test and evaluate your code.
- ☐ The profiling environment is a special environment for collecting statistics about code execution. You can use the profiling environment to identify areas in your code where you want to improve performance.

In addition to the debugger environment in the emulator version of the debugger, you can use the parallel debug manager (PDM). The PDM allows you to control and coordinate multiple debuggers, giving you the flexibility and power to debug your entire application for your multiprocessing system. The PDM and its functions and features are described in this book.

Before you use this book, you should install the C source debugger and any necessary hardware.

This book is meant to be used with the online help included with the C source debugger. The online help provides you with information about the windows, menu items, icons, and dialog boxes of the debugger interface. For information on how to access the online help, see section 1.7 on page 1-14.

Notational Conventions

This document uses the following conventions.

- ❑ The TMS320C6x family of devices is referred to as 'C6x.
- ❑ Debugger commands are not case sensitive; you can enter them in lower-case, uppercase, or a combination. To emphasize this fact, commands are shown throughout this user's guide in both uppercase and lowercase.
- ❑ Program listings and examples are shown in a special font. Some examples use a **bold version** to identify code, commands, or portions of an example that *you* enter. Here is an example:

Command	Result Displayed in the Command Window
whatis aai	int aai[10][5];
whatis xxx	struct xxx { int a; int b; int c; int f1 : 2; int f2 : 4; struct xxx *f3; int f4[10]; }

In this example, the left column identifies debugger commands that you type in. The right column identifies the result that the debugger displays in the display area of the Command window.

- ❑ In syntax descriptions, the instruction or command is in a **bold face**, and parameters are in *italics*. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are in *italics* describe the kind of information to be entered. Here is an example of a command syntax:

load *object filename*

load is the command. This command has one required parameter, indicated by *object filename*.

- ❑ Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you supply the information specified within the brackets; you do not enter the brackets themselves. Here is an example of a command that has an optional parameter:

run [*expression*]

The RUN command has one parameter, *expression*, which is optional.

- Braces ({ and }) indicate a list. The symbol | (read as *or*) separates items within the list. Here is an example of a list:

sound {on | off}

This provides two choices: **sound on** or **sound off**.

Unless the list is enclosed in square brackets, you must choose one item from the list.

Related Documentation From Texas Instruments

The following books describe the TMS320C6x and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C6x Assembly Language Tools User's Guide (literature number SPRU186) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C6x generation of devices.

TMS320C6x Optimizing C Compiler User's Guide (literature number SPRU187) describes the 'C6x C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the 'C6x generation of devices. The assembly optimizer helps you optimize your assembly code.

TMS320C62x/C67x CPU and Instruction Set Reference Guide (literature number SPRU189) describes the 'C62x/C67x CPU architecture, instruction set, pipeline, and interrupts for these digital signal processors.

TMS320C6201/C6701 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6201/C6701 digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port, serial ports, direct memory access (DMA), clocking and phase-locked loop (PLL), and the power-down modes.

TMS320C62x/C67x Programmer's Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C62x/C67x DSPs and includes application program examples.

TMS320C62x/C67x Technical Brief (literature number SPRU197) gives an introduction to the 'C62x/C67x digital signal processors, development tools, and third-party support.

XDS51x Emulator Installation Guide (literature number SPNU070) describes the installation of the XDS510™, XDS510PP™, and XDS510WS™ emulator controllers. The installation of the XDS511™ emulator is also described.

Related Documentation

If you are an assembly language programmer and would like more information about C or C expressions, you may find these books useful:

American National Standard for Information Systems—Programming Language C X3.159-1989, American National Standards Institute (ANSI standard for C)

Programming in C, Kochan, Steve G., Hayden Book Company

The C Programming Language (second edition, 1988), by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey

FCC Warning

This equipment is intended for use in a laboratory test environment only. It generates, uses, and can radiate radio frequency energy and has not been tested for compliance with the limits of computing devices pursuant to subpart J of part 15 of FCC rules, which are designed to provide reasonable protection against radio frequency interference. Operation of this equipment in other environments may cause interference with radio communications, in which case the user at his own expense will be required to take whatever measures may be required to correct this interference.

Trademarks

320 Hotline On-line is a trademark of Texas Instruments Incorporated.

PC is a trademark of International Business Machines Corporation.

SunOS and OpenWindows are trademarks of Sun Microsystems, Inc.

SPARCstation is trademark of SPARC International, Inc., but licensed exclusively to Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Windows, Windows 95, and Windows NT are registered trademarks of Microsoft Corporation.

XDS, XDS510, XDS510PP, XDS510WS, XDS511 are trademarks of Texas Instruments Incorporated.

X Window System is a trademark of the Massachusetts Institute of Technology.

If You Need Assistance . . .☐ **World-Wide Web Sites**

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.htm

☐ **North America, South America, Central America**

Product Information Center (PIC)	(972) 644-5580	
TI Literature Response Center U.S.A.	(800) 477-8924	
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285	
U.S. Technical Training Organization	(972) 644-5580	
DSP Hotline	(281) 274-2320	Fax: (281) 274-2324 Email: dsph@ti.com
DSP Modem BBS	(281) 274-2323	
DSP Internet BBS via anonymous ftp to	ftp://ftp.ti.com/pub/tms320bbs	

☐ **Europe, Middle East, Africa**

European Product Information Center (EPIC) Hotlines:		
Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32
Email: epic@ti.com		
Deutsch	+49 8161 80 33 11 or +33 1 30 70 11 68	
English	+33 1 30 70 11 65	
Francais	+33 1 30 70 11 64	
Italiano	+33 1 30 70 11 67	
EPIC Modem BBS	+33 1 30 70 11 99	
European Factory Repair	+33 4 93 22 25 40	
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10

☐ **Asia-Pacific**

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	
Taiwan DSP Internet BBS via anonymous ftp to	ftp://dsp.ee.tit.edu.tw/pub/TI/	

☐ **Japan**

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type "Go TIASP"	

☐ **Documentation**

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated	Email: dsph@ti.com
Technical Documentation Services, MS 702	
P.O. Box 1443	
Houston, Texas 77251-1443	

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

Contents

1	Overview of the Code Development and Debugging System	1-1
	<i>Provides an overview of the C source debugger, describes the code development environment, and provides a brief overview of the debugging process. Also tells you how to access online help.</i>	
1.1	Key Features of the Debugger	1-2
1.2	About the C Source Debugger Interface	1-3
	Descriptions of the debugger windows and their contents	1-4
1.3	Developing Code for the TMS320C6x	1-7
1.4	Limited Versions of the Simulator	1-10
	Floating-point version of the simulator	1-10
	Fast version of the fixed-point simulator	1-10
	Debugger features not supported by the limited versions of the simulator	1-11
1.5	About the Parallel Debug Manager (Emulator Only)	1-12
1.6	Overview of the Debugging Process	1-13
1.7	Accessing Online Help	1-14
	Accessing a list of help topics	1-14
	Accessing context-sensitive help	1-14
	Accessing help for debugger commands	1-15
2	Getting Started With the Debugger	2-1
	<i>Explains how to prepare your program for debugging and explains what you need to do before invoking the debugger. Explains how to invoke the debugger, and summarizes the debugger options. Describes the debugging modes and explains how to exit the debugger.</i>	
2.1	Preparing Your Program for Debugging	2-2
	Debugging optimized code	2-2
	Profiling optimized code	2-2
2.2	Identifying Alternate Directories for the Debugger to Search (D_DIR)	2-3
	Setting up D_DIR for Windows operating systems	2-3
	Setting up D_DIR for SPARC and HPUNIX operating systems	2-3
2.3	Identifying Directories That Contain Program Source Files (D_SRC)	2-4
	Setting up D_SRC for Windows operating systems	2-4
	Setting up D_SRC for SPARC and HPUNIX operating systems	2-4
2.4	Setting Up Default Debugger Options (D_OPTIONS)	2-5
	Setting up D_OPTIONS for Windows operating systems	2-5
	Setting up D_OPTIONS for SPARC and HPUNIX operating systems	2-5

2.5	Resetting the Emulator	2-6
2.6	Invoking the Debuggers and the PDM	2-7
	Invoking a stand-alone debugger	2-7
	Invoking multiple debuggers (emulator only)	2-8
2.7	Summary of Debugger Options	2-10
	Clearing the .bss section (-c option)	2-10
	Displaying the debugger on a different machine (-d option)	2-10
	Identifying a new configuration file (-f option)	2-11
	Identifying additional directories (-i option)	2-11
	Selecting big-endian format (-me option)	2-11
	Identifying the processor to be debugged (-n option)	2-11
	Identifying the port address (-p option)	2-12
	Entering the profiling environment (-profile option)	2-12
	Loading the symbol table only (-s option)	2-13
	Identifying a new initialization file (-t option)	2-13
	Loading without the symbol table (-v option)	2-13
	Ignoring D_OPTIONS (-x option)	2-13
2.8	Debugging Modes	2-14
	Auto mode	2-14
	Assembly mode	2-15
	Mixed mode	2-16
	Restrictions associated with debugging modes	2-17
2.9	Exiting the Debugger or the PDM	2-18
3	Entering and Using Commands	3-1
	<i>Tells you how to define your own command strings, enter operating system commands, and enter commands using a batch file.</i>	
3.1	Defining Your Own Command Strings	3-2
	Defining an alias	3-3
	Defining an alias with parameters	3-3
	Editing or redefining an alias	3-4
	Deleting an alias	3-4
	Considerations for using alias definitions	3-4
3.2	Entering Operating-System Commands From Within the Debugger	3-5
	Entering a single command from the debugger command line	3-5
	Entering several commands from a system shell	3-6
3.3	Creating and Executing a Batch File	3-7
	Echoing strings in a batch file	3-7
	Executing commands conditionally in a batch file	3-8
	Looping command execution in a batch file	3-9
	Pausing the execution of a batch file	3-11
	Executing a batch file	3-11
3.4	Creating a Log File to Reexecute a Series of Commands	3-12

4	Defining a Memory Map	4-1
	<i>Contains instructions for setting up a memory map that enables the debugger to access target memory correctly; includes hints about using a batch file to set up a memory map.</i>	
4.1	The Memory Map: What It Is and Why You Must Define It	4-2
	Potential memory map problems	4-2
4.2	Creating or Modifying the Memory Map	4-3
	Adding a range of memory	4-3
	Creating a customized memory type	4-4
	Deleting a range of memory	4-6
	Modifying a defined range of memory	4-6
4.3	Enabling Memory Mapping	4-7
4.4	A Sample Memory Map	4-9
4.5	Defining and Executing a Memory Map in a Batch File	4-10
	Defining a memory map in a batch file	4-10
	Executing a memory map batch file	4-11
4.6	Returning to the Original Memory Map	4-12
4.7	Using Multiple Memory Maps for Multiple Target Systems	4-13
4.8	Simulating I/O Space (Simulator Only)	4-14
	Connecting an I/O port	4-14
	Disconnecting an I/O port	4-15
4.9	Simulating External Interrupts (Simulator Only)	4-16
	Setting up your input file	4-16
	Connecting your input file to the interrupt pin	4-17
	Disconnecting your input file from the interrupt pin	4-18
	Listing the interrupt pins and connecting input files	4-18
5	Loading and Displaying Code	5-1
	<i>Tells you how to use the debugger modes to view the source files and how to load source files and object files.</i>	
5.1	Loading and Displaying Assembly Language Code	5-2
	Loading an object file and its symbol table	5-2
	Loading an object file without its symbol table	5-3
	Loading a symbol table only	5-3
	Loading code while invoking the debugger	5-3
	Displaying portions of disassembly	5-4
	Displaying assembly source code	5-5
5.2	Displaying C Code	5-6
	Displaying the contents of a text file	5-6
	Displaying a specific C function	5-7
	Displaying code beginning at a specific point	5-8

6	Running Code	6-1
	<i>Describes the basic run commands and single-step commands, tells you how to halt program execution, and discusses software breakpoints.</i>	
6.1	Defining the Starting Point for Program Execution	6-2
6.2	Using the Basic Run Commands	6-4
	Running an entire program	6-4
	Running code up to a specific point in a program	6-5
	Running the code in the current C function	6-6
	Running code while disconnected from the target system (emulator only)	6-6
	Running code through breakpoints	6-6
	Resetting the simulator	6-7
	Resetting the emulator	6-7
6.3	Single-Stepping Through Code	6-8
	Single-stepping through assembly language or C code	6-8
	Single-stepping through C code	6-9
	Continuously stepping through code	6-10
	Single-stepping through code and stepping over C functions	6-10
6.4	Running Code Conditionally	6-11
6.5	Benchmarking	6-12
6.6	Halting Program Execution	6-13
	What happens when you halt the emulator	6-13
6.7	Using Software Breakpoints	6-14
	Setting a software breakpoint	6-15
	Clearing a software breakpoint	6-17
	Clearing all software breakpoints	6-17
	Saving breakpoint settings	6-18
	Loading saved breakpoint settings	6-19
7	Managing Data	7-1
	<i>Describes the data-display windows and tells you how to edit data (memory contents, register contents, and individual variables).</i>	
7.1	Where Data Is Displayed	7-2
7.2	How the Emulator Displays Data for Load and Store Instructions	7-2
7.3	Basic Commands for Managing Data	7-3
	Determining the type of a variable	7-3
	Evaluating an expression	7-3
7.4	Basic Methods for Changing Data Values	7-5
	Editing data displayed in a window	7-5
	Editing data using expressions that have side effects	7-5
7.5	Managing Data in Memory	7-7
	Changing the memory range displayed in a Memory window	7-7
	Opening an additional Memory window	7-8
	Displaying memory contents while you are debugging C	7-9
	Saving memory values to a file	7-10
	Filling a block of memory	7-11

7.6	Managing Register Data	7-13
	Displaying register contents	7-13
	Accessing single-precision floating-point registers	7-16
	Accessing double-precision floating-point registers	7-17
7.7	Managing Data in a Watch Window	7-18
	Displaying data in a Watch window	7-19
	Displaying additional data	7-20
	Deleting watched values	7-20
7.8	Displaying Data in Alternative Formats	7-22
	Changing the default format for specific data types	7-22
	Changing the default format with data-management commands	7-24
8	Profiling Code Execution	8-1
	<i>Describes the profiling environment and tells you how to collect statistics about code execution.</i>	
8.1	Overview of the Profiling Environment	8-2
8.2	Overview of the Profiling Process	8-3
	A profiling strategy	8-3
8.3	Entering the Profiling Environment	8-4
8.4	Defining Areas for Profiling	8-5
	Marking an area with a mouse	8-5
	Marking an area with a dialog box	8-8
	Disabling an area	8-10
	Reenabling a disabled area	8-11
	Unmarking an area	8-12
	Restrictions on profiling areas	8-12
8.5	Defining a Stopping Point	8-15
	Setting a software breakpoint	8-15
	Clearing a software breakpoint	8-16
8.6	Running a Profiling Session	8-17
	Running a full or a quick profiling session	8-17
	Resuming a profiling session that has halted	8-19
8.7	Viewing Profile Data	8-20
	Viewing different profile data	8-21
	Sorting profile data	8-23
	Viewing different profile areas	8-24
	Interpreting session data	8-25
	Viewing code associated with a profile area	8-25
8.8	Saving Profile Data to a File	8-27
	Saving the contents of the Profile window	8-27
	Saving all data for currently displayed areas	8-28

9	Using Simulator Memory System Analysis	9-1
	<i>Explains how to count or set breakpoints on various CPU events by using the analysis menu and dialog box. Also describes the memory system analysis commands and provides a sample batch file using these commands.</i>	
9.1	Major Functions of Simulator Memory System Analysis	9-2
	Set up event breakpoints	9-2
	Count system events	9-2
9.2	Overview of the Analysis Process	9-3
9.3	Enabling Memory System Analysis	9-4
9.4	Defining the Conditions for Analysis	9-5
	Description of available system events	9-6
	Counting system events	9-6
	Setting event breakpoints	9-7
	Removing a defined count or break event	9-7
9.5	Running Your Program	9-8
9.6	Viewing the Analysis Data	9-9
	Interpreting the information in the Analysis Statistics window	9-9
	Resetting the event counters	9-9
9.7	Summary of Memory System Analysis Commands	9-10
	event_enable (enable specified event)	9-10
	event_disable (disable specified event)	9-11
	event_break (set breakpoint on specified event)	9-11
	event_counter_start (count each occurrence of specified event)	9-11
	event_counter_reset (reset counter for specified event)	9-12
	event_reset (disable and clear configuration for all events)	9-12
	event_list (list configuration of all events)	9-12
9.8	Entering Analysis Commands Through a Batch File	9-13
10	Monitoring Hardware Functions With the Emulator Analysis Module	10-1
	<i>Describes the analysis environment for the emulator and tells you how to set hardware breakpoints.</i>	
10.1	Major Functions of the Analysis Module	10-2
10.2	Overview of the Analysis Process	10-3
10.3	Enabling the Analysis Module	10-4
10.4	Defining the Conditions for Analysis	10-5
	Counting events	10-6
	Enabling the external counter	10-7
	Setting hardware breakpoints	10-8
	Setting up the EMU0/1 pins to set global breakpoints	10-9
10.5	Running Your Program	10-10
	How to run the entire program	10-10
	How the Run Benchmarks (RUNB) command affects analysis	10-10
10.6	Viewing the Analysis Data	10-11
	Interpreting the information in the Analysis Statistics window	10-11

10.7	Creating Customized Analysis Commands	10-12
10.8	Summary of Analysis Pseudoregisters	10-13
	AEN (enable analysis)	10-13
	ABE (configure hardware breakpoints)	10-13
	ADR (program address breakpoint value)	10-13
	ACE (configure analysis counter events)	10-14
	ICNT (internal counter value)	10-14
	XCNT (external counter value)	10-14
	AST (analysis status)	10-14
11	Using the Parallel Debug Manager	11-1
	<i>Describes the parallel debug manager (PDM) for the TMS320C6x system, tells you how to invoke the PDM and individual debuggers, and describes execution-related commands. Also includes information about describing your target system in a configuration file.</i>	
11.1	Identifying Processors and Groups	11-2
	Assigning names to individual processors	11-2
	Organizing processors into groups	11-3
11.2	Sending Debugger Commands to One or More Debuggers	11-6
11.3	Running and Halting Code	11-7
	Halting processors at the same time	11-8
	Sending ESCAPE to all processors	11-8
	Finding the execution status of a processor or a group of processors	11-8
11.4	Entering PDM Commands	11-9
	Executing PDM commands from a batch file	11-9
	Recording information from the PDM display area	11-10
	Controlling PDM command execution	11-10
	Echoing strings to the PDM display area	11-12
	Pausing command execution	11-13
	Using the command history	11-13
11.5	Defining Your Own Command Strings	11-15
11.6	Entering Operating-System Commands	11-16
11.7	Understanding the PDM's Expression Analysis	11-17
11.8	Using System Variables	11-18
	Creating your own system variables	11-18
	Assigning a variable to the result of an expression	11-19
	Changing the PDM prompt	11-19
	Checking the execution status of the processors	11-20
	Listing system variables	11-20
	Deleting system variables	11-20
11.9	Evaluating Expressions	11-21

12 Summary of Commands	12-1
<i>Provides functional and alphabetical summaries of the basic debugger commands and the profiling commands.</i>	
12-1	
12.1 Functional Summary of Debugger Commands	12-2
Managing multiple debuggers	12-3
Changing modes	12-4
Managing windows	12-4
Customizing the screen	12-4
Displaying files and loading programs	12-4
Displaying and changing data	12-5
Performing system tasks	12-6
Managing breakpoints	12-7
Memory mapping	12-7
Running programs	12-8
Profiling commands	12-9
Memory system analysis commands (simulator only)	12-10
12.2 Alphabetical Summary of Debugger and PDM Commands	12-11
12.3 Summary of Profiling Commands	12-62
13 Basic Information About C Expressions	13-1
<i>Many of the debugger commands accept C expressions as parameters. This chapter provides general information about the rules governing C expressions and describes specific implementation features related to using C expressions as command parameters.</i>	
13.1 C Expressions for Assembly Language Programmers	13-2
13.2 Using Expression Analysis in the Debugger	13-4
Restrictions	13-4
Additional features	13-4
A What the Debugger Does During Invocation	A-1
<i>In some circumstances, you may find it helpful to know the steps that the debugger goes through during the invocation process; this appendix lists these steps.</i>	
B Describing Your Target System to the Debugger	B-1
<i>Explains how to supply the information about your target configuration to the debugger.</i>	
B.1 Step 1: Create the Board Configuration Text File	B-2
B.2 Step 2: Translate the Configuration File to a Debugger-Readable Format	B-5
B.3 Step 3: Specify the Configuration File When Invoking the Debugger	B-6
C Debugger Messages	C-1
<i>Describes progress and error messages that the debugger may display.</i>	
C.1 Associating Sound With Error Messages	C-2
C.2 Alphabetical Summary of Debugger Messages	C-2
C.3 Alphabetical Summary of PDM Messages	C-22
C.4 Additional Instructions for Expression Errors	C-26
C.5 Additional Instructions for Hardware Errors	C-26
D Glossary	D-1
<i>Defines acronyms and key terms used in this book.</i>	

Figures

1-1	The Basic Debugger Display	1-3
1-2	TMS320C6x Software Development Flow	1-7
1-3	The PDM Environment	1-12
2-1	Typical Assembly Display (for Auto Mode and Assembly Mode)	2-15
2-2	Typical C Display (for Auto Mode Only)	2-16
2-3	Typical Mixed Display (for Mixed Mode Only)	2-17
4-1	Sample Memory Map for Use With a TMS320C6x Simulator	4-9
7-1	The Default Memory Window	7-7
7-2	Reordering Registers in the CPU Window Using the Drag-and-Drop Method	7-14
8-1	An Example of the Profile Window	8-20
8-2	Cycling Through the Profile Window Fields	8-22
9-1	Enabling/Disabling the Analysis Interface	9-4
9-2	Analysis Events Dialog Box	9-5
9-3	Analysis Statistics Window Displaying an Ongoing Status Report	9-9
10-1	Enabling/Disabling the Analysis Module	10-4
10-2	Analysis Events Dialog Boxes	10-5
10-3	EMU1 Pin Set Up to Trigger Out on Hardware Break Events	10-9
10-4	Analysis Statistics Window Displaying an Ongoing Status Report	10-11
11-1	Grouping Processors	11-3

Tables

1–1	Summary of Debugger Window Descriptions	1-5
2–1	Summary of Debugger Options	2-10
3–1	Predefined Constants for Use With Conditional Commands	3-8
7–1	Pseudoregister Names for Single-Precision Floating-Point Registers	7-16
7–2	Pseudoregister Names for Double-Precision Floating-Point Registers	7-17
7–3	Display Formats for Debugger Data	7-22
7–4	Data Types for Displaying Debugger Data	7-23
8–1	Debugger Commands That Can/Cannot Be Used in the Profiling Environment	8-4
8–2	Using the Profile Marking Dialog Box to Mark Areas	8-9
8–3	Disabling, Enabling, Unmarking, or Viewing Areas	8-13
8–4	Types of Data Shown in the Profile Window	8-21
9–1	Description of Analysis Counter Events	9-6
9–2	Memory System Analysis Command Summary	9-10
10–1	Description of Analysis Counter Events	10-6
11–1	PDM Operators	11-17
12–1	Marking areas	12-62
12–2	Disabling marked areas	12-62
12–3	Enabling disabled areas	12-63
12–4	Unmarking areas	12-64
12–5	Changing the profile window display	12-65

Overview of the Code Development and Debugging System

The C source debugger is an advanced programmer's interface that helps you to develop, test, and refine 'C6x C programs (compiled with the 'C6x optimizing ANSI C compiler) and assembly language programs. The debugger is the interface to the 'C6x simulator and the scan-based emulator.

This chapter gives an overview of the C source debugger, describes the code development environment, and explains how you must prepare your program for debugging. This chapter also describes the parallel debug manager (PDM) for use with the 'C6x emulator.

You can access context-sensitive online help at any time during the debugging process to explain the functions of the windows, dialog boxes, and menus of the debugger interface. This chapter also explains how to access online help and how to exit the debugger when you have completed your debugging session.

Topic	Page
1.1 Key Features of the Debugger	1-2
1.2 About the C Source Debugger Interface	1-3
1.3 Developing Code for the TMS320C6x	1-7
1.4 Limited Versions of the Simulator	1-10
1.5 Description of the Parallel Debug Manager (Emulator Only)	1-12
1.6 Overview of the Debugging Process	1-13
1.7 Accessing Online Help	1-14

1.1 Key Features of the Debugger

- ❑ **Multilevel debugging.** The debugger allows you to debug both C and assembly language code. If you are debugging a C program, you can choose to view only the C source, the disassembly of the object code created from the C source, or both. You can also use the debugger as an assembly language debugger and view the original assembly source code.
- ❑ **Fully configurable graphical user interface.** The C source debugger separates code, data, and commands into manageable portions. The graphical user interface is intuitive and follows the conventions used by your windowing system.
- ❑ **Comprehensive data displays.** You can easily create windows for displaying and editing the values of variables, arrays, structures, pointers—any kind of data—in their natural format (float, int, char, enum, or pointer). You can even display entire linked lists.
- ❑ **On-screen editing.** You can change any data value displayed in any window—just click and type.
- ❑ **Automatic update.** The debugger automatically updates information on the screen, highlighting changed values.
- ❑ **Dynamic profiling.** In addition to the basic debugging environment, a second environment—the *profiling environment*—is available. The profiling environment provides a method for collecting execution statistics about specific areas in your code. This gives you immediate feedback on your application's performance and helps you identify bottlenecks within the code.
- ❑ **Analysis module.** In addition to the basic debugger features, the 'C6x has an analysis module on the chip that allows the emulator to monitor the operations of your target system. This expands your debugging capabilities beyond simple software breakpoints.
- ❑ **All the standard features you expect in a world-class debugger.** The debugger provides you with complete control over program execution with features like conditional execution and single-stepping (including single-stepping into or over function calls). You can set or clear a breakpoint with a click of the mouse. You can define a memory map that identifies the portions of target memory that the debugger can access. The debugger can execute commands from a batch file, providing you with an easy method for entering often-used command sequences.

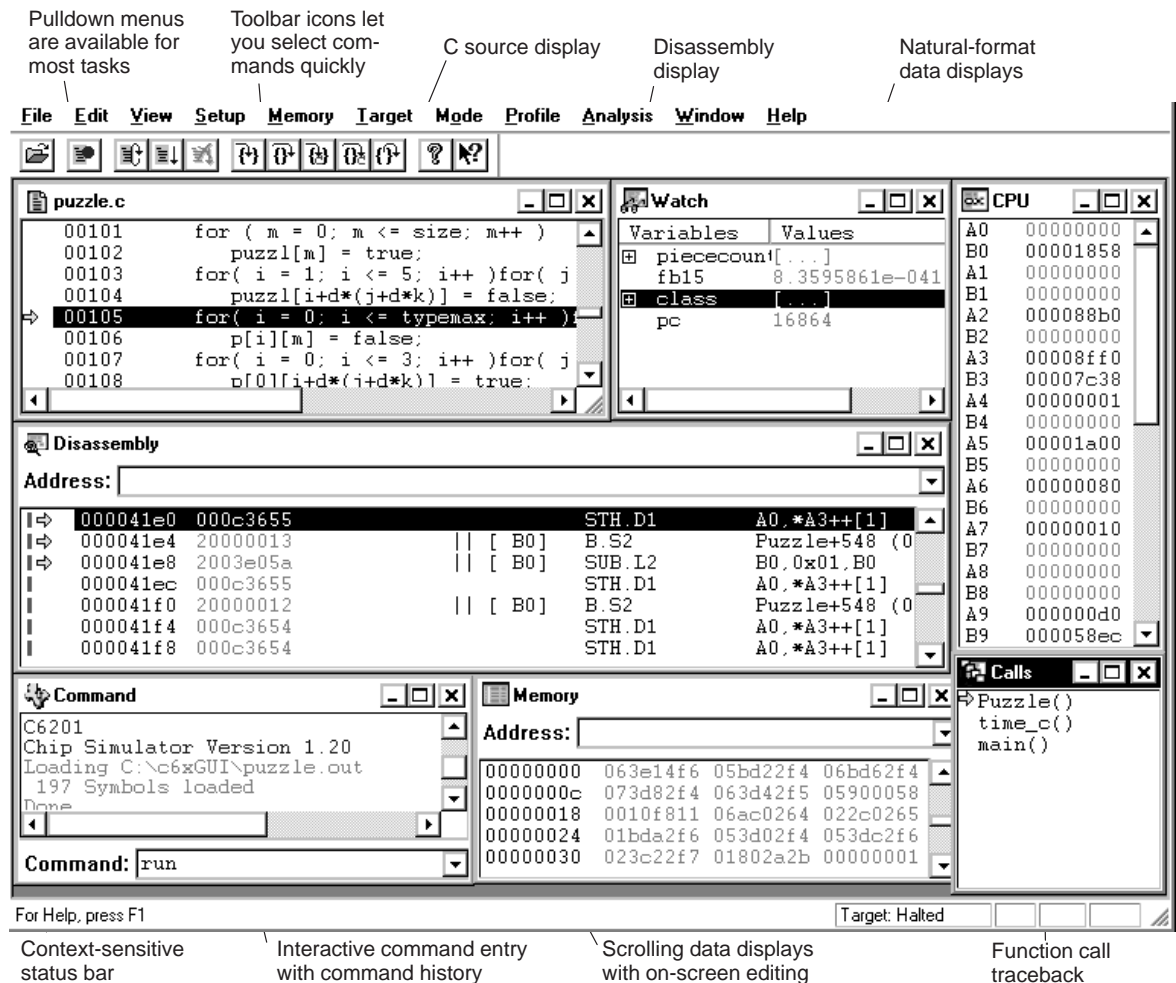
1.2 About the C Source Debugger Interface

The C source debugging interface improves productivity by allowing you to debug a program in the language it was written in. You can choose to debug your programs in C, assembly language, serial assembly, or all three.

The Texas Instruments advanced programmer's interface follows the conventions used by your windowing system, reducing learning time and eliminating the need to memorize complex commands. A shortened learning curve and increased productivity reduce the software development cycle, so you can get to market faster.

Figure 1–1 identifies several features of the debugger display.

Figure 1–1. The Basic Debugger Display



Descriptions of the debugger windows and their contents

The debugger can show several types of windows. Each type of window serves a specific purpose and has unique characteristics. Every window is identified by a name in its upper left corner. For the File window, the debugger displays the name of the file shown in the window instead of the word File. There are eight different windows, divided into these general categories:

- ☐ Code-display windows display assembly language or C code. There are three code-display windows:
 - A File window displays any text file that you want to display; its main purpose, however, is to display C source code. You can display multiple File windows at one time.
 - The Disassembly window displays the disassembly (assembly language version) of memory contents.
 - The Calls window identifies the current function and previous function calls if you are debugging a C program.
- ☐ The Profile window displays statistics about code execution.
- ☐ Data-display windows are for observing and modifying various types of data. There are three data-display windows:
 - A Memory window displays the contents of a range of memory. You can display multiple Memory windows to allow you to view different sections of memory at one time.
 - The CPU window displays the contents of 'C6x registers.
 - A Watch window displays selected data such as variables, specific registers, or memory locations. You can display multiple Watch windows to allow you to view multiple variables, register, or memory locations at one time.
- ☐ The Command window provides an area for typing in commands and re-entering commands and an area for displaying various types of information, such as progress messages, error messages, or command output.

Table 1–1 summarizes the purpose of each window, how each window is created, and in which debugging mode each window is visible.

Table 1–1. Summary of Debugger Window Descriptions

Window	Purpose	Created	Mode
Calls	Lists the current function, its caller, and the caller's caller, etc. for C functions	<input type="checkbox"/> Automatically when you are displaying C code <input type="checkbox"/> With the CALLS command if you previously closed the Calls window	<input type="checkbox"/> Auto <input type="checkbox"/> Mixed
Command	<input type="checkbox"/> Provides a command line for entering commands <input type="checkbox"/> Provides a display area for echoing commands and displaying command output, errors, and messages	Automatically	All
CPU	Shows the contents of the 'C6x registers	Automatically	All
Disassembly	Displays the disassembly (or reverse assembly) of memory contents	Automatically	All
File	<input type="checkbox"/> Displays C source files <input type="checkbox"/> Displays assembly source files <input type="checkbox"/> Displays text files <input type="checkbox"/> Displays serial assembly files assembly optimized with <code>-g</code> option	<input type="checkbox"/> With the File→Open menu option <input type="checkbox"/> Automatically when your program executes C code, assembly code, or serial assembly code assembled with the <code>-g</code> assembler option	<input type="checkbox"/> Auto <input type="checkbox"/> Mixed
Memory	Displays the contents of memory. Reference addresses, determined by the size of the window, are listed in the first column.	<input type="checkbox"/> Automatically for the default Memory window only <input type="checkbox"/> With the MEM command and a unique <i>window name</i> for additional Memory windows	All
Profile	Displays statistics collected during a profiling session	By entering the profiling environment: Profile→Profile Mode	Mixed
Watch	Displays the values of selected expressions, structures, arrays, or pointers	<input type="checkbox"/> With the Setup→Watch Variable menu option <input type="checkbox"/> With the WA and DISP commands	All

All of the windows have context menus that allow you to display or hide information in a window and control how a window is displayed. To display a context menu, follow these steps:

- 1) Move your pointer over a debugger window.
- 2) Click the right mouse button. This displays a context menu like the following example:

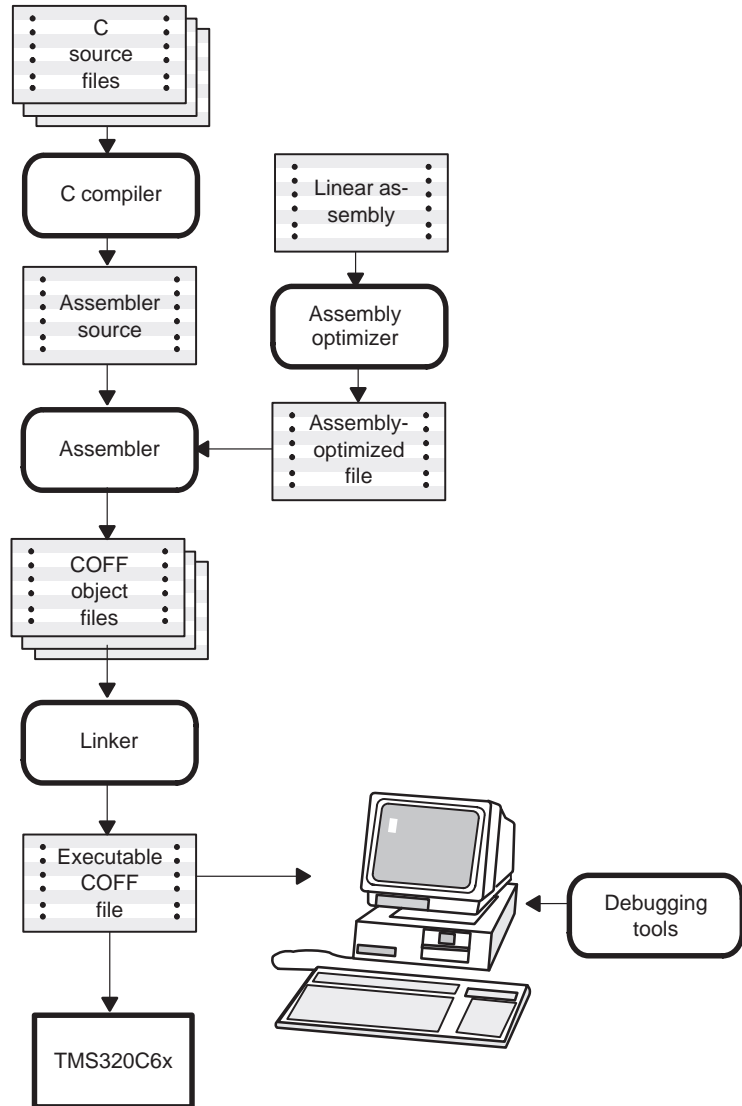


Each context menu option that is currently selected has a check mark (✓) preceding it, and those that are unselected do not. Clicking an option toggles between selected and unselected.

1.3 Developing Code for the TMS320C6x

The 'C6x is well supported by a complete set of hardware and software development tools, including a C compiler, an assembly optimizer, an assembler, and a linker. Figure 1–2 illustrates the basic 'C6x code development flow.

Figure 1–2. TMS320C6x Software Development Flow



Common object file format (COFF) allows you to divide your code into logical blocks, define your system's memory map, and then link code into specific memory areas. COFF also provides rich support for source-level debugging.

The following list describes the tools shown in Figure 1–2.

- ❑ The **assembly optimizer** allows you to write linear assembly code without being concerned with the TMS320C6x pipeline structure or with assigning registers. It assigns registers and uses loop optimization to turn linear assembly into highly parallel assembly that takes advantage of software pipelining.

See the *TMS320C6x Optimizing C Compiler User's Guide* for more information.

- ❑ The **C compiler** accepts C source code and produces TMS320C6x assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are included in the compiler package:

- The shell program enables you to compile, assemble, and link source modules in one step.
- The optimizer modifies code to improve the efficiency of C programs.
- The interlist utility interlists C source statements with assembly language output to correlate code produced by the compiler with your source code.

See the *TMS320C6x Optimizing C Compiler User's Guide* for more information.

- ❑ The **assembler** translates assembly language source files into machine language COFF object files.

See the *TMS320C6x Assembly Language Tools User's Guide* for more information.

- ❑ The **linker** combines object files into a single executable COFF object module. As it creates the executable module, it performs relocation and resolves external references. The linker allows you to define your system's memory map and to associate blocks of code with defined memory areas.

See the *TMS320C6x Assembly Language Tools User's Guide* for more information.

- ❑ The main product of this development process is a module that can be executed in a **TMS320C6x target system**.

- You can use debugging tools to refine and correct your code. Available products include:
 - An instruction-accurate and clock-accurate fixed-point software simulator
 - An instruction-accurate, limited floating-point software simulator
 - A faster, limited version of the fixed-point simulator
 - An XDS™ emulator

1.4 Limited Versions of the Simulator

This release provides you with three simulator versions:

- ☐ Fixed-point simulator for use with the TMS320C6201
- ☐ Limited, faster version of the fixed-point simulator for use with the TMS320C6201
- ☐ Limited, floating-point simulator for use with the TMS320C6701

The fixed-point simulator provides you with full debugger functionality. The limited versions of the simulator allow you to do specialized tasks, but do not support the full range of debugger capabilities. The limited versions of the simulator are separate executables, available on the following platforms:

- ☐ UNIX:
 - Sun SPARC
 - HP
- ☐ PC:
 - Windows NT
 - Windows 95

Floating-point version of the simulator

Use the floating-point version of the simulator with the TMS320C6701. The floating-point version of the simulator supports the TMS320C67x instruction set and is a separate executable.

For information on invoking the floating-point version of the simulator executable, see page 2-7.

Fast version of the fixed-point simulator

The fast version of the fixed-point simulator simulates at almost 50 times the rate of the standard fixed-point simulator. The fast simulator supports the complete 'C6x instruction set as well as both big-endian and little-endian modes.

Use the fast simulator to quickly validate code when information such as cycle-count accuracy is not critical.

For information on invoking the floating-point version of the simulator executable, see page 2-7.

Debugger features not supported by the limited versions of the simulator

The floating-point version of the simulator and the fast version of the fixed-point simulator do not support the following features of the debugger:

☐ Internal memory map checks

The simulator core for the limited simulators do not perform dynamic map checks. The limited simulators assume that all memory is available. Therefore, if your code accesses unmapped memory locations, the limited simulators do not give you an error message.

☐ Internal program RAM as cache

The limited simulators do not allow you to configure Internal program RAM as cache.

☐ On-chip data memory conflicts

The limited simulators cannot detect memory bank conflicts in on-chip data memory.

☐ Simulation of memory-mapped I/O through file connects to memory

The limited simulators do not support the MC and MI commands.

☐ Random interrupts

The limited simulators do not allow you to generate random interrupts.

☐ On-chip peripheral registers

The limited simulators do not model any of the 'C6x peripherals.

☐ Accurate cycle-count

The cycle count of the limited simulators do not account for stalls due to external accesses and memory bank conflicts.

☐ Analysis

The limited simulators do not support analysis capability. You cannot count system events or set up breakpoints on system events while using either of the limited versions of the simulator.

1.5 About the Parallel Debug Manager (Emulator Only)

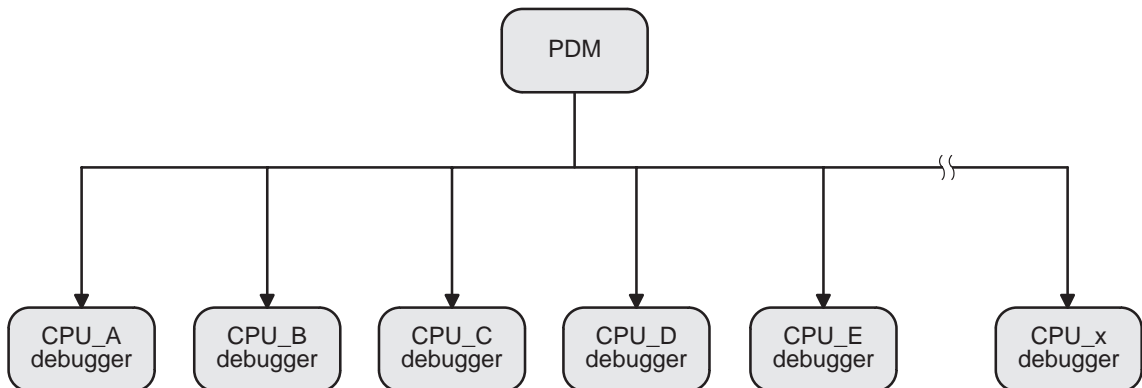
The TMS320C6x emulation system is a true multiprocessing debugging system. It allows you to debug your entire application by using the parallel debug manager (PDM). The PDM is a command shell that controls and coordinates multiple debuggers, providing you with the ability to:

- ☐ Create and control debuggers for one or more processors
- ☐ Organize debuggers into groups
- ☐ Send commands to one or more debuggers
- ☐ Synchronously run, step, and halt multiple processors in parallel
- ☐ Gather system information in a central location

The PDM is invoked and PDM commands are executed from a command shell window under the host windowing system. From the PDM, you can invoke and control debuggers for each of the processors in your multiprocessing system.

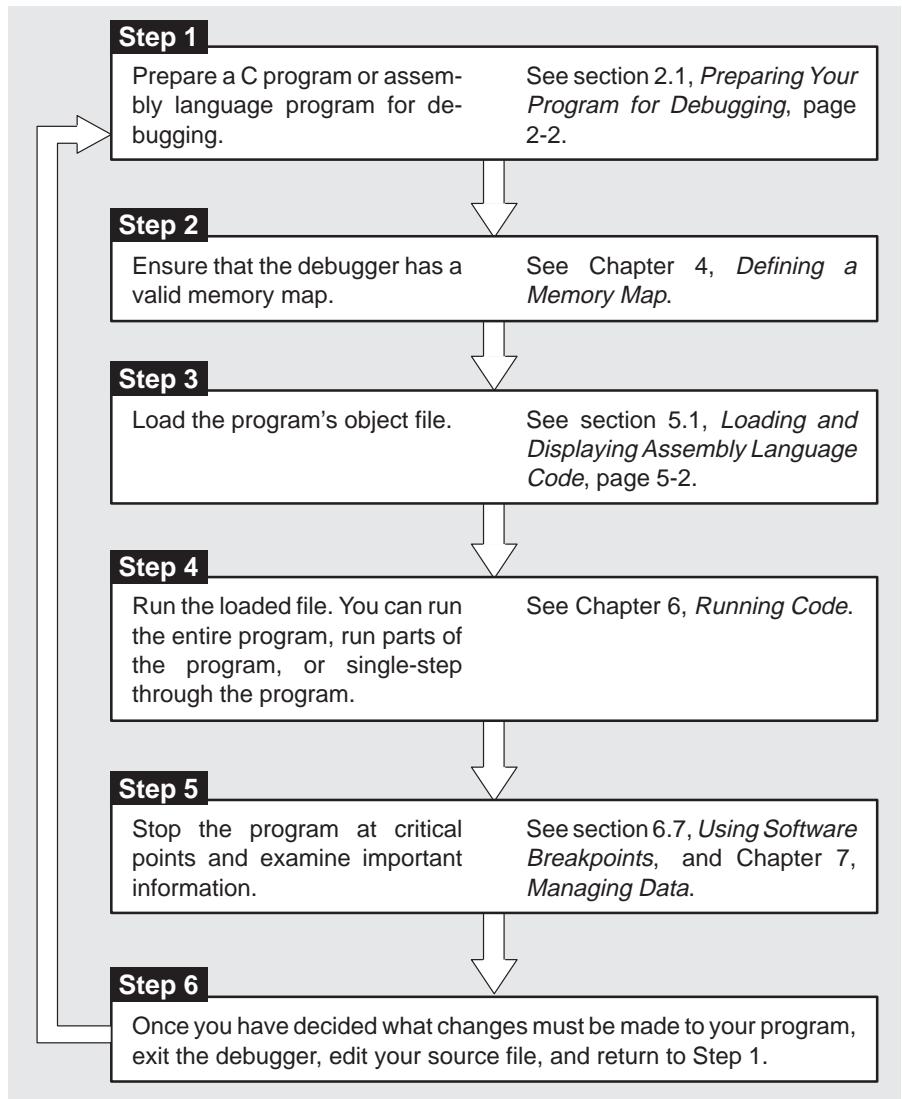
As Figure 1–3 shows, you can run multiple debuggers under the control of the PDM.

Figure 1–3. The PDM Environment



1.6 Overview of the Debugging Process

Debugging a program is a multiple-step process. These steps are described below, with references to parts of this book that help you accomplish each step.



1.7 Accessing Online Help

Online help is available to provide information about menu options, dialog boxes, debugger windows, and debugger commands.

Accessing a list of help topics

To display a list of help topics, follow these steps:

- 1) Open the list of help topics by using one of these methods:

- ☐ Click the Help Topics icon on the toolbar:



- ☐ From the Help menu, select Help Topics.

- ☐ From the command line, enter:

`help` 

- 2) Double-click the topic that you want to view.

Accessing context-sensitive help

You can access context-sensitive help using the following methods:

- ☐ To find out about an item in the debugger display, follow these steps:

- 1) Click the Help icon on the toolbar:



This changes the pointer to a question mark.

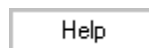
- 2) Select the menu option or click on the item that you want more information about.

- ☐ To find out about a dialog box or a window, follow these steps:

- 1) Make the window or the dialog box active.

- 2) Press .

For all dialog boxes, you can also click the Help button in that dialog box to view context-sensitive help:



Accessing help for debugger commands

To find out about a specific debugger command, use the HELP command. The syntax for this command is:

help *debugger command*

The HELP command opens a help topic that describes the *debugger command*.

Getting Started With the Debugger

Before or after you install the debugger, you can define environment variables that set certain debugger parameters you normally use. When you use environment variables, default values are set, making each individual invocation of the debugger simpler because these parameters are automatically specified. When you invoke the debugger, you can use command-line options to override many of the defaults that are set with environment variables. These options are summarized in this chapter.

Once you have set up the environment variables and invoked the debugger, you must select the correct debugging mode for your program. This chapter describes these debugging modes and provides an overview of the debugging process.

Topic	Page
2.1 Preparing Your Program for Debugging	2-2
2.2 Identifying Alternate Directories for the Debugger to Search (D_DIR)	2-3
2.3 Identifying Directories That Contain Program Source Files (D_SRC)	2-4
2.4 Setting Up Default Debugger Options (D_OPTIONS)	2-5
2.5 Resetting the Emulator	2-6
2.6 Invoking the Debuggers and the PDM	2-7
2.7 Summary of Debugger Options	2-10
2.8 Debugging Modes	2-14
2.9 Exiting the Debugger or the PDM	2-18

2.1 Preparing Your Program for Debugging

Before you use the debugger, you must create an executable object file. To do so, start with C source, assembly optimizer source, and/or assembly language code. You can use the cl6x shell program to compile, assemble, and link your source code, creating an executable object file. To be able to debug the object file, you must use the `-g` shell option. The `-g` option generates symbolic debugging directives that are used by the debugger.

If you want to profile the execution of the object file, you must use the `-as` shell option. The `-as` option puts labels in the symbol table. Label definitions are written to the COFF symbol table for use with symbolic debugging.

For more information about the cl6x shell program and its options and about creating an executable object file for use with the debugger, see the *TMS320C6x Optimizing C Compiler User's Guide*.

Debugging optimized code

If you intend to *debug* optimized code, use the `-g` shell option with the `-o` shell option. The `-g` option generates symbolic debugging directives that are used by the debugger for C source debugging, but it disables many compiler optimizations. When you use the `-o` option (which invokes the optimizer) with the `-g` option, you turn on the maximum amount of optimization that is compatible with debugging. The `-o` option applies only to C code, not to assembly.

If you have trouble debugging loops in your code, you can use the `-mu` shell option to turn off software pipelining. Software-pipelined loops are sometimes difficult to debug because the code is not presented serially.

Profiling optimized code

If you intend to *profile* optimized code, use the `-mg` shell option with the `-g` and `-o` options. The `-mg` option allows you to profile optimized code by turning on the maximum amount of optimization that is compatible with profiling. When you combine the `-g` and `-o` options with the `-mg` option, all of the line directives are removed except for the first one and the last one.

2.2 Identifying Alternate Directories for the Debugger to Search (D_DIR)

The debugger uses the information you provide via the D_DIR environment variable to locate the directory that contains the auxiliary files (such as `siminit.cmd` or `emuinit.cmd`) that it needs.

Setting up D_DIR for Windows operating systems

To set the D_DIR environment variable for Windows™ operating systems, use this syntax:

```
SET D_DIR=pathname1[:pathname2 . . . ]
```

For example, to set up a directory named `tools_dir` for auxiliary files on your hard drive, enter:

```
SET D_DIR=c:\tools_dir
```

(Be careful not to precede the equal sign with a space.)

Setting up D_DIR for SPARC and HPUX operating systems

To set the D_DIR environment variable for Sparc™ and HPUX™ operating systems, use this syntax:

```
setenv D_DIR "pathname"
```

If you are using SunOS™ :

☐ For C shells:

```
setenv D_DIR "pathname"
```

☐ For Bourne or Korn shells:

```
D_DIR="pathname"  
export D_DIR
```

(Be sure to enclose the directory name within quotes.)

2.3 Identifying Directories That Contain Program Source Files (D_SRC)

The debugger uses the information you provide via the D_SRC environment variable to locate the directories that contain program source files that you want to access from the debugger.

Setting up D_SRC for Windows operating systems

To set the D_SRC environment variable for a Windows operating system, use this syntax:

```
SET D_SRC=pathname1[;pathname2 . . .]
```

For example, if your 'C6x programs were in a directory named source on drive C, the D_SRC setup would be:

```
SET D_SRC=c:\source
```

(Be careful not to precede the equal sign with a space.)

Setting up D_SRC for SPARC and HP/UX operating systems

To set the D_SRC environment variable for a Solaris or HP/UX operating system, use this syntax:

```
setenv D_SRC "pathname1[;pathname2;...]"
```

If you are using SunOS:

- ☐ For C shells:

```
setenv D_SRC "pathname1[;pathname2 . . .]"
```

- ☐ For Bourne or Korn shells:

```
D_SRC="pathname"  
export D_SRC
```

(Be sure to enclose the path names within one set of quotes.)

2.4 Setting Up Default Debugger Options (D_OPTIONS)

Use the D_OPTIONS environment variable to set the debugger invocation options that you want to use regularly. When you use the D_OPTIONS environment variable, the debugger uses the default options and/or input filenames that you name with D_OPTIONS every time you invoke the debugger.

Setting up D_OPTIONS for Windows operating systems

To set the D_OPTIONS environment variable for Windows operating systems, use this syntax:

SET D_OPTIONS= [*filename*] [*options*]

(Be careful not to precede the equal sign with a space.)

The *filename* identifies the optional object file for the debugger to load, and *options* lists the options you want to use at invocation. Section 2.7 on page 2-10 summarizes the options that you can identify with D_OPTIONS.

Setting up D_OPTIONS for SPARC and HP/UX operating systems

To set the D_OPTIONS environment variable for SPARC and HP/UX operating systems, use this syntax:

setenv D_OPTIONS "[*filename*] [*options*]"

If you are using SunOS:

☐ For C shells:

setenv D_OPTIONS "[*filename*] [*options*]"

☐ For Bourne or Korn shells:

D_OPTIONS="[*filename*] [*options*]"
export D_OPTIONS

(Be sure to enclose the filename and options within one set of quotes.)

The *filename* identifies the optional object file for the debugger to load, and *options* list the options you want to use at invocation. Section 2.7 summarizes the options that you can identify with D_OPTIONS.

2.5 Resetting the Emulator

You must reset the emulator *before* invoking the debugger. Reset can occur only after you have powered up the target board. You can reset the emulator by adding the following command to the autoexec.bat file:

emurst [-x] [-p *number*]

The `-x` option tells the emurst utility to ignore any options specified with the `D_OPTIONS` environment variable. For more information about `-x`, see page 2-13.

The `-p` option *number* identifies the I/O port address that the debugger uses for communicating with the emulator. For more information about `-p`, see page 2-12.

If the following message appears after the emulator is reset, you have a hardware error:

CANNOT DETECT TARGET POWER

One of several problems can cause this error message to appear. Answer each of the following questions about your system and restart your PC. Check:

- ☐ Is the emulator board installed snugly?
- ☐ Is the cable connecting your emulator and target system loose?
- ☐ Is the target power on?
- ☐ Is your target board getting the correct voltage?
- ☐ Is your emulator scan path uninterrupted?
- ☐ Is your port address set correctly?
- ☒ Ensure that the `-p` option's parameter matches the I/O address defined by your switch settings. For information about the switch settings, see the *XDS51x Emulator Installation Guide*.
- ☒ Check to ensure that the address you entered as the `-p` option's parameter does not conflict with the address space with another bus setting. If you have a conflict, change the switches on your board to one of the alternate settings. Modify the `-p` option's parameter to reflect the change in your switch settings.

2.6 Invoking the Debuggers and the PDM

If you are using an emulator, there are two ways to invoke the debugger:

- ☐ You can invoke a stand-alone debugger that is *not* controlled by the parallel debug manager (PDM).
- ☐ You can invoke several debuggers that are under control of the PDM.

If you are using a simulator, you can invoke *only* a standalone debugger.

This section describes how to invoke any version of the debugger and how to invoke the PDM.

Invoking a stand-alone debugger

To invoke the debugger on a PC™, use one of the following methods:

- ☐ Double-click the shortcut icon for the debugger.
- ☐ From the Start menu, select Run.... Enter the path for the debugger executable file.

You can specify debugger options at invocation by modifying the command line in the property sheet for your debugger icon.

To invoke the debugger on a SPARCstation™, enter the following command from a command shell:

sim6x | emu6x | sim67x | sim6xfast *[filename]* *options*

sim6x	invokes the debugger for the fixed-point simulator.
emu6x	invokes the debugger for the emulator.
sim67x	invokes the debugger for the floating-point simulator.
sim6xfast	invokes the fast version of the fixed-point simulator.
<i>filename</i>	is an optional parameter that names an object file that the debugger loads into memory during invocation. The debugger looks for the file in the current directory; if the file is not in the current directory, you must supply the entire path-name. If you do not supply an extension for the filename, the debugger assumes that the extension is .out.
<i>options</i>	supply the debugger with information on how to handle files, manage the display, and input information.

Invoking multiple debuggers (emulator only)

Before you can invoke multiple debuggers in a multiprocessing environment, you must first invoke the parallel debug manager (PDM). The PDM is invoked and PDM commands are executed from a command shell window within the host windowing system. The format for invoking the PDM is:

```
pdm [-t filename]
```

Once the PDM is invoked, you will see the PDM command prompt (PDM:1>>) and can begin entering commands.

When you invoke the PDM, it searches for a file called `init.pdm`. This file contains initialization commands for the PDM. The PDM searches for the `init.pdm` file in the current directory and in the directories you specify with the `D_DIR` environment variable. If the PDM cannot find the initialization file, you will see this message:

```
Cannot open take file.
```

Note:

The PDM environment uses the interprocess communication (IPC) features of UNIX (shared memory, message queues, and semaphores) to provide and manage communications between the different tasks. If you are not sure whether the IPC features are enabled, see your system administrator. To use the PDM environment, you should be familiar with the IPC status (`ipcs`) and IPC remove (`ipcrm`) UNIX commands. If you use the UNIX task kill (`kill`) command to terminate execution of tasks, you will also need to use the `ipcrm` command to terminate the shared memory, message queues, and semaphores used by the PDM.

When you debug a multiprocessing application, each processor must have its own debugger. These debuggers can be invoked individually from the PDM command line.

To invoke a debugger, use the SPAWN command. The syntax for SPAWN is:


```
spawn emu6x -n processor_name [filename] [options]
```

- ❑ **emu6x** is the executable that invokes the debugger.

To invoke a debugger, the PDM must be able to find the executable file for that debugger. The PDM first searches the current directory and then searches the directories listed with the `PATH` statement or path environment variable.

- ❑ **-n** *processor name* supplies a processor name. You *must* use the **-n** option because the PDM uses processor names to identify the various debuggers that are running.

The processor name must match one of the names defined in your board configuration file (see Appendix B, *Describing Your Target System to the Debugger*). For example, to invoke a debugger for a 'C6x that you had defined as CPU_A, you would enter:

```
spawn emu6x -n CPU_A 
```

The processor name can consist of up to eight alphanumeric characters or underscore characters and must begin with an alphanumeric character. Note that the name is not case sensitive.

- ❑ *filename* is an optional parameter that names an object file that the debugger loads into memory during invocation. The debugger looks for the file in the current directory; if the file is not in the current directory, you must supply the entire pathname.

If you do not supply an extension for the filename, the debugger assumes that the extension is .out, unless you are using multiple extensions. You must specify the *entire* filename if the filename has more than one extension.

- ❑ **-options** supply the debugger with additional information. See section 2.7, page 2-10, for a summary table of debugger options.

2.7 Summary of Debugger Options

Table 2–1 summarizes the debugger options that you can use when invoking a debugger (see section 2.6 on page 2-7 for information on how to invoke the debugger with debugger options for your particular operating system). The rest of this section describes these options in more detail. You can also specify file-name and option information with the `D_OPTIONS` environment variable by following the instructions in section 2.4 on page 2-5.

Table 2–1. Summary of Debugger Options

Option	Brief Description	Debugger Tools
<code>-c</code>	Clear the .bss section	All
<code>-d machine name</code>	Display the debugger on different machine	All (X Window System™ only)
<code>-f filename</code>	Identify a new board configuration file	Emulator
<code>-i pathname</code>	Identify additional directories	All
<code>-me</code>	Select big-endian format	All
<code>-n device_name</code>	Identify device for debugging	Emulator
<code>-p port_address</code>	Identify the port address	Emulator
<code>-profile</code>	Enter the profiling environment	All
<code>-s filename</code>	Load the symbol table only	All
<code>-t filename</code>	Identify a new initialization file	All
<code>-v</code>	Load without the symbol table	All
<code>-x</code>	Ignore <code>D_OPTIONS</code>	All

Clearing the .bss section (`-c` option)

The `-c` option clears the .bss section when the debugger loads code. Use this option when you have C programs that use the RAM initialization model (specified with the `-cr` linker option described in the *TMS320C6x Assembly Language Tools User's Guide*).

Displaying the debugger on a different machine (`-d` option)

If you are using the X Window System, you can use the `-d` option to display the debugger on a different machine than the one the program is running on. The format for this option is:

`-d machine_name:0`

The **:0** must follow the name of the machine on which you want to view the debugger.

You can also specify a different machine by using the **DISPLAY** environment variable (see your getting started guide for more information). If you use both the **DISPLAY** environment variable and the **-d** option, **-d** overrides **DISPLAY**.

Identifying a new configuration file (-f option)

If you are using the emulator, the **-f** option allows you to specify a board configuration file to be used instead of **board.dat**. The format for this option is:

-f filename.dat

See Appendix B, *Describing Your Target System to the Debugger*, for information about creating a board configuration file.

Identifying additional directories (-i option)

The **-i** option identifies additional directories that contain your source files. You can specify as many pathnames as necessary; use the **-i** option with each pathname in this format:

-i pathname₁ -i pathname₂ -i pathname₃...

Using **-i** is similar to using the **D_SRC** environment variable (see the information about setting up the **D_SRC** environment variable in section 2.3 on page 2-4). If you name directories with both **-i** and **D_SRC**, the debugger first searches through directories named with **-i**. The debugger can track a cumulative total of 20 paths (including paths specified with **-i**, **D_SRC**, and the debugger **USE** command).

Selecting big-endian format (-me option)

The **-me** option tells the debugger that the object file to be loaded is in big-endian format. The default is little-endian format.

Identifying the processor to be debugged (-n option)

The **-n** option is valid only when you are using the emulator. The **-n** option allows you to specify which particular 'C6x to debug when you are using the **spawn** command to invoke multiple debuggers. The processor name must match one of the names defined in your **board.cfg** file. The format for this option is:

-n device_name

Device names can be any string less than 32 characters long; however, they cannot contain double quotes, a line feed, or a newline character. For more information about the board.cfg file, see Appendix B, *Describing Your Target System to the Debugger*.

Identifying the port address (–p option)

The –p option specifies which port the debugger uses to communicate with the emulator. The –p option is valid only when you are using the emulator. The format for the –p option is:

–p *port_address*

The –p option identifies the I/O port address that the debugger uses for communicating with the emulator. If you used the default switch settings, you do not need to use the –p option. **If you use nondefault switch settings, you must use –p.** For information on switch settings, see the *XDS51x Installation Guide*; determine your switch settings, and replace *port address* with one of these values:

If your Switch 1 is...	and your Switch 2 is...	Use this –p option...
On (default)	On (default)	240 (optional)
On	Off	280
Off	On	320
Off	Off	340

If you did not note your I/O switch settings, you can use a trial-and-error approach to find the correct –p setting. If you use the wrong setting, you will see an error message when you invoke the debugger. (See the *XDS51x Installation Guide* for more information.)

If you are using a UNIX workstation, the –p option specifies the SCSI port the debugger uses for communicating with the emulator. For more information, see the *XDS51x Installation Guide*.

Entering the profiling environment (–profile option)

This option is valid only when you are using the simulator. The –profile option allows you to bring up the debugger in a profiling environment so that you can collect statistics about code execution. Only a subset of the basic debugger features is available in the profiling environment. For more information about the profiling environment, see Chapter 8.

You can also enter the profiling environment after invoking the debugger by using the debugger’s Profile→Profile Mode menu option or PROFILE command within the debugger environment.

Loading the symbol table only (-s option)

The `-s` option allows you to load only a file's symbol table (without the file's object code). This option is most useful in an emulation environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). In such an environment, loading the symbol table allows you to perform symbolic debugging and examine the values of C variables. The format for this option is:

`-s filename.out`

Using this option is similar to loading a file by using the debugger's File→Load Symbols menu option or the SLOAD command within the debugger environment.

Identifying a new initialization file (-t option)

The `-t` option allows you to specify your own customized initialization command file to use instead of `siminit.cmd`, `emuinit.cmd`, or `init.cmd`. The format for the `-t` option is:

`-t filename.cmd`

Using this option is similar to loading a batch file by using the debugger's File→Execute Take File... menu option or the TAKE command within the debugger environment.

Loading without the symbol table (-v option)

The `-v` option prevents the debugger from loading the entire symbol table when you load an object file. The debugger loads only the global symbols and later loads local symbols as it needs them. This speeds up the loading time and consumes less memory.

The `-v` option affects all loads, including those performed when you invoke the debugger and those performed with the File→Load Program menu option or the LOAD command within the debugger environment.

Ignoring D_OPTIONS (-x option)

The `-x` option tells the debugger to ignore any information supplied with the `D_OPTIONS` environment variable (described in section 2.4 on page 2-5).

2.8 Debugging Modes

The debugger has three debugging modes: auto, assembly, and mixed. Each mode changes the debugger display by adding or hiding specific windows. This section shows the default displays and the windows that the debugger automatically displays for these modes. These modes cannot be used within the profiling environment; the Command, Profile, Disassembly, and File windows are the only available windows in the profiling environment.

Auto mode

In *auto mode*, the debugger automatically displays whichever type of code is currently running: assembly language or C. This is the default mode. Auto mode has two types of displays:

- ☐ When the debugger is running assembly language code, you see an assembly display similar to the one in Figure 2–1. The Disassembly window displays the reverse assembly of memory contents.

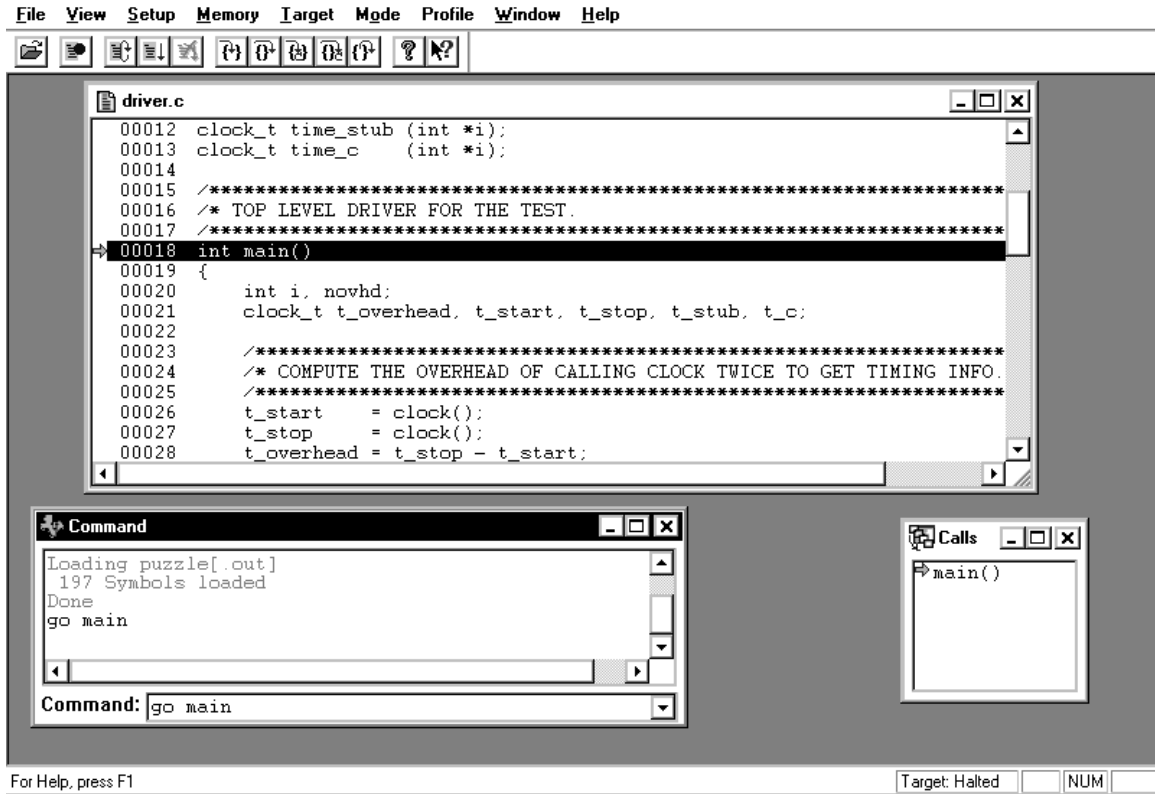
When you first invoke the debugger, you see a display similar to this.

- ☐ When the debugger is running C code or serial assembly compiled with the `-g` option, you see a C display similar to the one in Figure 2–2. (This assumes that the debugger can find your C source file to display in the File window. If the debugger cannot find your source, it displays the disassembly code only.)

When you are running assembly language code, the debugger automatically displays a Memory window, the Disassembly window, the CPU register window, and the Command window. In addition to these windows, you can open Watch windows and additional Memory windows.

When you are running C code, the debugger automatically displays the Command, Calls, and File windows. In addition to these windows, you can open Watch windows.

Figure 2–2. Typical C Display (for Auto Mode Only)



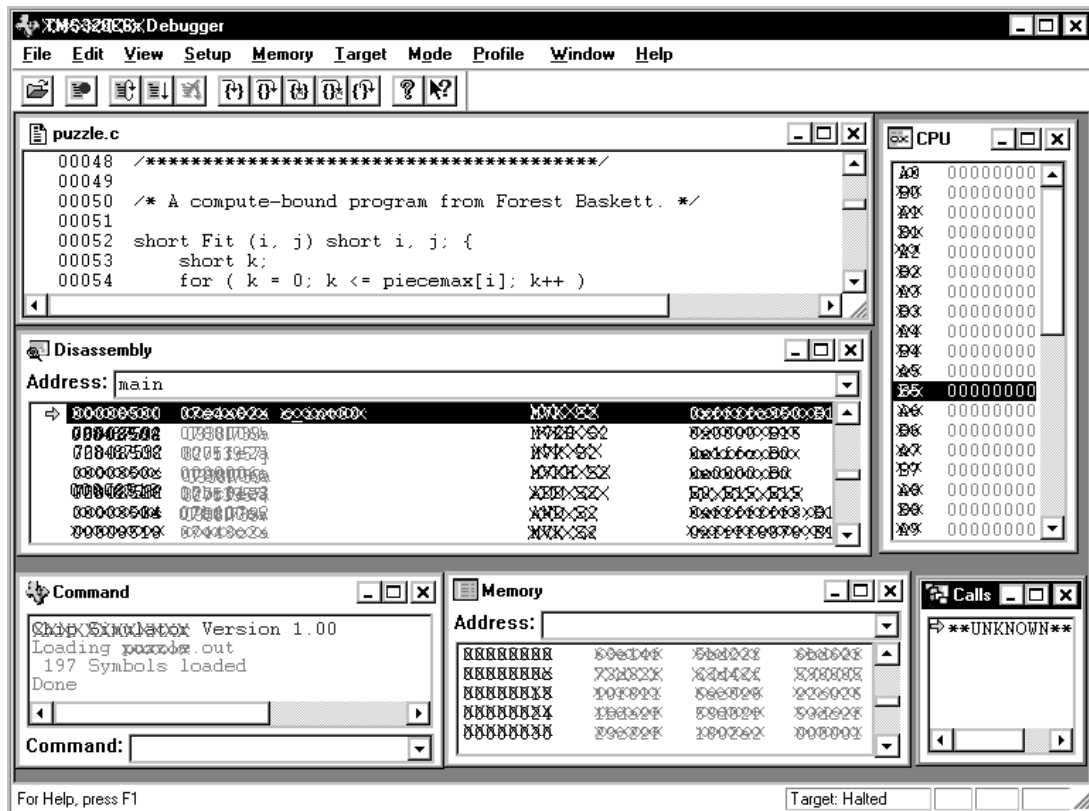
Mixed mode

Mixed mode is for viewing assembly language and C code at the same time. Figure 2–3 shows the default display for mixed mode.

In mixed mode, the debugger displays all windows that can be displayed in auto and assembly modes, regardless of whether you are currently running assembly language or C code. This is useful for finding bugs in C programs that exploit specific architectural features of the target device.

If you assemble your code with the `-g` assembler option, the debugger displays in the File window the contents of the assembly source file, in addition to displaying the reverse assembly of memory contents in the Disassembly window.

Figure 2–3. Typical Mixed Display (for Mixed Mode Only)



Restrictions associated with debugging modes


The assembly language code that the debugger shows you in the Disassembly window is the disassembly (reverse assembly) of the memory contents. If you load object code into memory, the assembly language code in the Disassembly window is the disassembly of that object code. If you do not load an object file, the disassembly will not be very useful.

Some commands are valid only in certain modes, especially if a command applies to a window that is visible only in certain modes. In this case, entering the command causes the debugger to switch to the mode that is appropriate for the command. The following commands are valid only in the modes listed:

- ☐ The CALLS, DISP, FUNC, and FILE commands are valid only in auto and mixed modes.
- ☐ The MEM command is valid only in assembly and mixed modes.

2.9 Exiting the Debugger or the PDM

To exit the debugger, use one of these methods:

- ☐ From File menu at the top of the debugger display, select Exit.
- ☐ Close the application window for the debugger.
- ☐ From the command line, enter:
`quit` 

You can also enter QUIT from the command line of the PDM to quit all of the debuggers (and also close the PDM).

Entering and Using Commands

The debugger provides you with several methods for entering commands:

- ☐ From the toolbar
- ☐ From the menu bar
- ☐ With function keys
- ☐ From the command line
- ☐ From a batch file

This chapter describes how you can create aliases for commands and command sequences that you enter frequently, as well as information about using a batch file or a log file for entering commands.

Topic	Page
3.1 Defining Your Own Command Strings	3-2
3.2 Entering Operating-System Commands From Within the Debugger	3-5
3.3 Creating and Executing a Batch File	3-7
3.4 Creating a Log File to Reexecute a Series of Commands	3-12

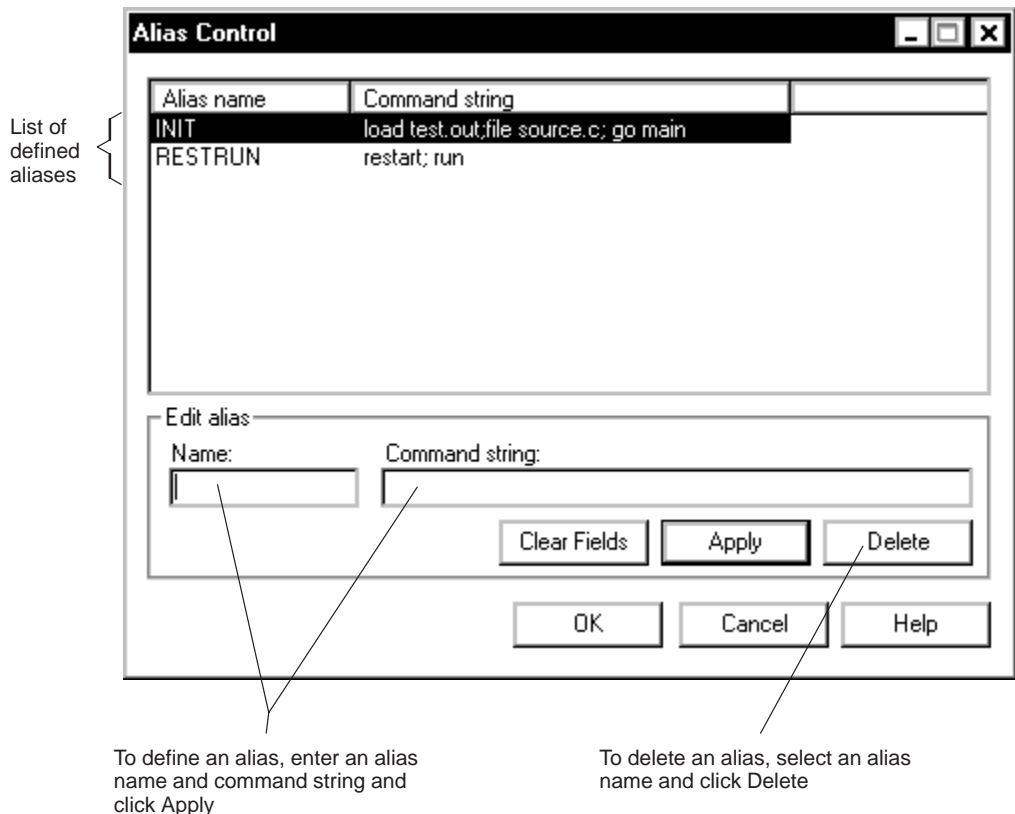
3.1 Defining Your Own Command Strings

The debugger provides a shorthand method of entering often-used commands or command sequences. This process is called *aliasing*. Aliasing allows you to define an alias name for the command(s) and then enter the alias name as if it were a debugger command.

Note:

Creating aliased commands in PDM is different from creating aliased commands in the debugger. For information about the PDM versions of the ALIAS and UNALIAS commands, see page 11-15.

To use the aliasing feature, select Alias Commands from the Setup menu. This displays the Alias Control dialog box:



Defining an alias

To define an alias, follow these steps:

- 1) From the Setup menu, select Alias Commands. This displays the Alias Control dialog box.
- 2) In the Name field, enter a name for the alias.
- 3) In the Command string field, enter the command string that you want to associate with the alias name. If you want to associate multiple commands with the alias, separate the commands with a semicolon.

Edit alias
 Name: Enter a name for the alias
 Command string: Enter the command string that you want to associate with the alias name

- 4) Click Apply.
- 5) Click OK to close the Alias Control dialog box.

You can include a defined alias name in the command string of another alias definition.

Defining an alias with parameters

The command string that you use to define an alias can include parameter variables for which you supply the values when you use the alias. Use a percent sign and a number (%1) to represent each parameter. Use consecutive numbers (%1, %2, %3), unless you plan to reuse the same parameter value for multiple commands.

For example, suppose that every time you filled an area of memory, you also wanted to display that block in the Memory window. You could set up the following alias:

Edit alias
 Name:
 Command string:

Once you define this alias, you could enter the following from the command line:

```
mf1l 0x808020,0x18,0x1122
```

In this example, the first value (0x808020) is substituted for the first FILL parameter and the MEM parameter (%1). The second and third values are substituted for the second and third FILL parameters (%2 and %3).

Editing or redefining an alias

To edit or redefine an alias, follow these steps:

- 1) From the Setup menu, select Alias Commands. This displays the Alias Control dialog box.
- 2) From the list of aliases at the top of the dialog box, select the alias that you want to edit or redefine.
- 3) In the Name and Command string fields, make the appropriate changes.
- 4) Click Apply.
- 5) Click OK to close the Alias Control dialog box.

Deleting an alias

To delete an alias, follow these steps:

- 1) From the Setup menu, select Alias Commands. This displays the Alias Control dialog box.
- 2) From the list of aliases at the top of the dialog box, select the alias that you want to delete.
- 3) Click Delete.
- 4) Click OK to close the Alias Control dialog box.

Considerations for using alias definitions

Alias definitions are lost when you exit the debugger. If you want to reuse aliases, define them in a batch file. Use the ALIAS command, as described on page 12-14.

Individual commands within a command string are limited to an expanded length of 132 characters. The expanded length of the command includes the length of any substituted parameter values.

3.2 Entering Operating-System Commands From Within the Debugger

The debugger provides a simple method for entering operating-system commands without explicitly exiting the debugger environment. To do this, use the `SYSTEM` command. The format for this command is:

system [*operating-system command* [, *flag*]]

The `SYSTEM` command behaves in one of two ways, depending on whether or not you supply an operating-system command as a parameter:

- ☐ If you enter the `SYSTEM` command with an operating-system command as a parameter, then you stay within the debugger environment.
- ☐ If you enter the `SYSTEM` command without parameters, the debugger opens a *system shell*. This means that the debugger blanks the debugger display and temporarily exits to the operating-system prompt.

Use the first method when you have only one command to enter; use the second method when you have several commands to enter.

Entering a single command from the debugger command line

If you need to enter only a single operating-system command, supply it as a parameter to the `SYSTEM` command. For example, if you want to copy a file from another directory into the current directory, enter:

system copy a:\backup\sample.c sample.c 

If the operating-system command produces a display (such as a message), the debugger blanks the upper portion of the debugger display to show the information. In this situation, you can use the *flag* parameter to tell the debugger whether or not it should hesitate after displaying the results of the operating-system command. The *flag* parameter can be 0 or 1:

- 0** The debugger immediately returns to the debugger environment after the last item of information is displayed.
- 1** The debugger does not return to the debugger environment until you enter:

exit .

(This is the default.)

In the preceding example, the debugger would open a system shell to display the following message:

```
1 File(s) copied
```

The message would be displayed until you entered **exit** at the command prompt in the system shell.

If you wanted the debugger to display the message and then return immediately to the debugger environment, you could enter the command in this way:

```
system copy a:\backup\sample.c sample.c,0 
```

Entering several commands from a system shell

If you need to enter several commands, enter the **SYSTEM** command without parameters. The debugger opens a system shell and displays the operating-system prompt. You can enter any number of operating-system commands, one at a time, following each with a return.

When you are finished entering commands and are ready to return to the debugger environment, enter:

```
exit 
```

For information about the PDM version of the **SYSTEM** command, see page 11-16.

3.3 Creating and Executing a Batch File

You can create a batch file for several commands that you want to enter at one time. A batch file is useful for tasks such as defining aliases that you want to reuse, defining your memory map, setting up your screen configuration, loading object code, or any other task that you want to do each time you invoke the debugger.

You can create the batch file in any text editor. For each debugger command that you include in the batch file, use the same syntax that you would use if you were entering the command from the debugger's command line. Example 3–1 shows a sample batch file that you can create.

You can set up a batch file to call another batch file; they can be nested in this manner up to ten deep.

Example 3–1. Sample Batch File for Use With the Debugger

```
echo Loading object code
load testcode.out

echo Loading screen configuration
sconfig myconfig.clr

echo Defining aliases
alias restrun, "restart; run"
alias wavars, "wa pc; wa i; wa j"
```

Echoing strings in a batch file

When executing a batch file, you can display a string to the Command window by including the ECHO command in your batch file. The syntax for the command is:

echo *string*

This displays the *string* in the display area of the Command window.

For example, you might want to document what is happening during the execution of a certain batch file. To do this, you could use a line such as the following one in your batch file to indicate that you are creating a new memory map for your device:

echo Creating new memory map

(Notice that the string is not enclosed in quotes.)

When you execute the batch file, the following message appears:

```
.  
.   
.   
Creating new memory map  
.   
.   
. 
```

Any leading blanks in your string are removed when the ECHO command is executed.

For more information about the PDM version of the ECHO command, see page 11-12.

Executing commands conditionally in a batch file

To execute debugger commands conditionally in a batch file, use the IF/ELSE/ENDIF commands. The syntax is:

```
if Boolean expression  
  debugger commands  
[else  
  debugger commands]  
endif
```

If the Boolean expression evaluates to true (1), the debugger executes all commands between the IF and ELSE or ENDIF. The ELSE portion of the command is optional. (See Chapter 13, *Basic Information About C Expressions*, for more information.)

The debugger includes some predefined constants for use with IF. These constants evaluate to 0 (false) or 1 (true). Table 3–1 shows the constants and their corresponding tools.

Table 3–1. *Predefined Constants for Use With Conditional Commands*

Constant	Debugger Tool
\$\$EMU\$\$	Emulator
\$\$SIM\$\$	Simulator

One way you can use these predefined constants is to create an initialization batch file that works for any debugger tool. This is useful if you are using, for example, both the emulator and the simulator. To do this, you can set up a batch file such as the following (make the appropriate modifications for UNIX).

```

if $$EMU$$
echo Invoking initialization batch file for emulator.
use .\emu6x
take emuinit.cmd
.
.
.
endif

if $$SIM$$
echo Invoking initialization batch file for simulator.
use .\sim6x
take siminit.cmd
.
.
.
endif
.
.
.

```

In this example, the debugger executes only the initialization commands that apply to the debugger tool that you invoke.

The IF/ELSE/ENDIF command works with the following conditions:

- ☐ You can use conditional and looping commands only in a batch file.
- ☐ You must enter each debugger command on a separate line in the batch file.
- ☐ You cannot nest conditional and looping commands within the same batch file.

See *Controlling PDM command execution*, page 11-10, for more information about the PDM versions of the IF and LOOP commands.

Looping command execution in a batch file

To set up a looping situation to execute debugger commands in a batch file, use the LOOP/ENDLOOP commands. The syntax is:

```

loop expression
      debugger commands
endloop

```

These looping commands evaluate using the same method as the conditional RUN command expression. (See Chapter 13, *Basic Information About C Expressions*, for more information.)

If you use an *expression* that is not Boolean, the debugger evaluates the expression as a loop count. For example, if you wanted to execute a sequence of debugger commands ten times, you would use the following code sequence:

```
loop 10
step
.
.
.
endloop
```

The debugger treats the 10 as a counter and executes the debugger commands ten times.

If you use a Boolean *expression*, the debugger executes the commands repeatedly as long as the expression is true. This type of expression uses one of the following operators as the highest precedence operator in the expression:

>	> =	<
< =	= =	! =
&&		!

For example, if you want to trace some register values continuously, you can set up a looping expression like this one:

```
loop !0
step
? PC
? A0
endloop
```

The LOOP/ENDLOOP command works with the following conditions:

- ☐ You can use conditional and looping commands only in a batch file.
- ☐ You must enter each debugger command on a separate line in the batch file.
- ☐ You cannot nest conditional and looping commands within the same batch file.

See *Controlling PDM command execution*, page 11-10, for more information about the PDM versions of the IF and LOOP commands.

Pausing the execution of a batch file

You can pause the debugger or PDM while running a batch file or executing a flow control command. Pausing is especially helpful in debugging the commands in a batch file. To do so, include the PAUSE command in the batch file:

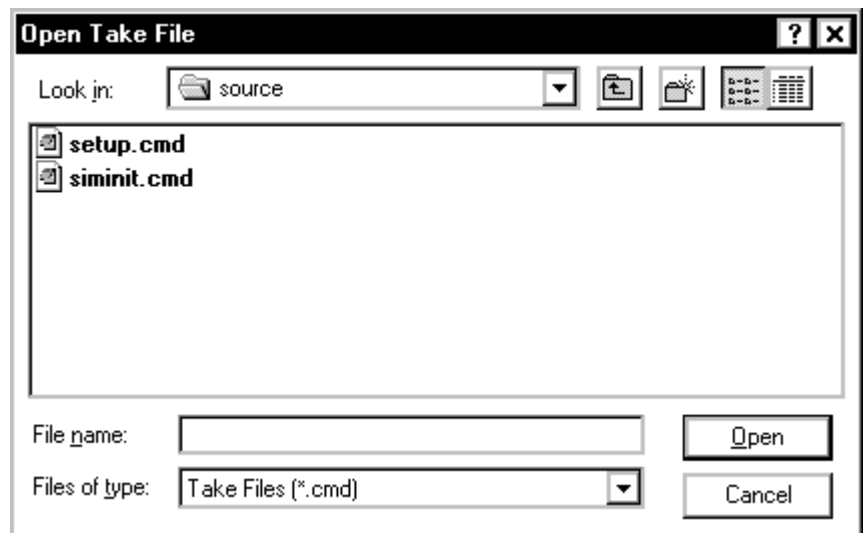
pause

When the debugger or PDM reads this command in a batch file or during a flow control command segment, the debugger/PDM stops execution and displays a dialog box. To continue processing, click OK or press **ENTER**.

Executing a batch file

Once you create a batch file, you can tell the debugger to read and execute or *take* its commands from the batch file (also known as a *take* file). To do so, follow these steps:

- 1) From the File menu, select Execute Take File. This displays the Open Take File dialog box:



- 2) Select the file that you want to execute. To do so, you might need to change the working directory.
- 3) Click Open.

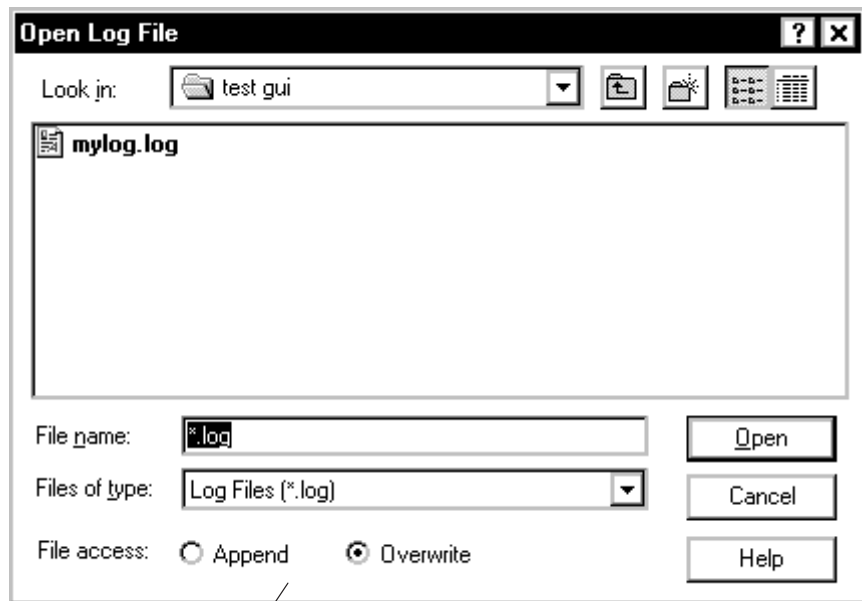
This causes the debugger to read and execute the commands in the batch file. To halt the debugger's execution of a batch file, press **ESC**.

3.4 Creating a Log File to Reexecute a Series of Commands

The information shown in the display area of the Command window can be written to a log file. The log file is a system file that contains commands you have entered from the command line, from the toolbar, from the menus, or with function keys. The log file also contains the results from commands and error or progress messages. The debugger automatically precedes all error or progress messages and command results with a semicolon to turn them into comments. This way, you can reexecute the commands in your log file by using the File→Execute Take File menu option. For information about creating a log file in PDM, see page 11-10. You can view the log file with any text editor.

To begin a recording session, follow these steps:

- 1) From the File menu, select Open Log File. This displays the Open Log File dialog box:



Select whether to append or overwrite an existing log file.

- 2) Select the directory where you want the file to be saved.

- 3) In the File name field, enter a name for the log file. Use a .log extension to identify the file as a log file.
- 4) Click Open.
- 5) If the file that you want to use already exists, select that file, then select one of the following actions in the File access field:
 - ☐ Append to add the log information to an existing file
 - ☐ Overwrite to write over the contents of an existing file
- 6) Click Open.

The debugger records all commands that you enter from the command line, from the toolbar, from the menus, or with function keys.

To end the recording session, from the File menu, select Close Log File.

Defining a Memory Map

Before you begin a debugging session, you must supply the debugger with a memory map. The memory map tells the debugger which areas of memory it can and cannot access.

Topic	Page
4.1 The Memory Map: What It Is and Why You Must Define It	4-2
4.2 Creating or Modifying the Memory Map	4-3
4.3 Enabling Memory Mapping	4-7
4.4 A Sample Memory Map	4-9
4.5 Defining and Executing a Memory Map in a Batch File	4-10
4.6 Returning to the Original Memory Map	4-12
4.7 Using Multiple Memory Maps for Multiple Target Systems	4-13
4.8 Simulating I/O Space (Simulator Only)	4-14
4.9 Simulating External Interrupts (Simulator Only)	4-16

4.1 The Memory Map: What It Is and Why You Must Define It

A memory map tells the debugger which areas of memory it can and cannot access. Memory maps vary, depending on the application. Typically, the map matches the MEMORY definition in your linker command file.

Note:

When the debugger compares memory accesses against the memory map, it performs this checking in software, not hardware. The debugger cannot prevent your program from attempting to access nonexistent memory.

A special default initialization batch file included with the debugger package defines a memory map for your version of the debugger. This memory map may be sufficient when you first begin using the debugger. However, the debugger enables you to modify the default memory map or define a new memory map interactively (as described in section 4.2 on page 4-3) or by defining the memory map in a batch file (see section 4.5 on page 4-10).

Potential memory map problems

You may experience these problems if the memory map is not correctly defined and enabled:

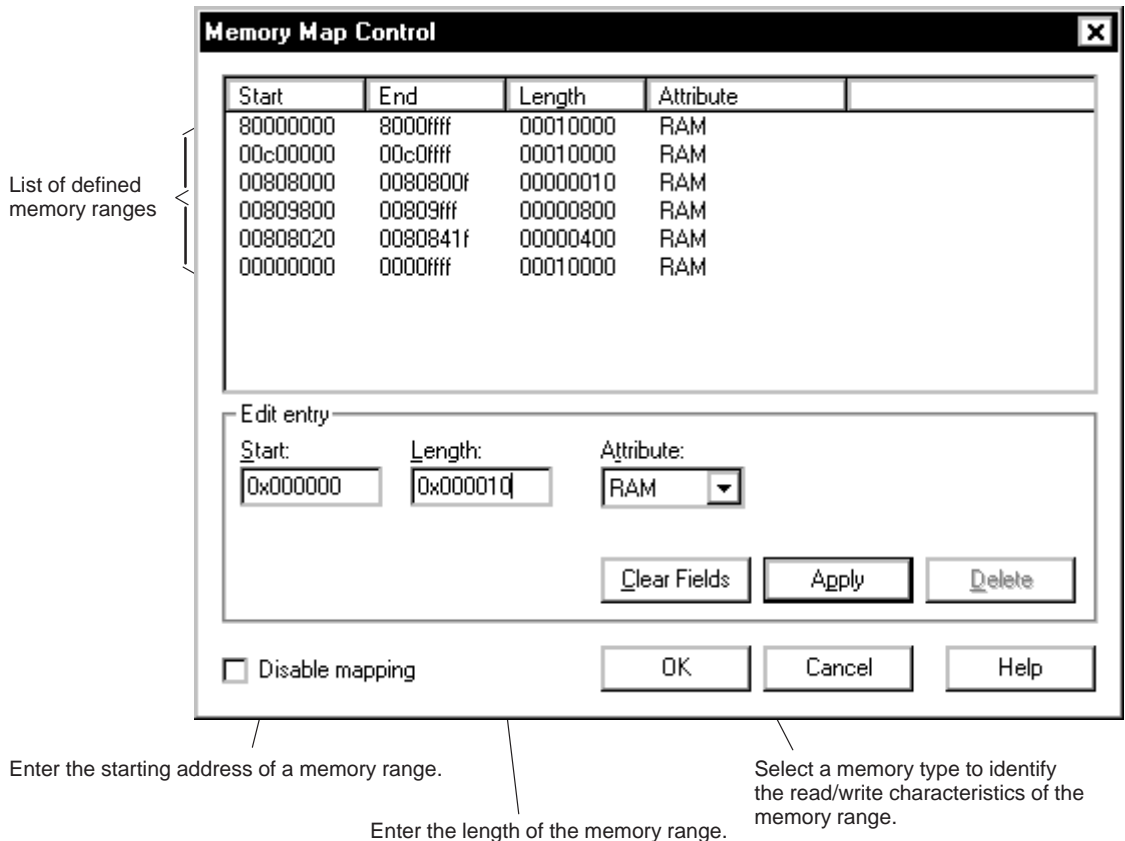
- ☐ **Accessing invalid memory addresses.** If you do not supply a batch file containing memory-map commands, the debugger is initially unable to access any target memory locations. Invalid memory addresses and their contents are displayed in red in the data-display windows by default.
- ☐ **Accessing an undefined or protected area.** When memory mapping is enabled, the debugger checks each of its memory accesses against the memory map. If you attempt to access an undefined or protected area, the debugger displays an error message.
- ☐ **Loading a COFF file with sections that cross a memory range.** Be sure that the map ranges you specify in a COFF file match those that you defined in a batch file or with the Memory Map Control dialog box. Alternatively, you can turn memory mapping off during a load by disabling memory mapping (described in section 4.3 on page 4-7). When mapping is off, you can still access memory locations.

Note:

If the emulator accesses an illegal or reserved memory location, it posts an error in the EMIF (external memory interface) control register and returns 0 as the data.

4.2 Creating or Modifying the Memory Map

To identify valid ranges of target memory, select Mapping from the Memory menu. This displays the Memory Map Control dialog box:



Adding a range of memory

To add a range of memory, follow these steps:

- 1) From the Memory menu, select Mapping. This displays the Memory Map Control dialog box.
- 2) In the Start field, enter the starting address for a memory range. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.
- 3) In the Length field, enter the length of the memory range. The length can be any C expression.

- 4) In the Attribute field, select a memory type to identify the read/write characteristics of the memory range.
- 5) Click Apply.
- 6) Click OK.

The following restrictions apply to identifying usable memory ranges:

- ☐ A new memory range cannot overlap an existing entry. If you define a range that overlaps an existing range, the debugger ignores the new range.
- ☐ Be sure that the map ranges that you specify in a COFF file match those that you define with the Memory Map Control dialog box.
- ☐ The origin and length values for a range that you define with the MEMORY directive in your linker command file must match the Start and Length values for the same range in the Memory Map Control dialog box.
- ☐ The debugger caches memory that is not defined as a port type (INPORT, OUTPORT, or IOPORT). For ranges that you do not want cached, be sure to map them as ports.

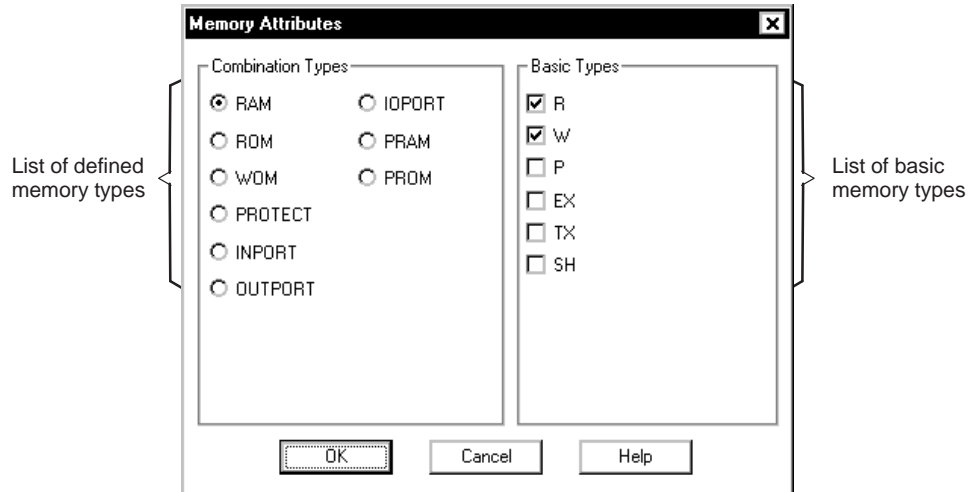
Creating a customized memory type

The Attribute drop list in the Memory Map Control dialog box allows you to select from several predefined memory types such as RAM or ROM. If the predefined memory types do not apply to your memory range, you can create a customized memory type.

To create a customized memory type:

- 1) From the Memory menu, select Mapping. This displays the Memory Map Control dialog box.
- 2) In the Start field, enter the starting address for the memory range you want to customize. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.
- 3) In the Length field, enter the length of the memory range. The length can be any C expression.

- 4) From the Attribute drop list, select Custom.... The Memory Attributes dialog box appears.



- 5) From the Basic Types column, select the individual memory attributes that you want to apply to the memory range that you are adding.

Mnemonic	Basic Memory Type
R	Readable
W	Writable
P	I/O port
EX	External
TX	Text
SH	Shared

- 6) Click OK. This closes the Memory Attributes dialog box and applies the customized memory attributes to the memory range in the Memory Map Control dialog box.
- 7) Add other memory ranges as needed, then click OK to close the Memory Map Control dialog box.

Deleting a range of memory

To delete a range of memory, follow these steps:

- 1) Select Mapping from the Memory menu. This displays the Memory Map Control dialog box.
- 2) From the list of defined ranges at the top of the dialog box, select the range that you want to delete.
- 3) Click Delete.
- 4) Click OK.

Before you can delete a memory address used as a simulated I/O port from the memory map, you must disconnect the address. See the *Disconnecting an I/O port* section on page 4-15 for information.

Modifying a defined range of memory

To modify a defined range of memory, follow these steps:

- 1) Select Mapping from the Memory menu. This displays the Memory Map Control dialog box.
- 2) From the list of defined ranges at the top of the dialog box, select the range that you want to modify.
- 3) In the Memory Type, Start, Length, Attribute and/or Wait states fields, make the appropriate changes.
- 4) Click Apply.
- 5) Click OK.

4.3 Enabling Memory Mapping

By default, mapping is enabled when you invoke the debugger. In some instances, you may want to explicitly enable or disable memory. To do so, open the Memory Map Control dialog box. From the Memory menu, select Mapping. In the lower left corner of the dialog box, there is an option for disabling memory mapping:

Start	End	Length	Attribute
80000000	8000ffff	00010000	RAM
00c00000	00c0ffff	00010000	RAM
00808000	008080ff	00000010	RAM
00809800	00809fff	00000800	RAM
00808020	0080841f	00000400	RAM
00000000	0000ffff	00010000	RAM

Edit entry

Start: Length: Attribute:

☐ Disable mapping

Click here to enable/disable memory mapping

☐ Memory mapping is enabled when the box is empty:

☐ Disable mapping

☐ Memory mapping is disabled when the box is checked:

☒ Disable mapping

Disabling memory mapping can cause bus fault problems in the target because the debugger may attempt to access nonexistent memory.

When you disable memory mapping with the simulator, you can still access memory locations. However, the debugger does not prevent you from accessing memory locations that you have not defined as valid in the memory map.

When you disable memory mapping with the emulator, only memory linked to the text section is downloaded over the program bus.

Note:

When memory mapping is enabled, you cannot:

- ☐ Access memory locations that are not listed in the Memory Control dialog box
- ☐ Modify the contents of memory areas that are defined as read only or protected

If you attempt to access memory in these situations, the debugger displays this message in the display area of the Command window:

Error in expression

4.4 A Sample Memory Map

Because you must define a memory map before you can run any programs, it is convenient to define the memory map in the initialization batch files. Figure 4–1 (a) shows the memory map that is defined in the initialization batch file that accompanies the 'C6x simulator. You can use the file as is, edit it, or create your own memory map batch file to match your own configuration. You can also define the memory map after you have invoked the debugger with the Memory Map Control dialog box (see section 4.2 on page 4-3).

If you are defining the memory map in a batch file, you can use MA (map add) commands to define valid memory ranges and identify the read/write characteristics of the memory ranges. (For more information about the MA command, see section 4.5 on page 4-10.) By default, mapping is enabled when you invoke the debugger. Figure 4–1 (b) illustrates the memory map defined by the MA commands in Figure 4–1 (a).

Figure 4–1. Sample Memory Map for Use With a TMS320C6x Simulator

(a) Memory map commands

```
ma 0x00000000, 0x01000000, RAM
ma 0x01000000, 0x00400000, RAM
ma 0x01400000, 0x00010000, RAM
ma 0x01800000, 0x00400000, RAM
ma 0x02000000, 0x01000000, RAM
ma 0x03000000, 0x01000000, RAM
ma 0x80000000, 0x00010000, RAM
```

(b) Memory map for TMS320C6x local memory

0x0000_0000 to 0x00ff_ffff	CE0 External Memory
0x0100_0000 to 0x013f_ffff	CE1 External Memory
0x0140_0000 to 0x0140_ffff	Internal Program Memory
0x0180_0000 to 0x016f_ffff	Internal Peripheral Space
0x0200_0000 to 0x02ff_ffff	CE2 External Memory
0x0300_0000 to 0x03ff_ffff	CE3 External Memory
0x8000_0000 to 0x8000_ffff	Internal Data Memory

4.5 Defining and Executing a Memory Map in a Batch File

You can create a batch file that contains memory map commands. This provides you with a convenient way to define a memory for each debugging session. You can define the memory map in the initialization batch file, which executes when you invoke the debugger, or you can define the memory map in a separate batch file of your own that you can execute using the File→Execute Take File menu option or the `-t` debugger option.

Defining a memory map in a batch file

To define a memory map in a batch file, use the `MA` command. The syntax for the `MA` command is:

ma *address, length, type*

- ☐ The *address* parameter defines the starting address of a range. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. If you want to specify a hex address, be sure to prefix the address number with `0x`; otherwise, the debugger treats the number as a decimal address.
- ☐ The *length* parameter defines the length of the range. This parameter can be any C expression.
- ☐ The *type* parameter identifies the read/write characteristics of the memory range. The *type* must be one of these keywords:

To identify this kind of memory . . .	Use this keyword as the <i>type</i> parameter . . .
Read-only memory	R or ROM
Write-only memory	W or WOM
Read/write memory	R W or RAM
Read-only program memory	PROM
Read/write program memory	PRAM
No-access memory	PROTECT
Input port	INPORT or P R
Output port	OUTPORT or P W
Input/output port	IOPORT or P R W

The memory ranges that you define have the same restrictions as those defined for the Memory→Mapping menu option described in section 4.2 on page 4-3.

Executing a memory map batch file

To execute the batch file, use one of these methods:

- ☐ Use the File→Execute Take File... menu option from within the debugger environment.
- ☐ Use the `-t` debugger option to specify the batch file when you invoke the debugger. For more information, see page 2-13.
- ☐ Use the TAKE command. For more information, see section 3.3, *Creating and Executing a Batch File*, on page 3-7.

When you invoke the debugger, it follows these steps to find the batch file that defines your memory map:


- 1) It checks to see whether you have used the `-t` debugger option. The `-t` option allows you to specify a batch file other than the initialization batch file shipped with the debugger. If it finds the `-t` option, the debugger reads and executes the specified file.
- 2) If you do not use the `-t` option, the debugger looks for the default initialization batch file. The batch filename for the simulator is called `siminit.cmd`. The batch filename for the emulator is called `emuinit.cmd`. If the debugger finds the proper initialization batch file, it reads and executes the file.
- 3) If the debugger does not find the `-t` option or the initialization batch file, it looks for a file called `init.cmd`.

This search mechanism allows you to have a single initialization batch file that works for more than one debugger tool. To set up this file, you can use the IF/ELSE/ENDIF commands (for more information, see *Executing commands conditionally in a batch file* on page 3-8) to indicate which memory map applies to each tool. If the debugger finds the file, it reads and executes the file.

4.6 Returning to the Original Memory Map

If you modify the memory map during a debugging session, you may want to go back to the original memory map without quitting and reinvoking the debugger. You can do this by resetting the memory map and then using the File→Execute Take File menu option to read in your original memory map from a batch file.

Suppose, for example, that you set up your memory map in a batch file named *mem.map*. You can enter these commands to go back to this map:

- 1) From the command line enter, **mr**  to reset the memory map.
- 2) From the File menu, select Execute Take File.
- 3) From the Open Take File dialog box, select *mem.map* to reread the default memory map.

The MR command resets the memory map. (You could put the MR command in the batch file, preceding the commands that define the memory map.) The File→Execute Take File menu option tells the debugger to execute commands from the specified batch file.

4.7 Using Multiple Memory Maps for Multiple Target Systems

If you are debugging multiple applications, you may need a memory map for each target system. Here is the simplest method for handling this situation.

- 1) Let the initialization batch file define the memory map for one of your applications.
- 2) Create a separate batch file that defines the memory map for the additional target system. The filename is unimportant, but for the purposes of this example, assume that the file is named *filename.x*. The general format of this file's contents is:

mr	<i>Reset the memory map</i>
<i>MA commands</i>	<i>Define the new memory map</i>
map on	<i>Enable mapping</i>

This sequence of commands resets the memory map, defines a new memory map, and enables mapping. (Of course, you can include any other appropriate commands in this batch file.)

- 3) Invoke the debugger as usual.
- 4) The debugger reads the initialization batch file during invocation. Before you begin debugging, read in the commands from the new batch file using the File→Execute Take File menu option.

This redefines the memory map for the current debugging session.

You can also use the `-t` option when you invoke the debugger instead of the File→Execute Take File menu option. The `-t` option allows you to specify a new batch file to be used instead of the default initialization batch file.

4.8 Simulating I/O Space (Simulator Only)

Note:

In the fixed-point simulator (sim62x) only external memory can be used for simulating I/O space. Simulating I/O space is not supported by the fast version of the fixed-point simulator (sim62xfast) or the floating-point version of the simulator (sim67x).

In addition to adding memory ranges to the memory map, you can use the Memory→Mapping menu option to add I/O ports to the memory map. Then, by connecting to the port address, the debugger simulates external I/O cycle reads and writes by allowing you to read data in from a file and/or write data out to a file.

Connecting an I/O port

To connect a port to an input or output file, follow these steps:

- 1) On the command line, enter **mc**. This displays the Connect port to file dialog box:



- 2) In the Port Address field, enter the address where you want to simulate an I/O port. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.
- 3) In the Length field, enter the length of the memory range. The length can be any C expression.
- 4) If you are connecting a port to be read from a file, in the Filename field, enter the name of the file to which you want to connect. If you connect a port to read from a file, the file must exist, or the MC command will fail.
- 5) In the Read/Write field, enter how the file will be used (for input or output, respectively). The keywords for the read/write characteristics are available on page 4-10.

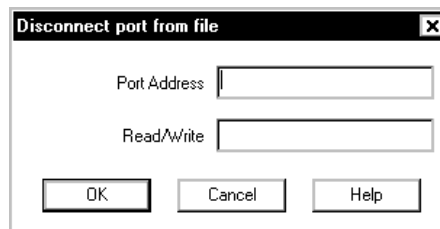
- 6) Click OK.

Any port in I/O space can be connected to a file. A maximum of one input and one output file can be connected to a single port; multiple ports can be connected to a single file. Memory-mapped ports can also be connected to files; any instruction that reads or writes to the memory-mapped port reads or writes to the associated file.

Disconnecting an I/O port

Before you can delete an I/O port from the memory map, you must use the MI command to disconnect the address. To disconnect a port from an input or output file, follow these steps:

- 1) In the command line, enter **mi**. This displays the Disconnecting port dialog box:



- 2) In the Port Address field, enter I/O port memory address that is to be closed.
- 3) In the Read/Write field, enter the characteristic used when the port was connected.
- 4) Click OK.

4.9 Simulating External Interrupts (Simulator Only)

Note:

Simulating external interrupts is not supported by the fast version of the fixed-point simulator or the floating-point version of the simulator.

The 'C6x allows you to simulate interrupts using the pin connect to file command, PINC. You can use any of the pins, NMI, INT4, INT5, INT6, and INT7.

Note:

The time interval is expressed as a function of CPU clock cycles. Simulation begins at the first clock cycle.

Setting up your input file

To simulate interrupts, you must first set up an input file that lists interrupt intervals. Your file must contain a clock cycle in the following format:

- ☐ The *clock cycle* parameter represents the CPU clock cycle where you want an interrupt to occur.

You can have two types of CPU clock cycles:

- **Absolute.** To use an absolute clock cycle, your cycle value must represent the actual CPU clock cycle where you want to simulate an interrupt. For example:

```
12 34 56
```

Interrupts are simulated at the 12th, 34th, and 56th CPU clock cycles. No operation is performed on the clock cycle value; the interrupt occurs exactly as the clock cycle value is written.

- **Relative.** You can also select a clock cycle that is relative to the time at which the last event occurred. A plus sign (+) before a clock cycle adds that value to the total clock cycles preceding it. For example:

```
12 +34 55
```

In this example, a total of three interrupts are simulated at the 12th, 46th (12 + 34), and 55th CPU clock cycles. You can mix both relative and absolute values in your input file.

- ❑ The **rpt {n | EOS}** parameter is optional and represents a repetition value.

You can have two forms of repetition to simulate interrupts:

- **Repetition on a fixed number of times.** You can format your input file to repeat a particular pattern for a fixed number of times. For example:

```
5 (+10 +20) rpt 2
```

The values inside the parentheses represent the portion that is repeated. Therefore, an interrupt is simulated at the 5th CPU cycle, then the 15th (5 + 10), 35th (15 + 20), 45th (35 + 10), and 65th (45 + 20) CPU clock cycles.

The *n* is a positive integer value.

- **Repetition to the end of simulation.** To repeat the same pattern throughout the simulation, add the string EOS to the line. For example:

```
10 (+5 +20) rpt EOS
```

Interrupts are simulated at the 10th CPU cycle, then the 15th (10 + 5), 35th (15 + 20), 40th (35 + 5), 60th (40 + 20), 65th (60 + 5), and 85th (65 + 20) CPU cycles, continuing in that pattern until the end of simulation.

Connecting your input file to the interrupt pin

To connect your input file to the interrupt pin, use the following command:

pinc *pinname, filename*

- ❑ The *pinname* identifies the interrupt pin
- ❑ The *filename* is the name of your input file.

Example 4–1 shows you how to connect your input file using the PINC command.

Example 4–1. Connecting the Input File With the PINC Command

Suppose you want to generate an external interrupt on INT4 at the 12th, 34th, 56th, and 89th clock cycles.

First, create a data file with an arbitrary name such as myfile:

```
12 34 56 89
```

To connect the input file to the pin, enter:

```
pinc, INT4, myfile
```

*Connects your data file
to the specific interrupt pin*

This command connects myfile to the pin. As a result, the simulator generates an external interrupt on at the 12th, 34th, 56th, and 89th clock cycles.

Disconnecting your input file from the interrupt pin

To end the interrupt simulation, disconnect the pin. You can do this with the following command:

```
pind pinname
```

The *pinname* parameter identifies the interrupt pin and must be one of the external interrupt pins (pins INT4–7, NMI).

The PIND command detaches the file from the input pin. After executing this command, you can connect another file to the same pin.

Listing the interrupt pins and connecting input files

To verify that your input file is connected to the correct pin, use the PINL command. The syntax for this command is:

```
pinl
```

The PINL command displays all of the unconnected pins first, followed by the connected pins. For a pin that has been connected, it displays the name of the pin and the absolute pathname of the file in the Command window.

Loading and Displaying Code

The main purpose of a debugging system is to allow you to load and run your programs in a test environment. This chapter tells you how to load your programs into the debugging environment, run them on the target system, and view the associated source code.

Topic	Page
5.1 Loading and Displaying Assembly Language Code	5-2
5.2 Displaying C Code	5-6

5.1 Loading and Displaying Assembly Language Code

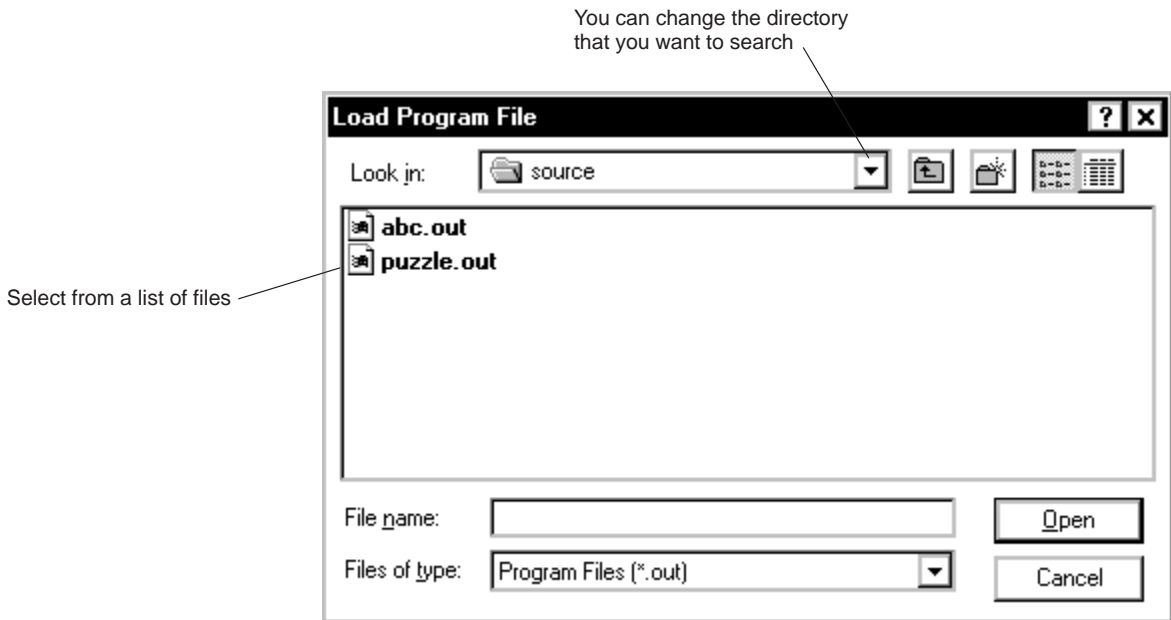
To debug a program, you must load the program's object code into memory. You create an object file by compiling, assembling, and linking your source files; see section 2.1, *Preparing Your Program for Debugging*, on page 2-2.

After you invoke the debugger, you can load object code and/or the symbol table associated with an object file.

Loading an object file and its symbol table

To load both an object file and its associated symbol table, follow these steps:

- 1) From the File menu, select Load Program. This displays the Load Program File dialog box:



- 2) Select the file that you want to open. To do so, you might need to change the working directory.
- 3) Click Open.

Loading an object file without its symbol table

You can load an object file *without* loading its associated symbol table. This is useful for reloading a program when memory has been corrupted.

To load an object file without its symbol table, select Reload Program from the File menu. The debugger reloads the file that you loaded last but does not load the symbol table.

If you want to load a new file without loading its associated symbol table, use the RELOAD command. The format for this command is:

reload *object filename*

Loading a symbol table only

You can load a symbol table without loading an object file. This is most useful in an emulation environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). In such an environment, loading the symbol table allows you to perform symbolic debugging and examine the values of C variables.

To load only a symbol table, select Load Symbols from the File menu. This displays the Load Symbols from File dialog box.

The File→Load Symbols menu option clears the existing symbol table before loading the new one but does not modify memory or set the program entry point.

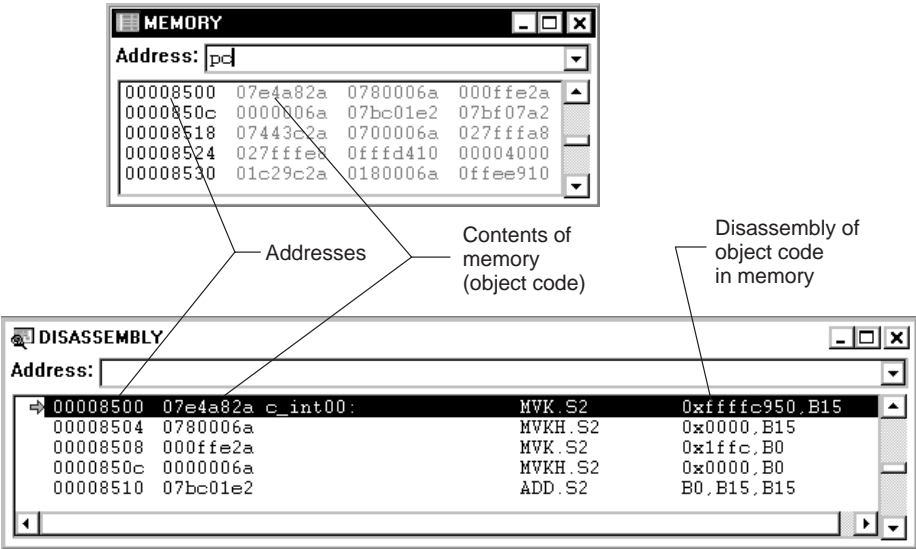
Loading code while invoking the debugger

You can load an object file when you invoke the debugger. (This has the same effect as using the File→Load Program menu option described on page 5-2.) To do this, enter the appropriate debugger-invocation command along with the name of the object file.

If you want to load only a file's symbol table when you invoke the debugger, use the `-s` option. (This has the same effect as using the File→Load Symbols menu option.) To do this, enter the appropriate debugger-invocation command along with the name of the object file and specify `-s` (see page 2-13 for more information).

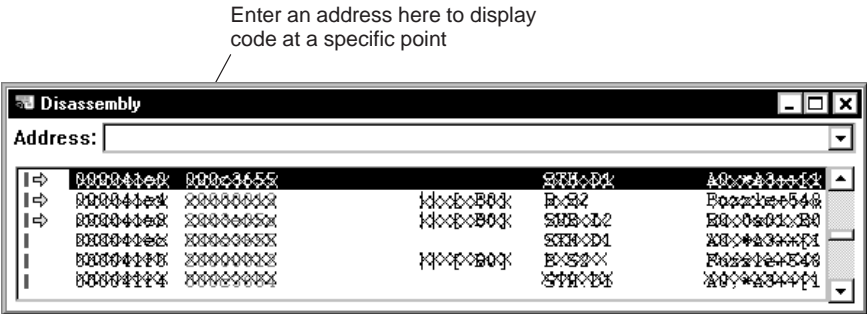
Displaying portions of disassembly

The assembly language code in the Disassembly window is the reverse assembly of program-memory contents. This code does not come from any of your text files or from the intermediate assembly files produced by the compiler.



When you invoke the debugger, it comes up in auto mode. If you load an object file when you invoke the debugger, the Disassembly window displays the reverse assembly of the object file that is loaded into memory. If you do not load an object file, the Disassembly window shows the reverse assembly of whatever is in memory, which may not be useful.

To display code beginning at a specific point, enter a new starting address in the Address field of the Disassembly window:



If you want to specify a hex address, be sure to prefix the address number with 0x; otherwise, the debugger treats the number as a decimal address.

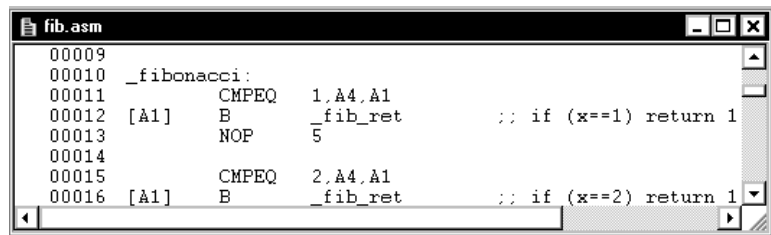
You can also move through the contents of the Disassembly window by using the scroll bar. Because the Disassembly window shows the reverse assembly of memory contents, the scroll bar handle is displayed in the middle of the scroll bar. The middle of the reverse assembly is defined as the most recent address or function name that you entered with the DASM command or in the Disassembly window's Address field. You can scroll up or down to see 1K bytes of reverse assembly on either side of the most recent address or function that you entered.



You can scroll through 1K bytes of reverse assembly above or below the scroll bar handle

Displaying assembly source code

If you assemble your code with the `-g` assembler option, the debugger displays the contents of your assembly source file in the File window, in addition to displaying the reverse assembly of memory contents in the Disassembly window. This allows you to view all assembly source comments and true assembly statements:



5.2 Displaying C Code

Unlike the assembly language code displayed in the Disassembly window, C code is not reconstructed from memory contents—the C code that you view is your original C source. You can display C code explicitly or implicitly:

- ☐ You can force the debugger to show C source by opening a C file or by entering the FUNC or ADDR command.
- ☐ In auto and mixed modes, the debugger automatically opens a File window if you are currently running C code.

Displaying the contents of a text file

To display the contents of any text file, follow these steps:

- 1) Use one of these methods to open the Open File dialog box:

- ☐ Click the Open icon on the toolbar:

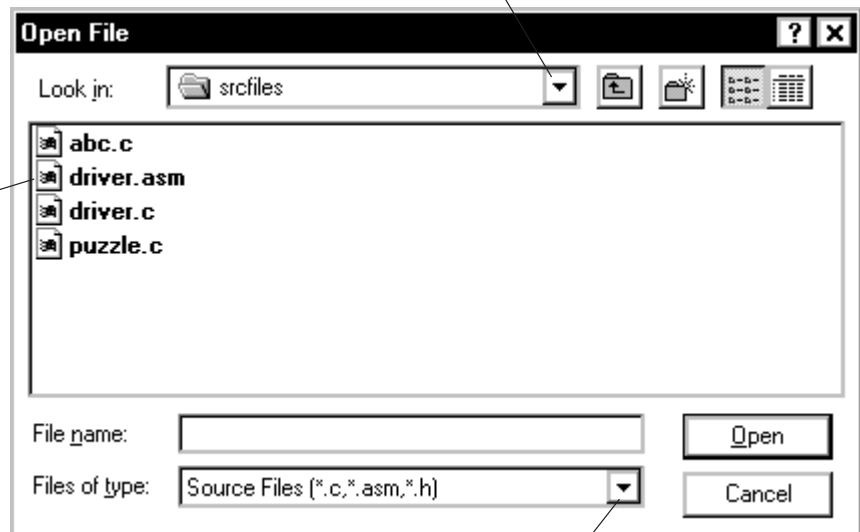


- ☐ From the File menu, select Open.

This displays the Open File dialog box:

You can change the directory that you want to search

Select from a list of files



Select the type of file you want to open

- 2) Select the file that you want to open. To do so, you might need to do one or more of the following actions:
 - ☐ Change the working directory.
 - ☐ Select the type of file that you want to open (for example, .c, .h).
- 3) Click Open.

The debugger opens a File window that contains the file that you selected. Although this command is most useful for viewing C code, you can use the Open File dialog box for displaying any text file. You might, for example, want to examine system files such as `autoexec.bat` or an initialization batch file. You can also view your original assembly language source files in the File window if you assemble your code with the `-g` assembler option. For every file that you open, the debugger displays the file in a new File window.

Displaying a C file *does not* load that file's object code. If you want to be able to run the program, you must load the file's associated object code as described in section 5.1, *Loading and Displaying Assembly Language Code*, on page 5-2.

Displaying a specific C function

To display a specific C function, use the FUNC command. The syntax for this command is:

func {*function name* | *address*}

FUNC modifies the display so that the code associated with the function or address that you specify is displayed within a File window. If you supply an *address* instead of a *function name*, the File window displays the function containing *address* and places the cursor at that line.

You can also use the functions in the Calls window to display a specific C function. This is similar to the FUNC or ADDR command but applies only to the functions listed in the Calls window. Choose one of these methods to display a function listed in the Calls window:

- ☐ Single-click the name of the C function.
- ☐ Select the name of the C function and press **(F9)**.

Displaying code beginning at a specific point

To display C or assembly code beginning at a specific point, use the ADDR command. The syntax for this command is:

addr {*address* | *function name*}

In a C display, ADDR works like the FUNC command, positioning the code starting at *address* or at *function name* as the first line of code in the File window. In mixed mode, ADDR affects both the File and Disassembly windows.

Running Code

To debug your programs, you must execute them on a debugging tool (the emulator or simulator). The debugger provides two basic types of commands to help you run your code:

- ☐ *Basic run commands* run your code without updating the display until you explicitly halt execution.
- ☐ *Single-step commands* execute assembly language or C code one statement at a time and update the display after each execution.

This chapter describes the basic run commands and the single-step commands, tells you how to halt program execution, and discusses using software breakpoints.

Topic	Page
6.1 Defining the Starting Point for Program Execution	6-2
6.2 Using the Basic Run Commands	6-4
6.3 Single-Stepping Through Code	6-8
6.4 Running Code Conditionally	6-11
6.5 Benchmarking	6-12
6.6 Halting Program Execution	6-13
6.7 Using Software Breakpoints	6-14

6.1 Defining the Starting Point for Program Execution

All run and single-step commands begin executing from the current PC. When you load an object file, the PC is automatically set to the starting point for program execution. You can easily identify the current PC by:

- ☐ Finding its entry in the CPU window
- ☐ Finding the line in the File or Disassembly window that has a yellow arrow next to it. To do this, execute one of these commands:

dasm PC

or

addr PC

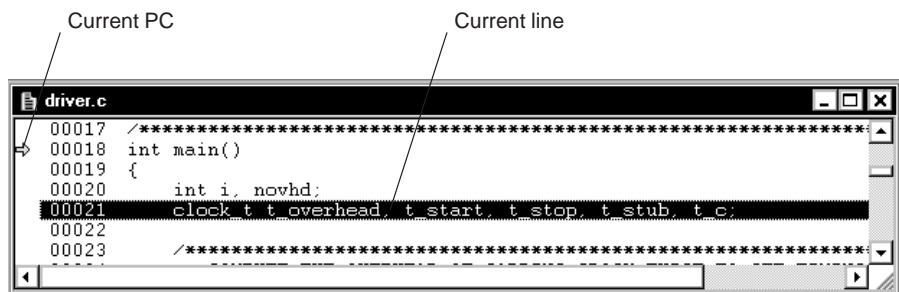
Sometimes you may want to modify the PC to point to a different position in your program. Choose one of these methods:

- ☐ If you executed some code and plan to rerun the program from the original program entry point, click the Restart icon on the toolbar:



Alternatively, you can select Restart from the Target menu.

- ☐ Set the PC to the current line in the File or Disassembly window. The current line is highlighted in the display:



To set the PC to the current line in the File or Disassembly window, follow these steps:

- 1) Open the context menu for the window. (For more information, see page 1-6.)
- 2) Select Set PC to Cursor from the context menu.

- ☐ Modify the PC's contents with one of these commands:
?PC = new value
or
eval pc = new value
- ☐ Modify the value of the PC in the CPU window. (For more information about changing values the displayed in the CPU window, see section 7.4, *Basic Methods for Changing Data Values*, on page 7-5.)

6.2 Using the Basic Run Commands

The debugger provides a basic set of run commands that allow you to do the following:

- ☐ Run an entire program
- ☐ Run code up to a specific point in a program
- ☐ Run code in the current C function
- ☐ Run code through breakpoints
- ☐ Run code while disconnected from the target system.

You can also use the debugger to reset the target system (emulator only) or simulator.

Running an entire program

To run the entire program, use one of these methods:

- ☐ Click the Run icon on the toolbar:



- ☐ From the Target menu, select Run.
- ☐ Press **F5**.
- ☐ From the command line, enter the RUN command. The format for this command is:

run [*expression*]

If you supply a logical or relational *expression*, the RUN command becomes a conditional run (see section 6.4 on page 6-11).

If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger executes *count* instructions, halts, then updates the display.

Note:

The emulator cannot step into an interrupt using the conditional RUN command with a 1 option (RUN 1). To halt on the first instruction of an interrupt service routine, you must set a software breakpoint or a program address breakpoint on the first instruction of the interrupt service routine and enter a RUN command.

When you run the entire program using one of these methods and do not supply an expression, the program executes until one of the following actions occurs:

- ☐ The debugger encounters a breakpoint. (For more information about how breakpoints affect a conditional run, see section 6.4 on page 6-11.)

- ☐ You click the Halt icon on the toolbar:



- ☐ You select Halt! from the Target menu.
- ☐ You press `(ESC)`.

Running code up to a specific point in a program

You can execute code up to a specific point in your program by using the GO command. The format for this command is:

go *[address]*

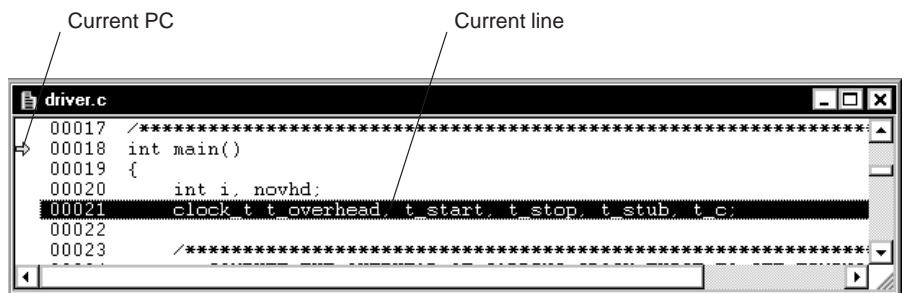
If you do not supply an *address* parameter, the program executes until one of the following actions occurs:

- ☐ The debugger encounters a breakpoint.
- ☐ You click the Halt icon on the toolbar:



- ☐ You select Halt! from the Target menu.
- ☐ You press `(ESC)`.

You can also execute code from the current PC to the current line in the File or Disassembly window. The current line is highlighted in the display:



To run code from the current PC to the current line in the File or Disassembly window, follow these steps:

- ☐ Open the context menu for the window. (For more information, see page 1-6.)
- ☐ Select Run to Cursor from the context menu.

Running the code in the current C function

You can execute the code in the current C function and halt when execution returns to the function's caller. To do so, use one of these methods:

- ☐ Click the Return icon on the toolbar:



- ☐ From the Target menu, select Return.

Breakpoints do not affect this command, but you can halt execution by doing one of the following:

- ☐ Click the Halt icon on the toolbar:



- ☐ From the Target menu, select Halt!.

- ☐ Press **(ESC)**.

Running code while disconnected from the target system (emulator only)

Use the RUNF command to disconnect the emulator from the target system while code is executing.

When you use the RUNF command, the debugger clears all breakpoints, disconnects the emulator from the target system, and causes the processor to begin execution at the current PC. You can quit the debugger, or you can continue to enter commands. However, any command that causes the debugger to access the target at this time produces an error.

Run Free is useful in a multiprocessor system. It is also useful in a system in which several target systems share an emulator; the Run Free option enables you to disconnect the emulator from one system and connect it to another.

Running code through breakpoints

You can use the debugger to execute code and run through breakpoints. This is referred to as a *continuous run*. When a breakpoint is encountered during a continuous run, execution does not halt. Instead, the debugger updates the display when a breakpoint is encountered.

To execute a continuous run, select Continuous Run from the Target menu.

To halt a continuous run, use one of the methods described in section 6.6 on page 6-13.

Resetting the simulator

You can use the debugger to reset the simulator by using a reset command. This is a *software* reset.

To execute a reset, select Reset Target from the Target menu.

If you are using the simulator and execute a software reset, the simulator simulates the 'C6x processor and peripheral reset operation, putting the processor in a known state.

Resetting the emulator

You can use the debugger to reset the target system by using a reset command. To execute a reset, select Reset Target from the Target menu.

6.3 Single-Stepping Through Code

Single-step execution is similar to running a program that has a breakpoint set on each line. The debugger executes one statement, updates the display, and halts execution. (You can supply a parameter that tells the debugger to single-step continuously; the debugger updates the display after each statement is executed.) You can single-step through assembly language code or C code.

The debugger supports several commands for single-stepping through a program. Command execution can vary, depending on whether you are single-stepping through C code or assembly language code.

Note:

If you use the STEP command or Target menu Step option to single-step through assembly language code, the debugger ignores interrupts.

Each of the single-step commands in this section has an optional *expression* parameter that works like this:

- ☐ If you do not supply an *expression*, the program executes a single statement, then halts.
- ☐ If you supply a logical or relational *expression*, this becomes a conditional single-step execution (see section 6.4 on page 6-11).
- ☐ If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger single-steps *count* assembly language statements unless you are currently in C code. If you are currently in C code, the debugger single-steps *count* C statements.

Single-stepping through assembly language or C code

The debugger has a basic single-step command that allows you to single-step through assembly language or C code. If you are currently in assembly language code, the debugger executes one assembly language statement at a time. If you are currently in C code, the debugger executes one C statement at a time.

If you are in mixed mode, the debugger executes one assembly language statement at a time.

To use the basic single-step command, choose one of these methods:

- ☐ Click the Step icon on the toolbar:



- ☐ From the Target menu, select Step.
- ☐ Press **(F8)**.
- ☐ From the command line, enter the STEP command. The format for this command is:

step [expression]

When you use the basic single-step command in C code and encounter a function call, the step command shows you the single-step execution of the called function (assuming that the function was compiled with the compiler's `-g` option). When function execution completes, single-step execution returns to the caller. If the function was not compiled with the `-g` option, the debugger executes the function but does not show single-step execution of the function.

For more information about the compiler's `-g` option, see the *TMS320C6x Optimizing C Compiler User's Guide*.

Single-stepping through C code

The basic single-step command, described in the *Single-stepping through assembly language or C code* section, always executes one statement at a time—no matter whether you are in assembly language code or in C code. If you want to single-step in terms of a C statement and execute all assembly language statements associated with a single C statement before updating the display, use the C single-step command. To use the C single-step command, choose one of these methods:

- ☐ Click the Single Step C icon on the toolbar:



- ☐ From the Target menu, select Step C.
- ☐ Press **(CONTROL) (F8)**.
- ☐ From the command line, enter the CSTEP command. The format for this command is:

cstep [expression]

Continuously stepping through code

You can use the debugger to watch your code as it executes. You can step through code continuously until the debugger reaches a breakpoint. This is referred to as a *continuous step*. When a breakpoint is encountered during a continuous step, execution halts.

To execute a continuous step, select Continuous Step from the Target menu.

If no breakpoints are set, you can halt a continuous step by using one of the methods described in section 6.6 on page 6-13.

Single-stepping through code and stepping over C functions

Besides single-stepping through *all* code with the basic single-step commands, you can single-step through assembly language or C code and step *over* function calls. This type of single-stepping always steps to the *next* consecutive statement and never shows the execution of called functions. You can use the *next* single-step command in one of two ways:

- ☐ To use the next single-step command and single-step in terms of assembly language or C statements (similar to the basic single-step command), choose one of these methods:

- Click the Next Statement icon on the toolbar:



- From the Target menu, select Next.
 - Press **F10**.
 - From the command line, enter the NEXT command. The format for this command is:

next [expression]

- ☐ To use the next single-step command and single-step in terms of C statements (similar to the C single-step command), choose one of these methods:

- Click the Next C Statement icon on the toolbar:



- From the Target menu, select Next C.
 - Press **CONTROL F10**.
 - From the command line, enter the CNEXT command. The format for this command is:

cnext [expression]

6.4 Running Code Conditionally

The RUN, STEP, CSTEP, NEXT, and CNEXT commands all have an optional *expression* parameter that can be a relational or logical expression. This type of expression uses one of the following operators as the highest precedence operator in the expression:

>	> =	<
< =	= =	! =
&&		!

When you use this type of expression with these commands, the command becomes a conditional run. The debugger executes the command repeatedly for as long as the expression evaluates to true.

You must use software breakpoints with conditional runs; the expression is evaluated each time the debugger encounters a breakpoint. (Breakpoints are described in section 6.7 on page 6-14.) For single-step commands, the expression is evaluated at each statement. Each time the debugger evaluates the conditional expression, it updates the screen.

Generally, you should set the breakpoints on statements that are related in some way to the expression. For example, if you are observing a particular variable in a Watch window, you may want to set breakpoints on statements that affect that variable and to use that variable in the expression.

6.5 Benchmarking

The debugger allows you to keep track of the number of CPU clock cycles consumed by a particular section of code. The debugger maintains the count in a pseudoregister named CLK. This process is referred to as *benchmarking*.

Benchmarking code is a multiple-step process:

- Step 1:** Set a software breakpoint at the statement that marks the beginning of the section of code that you want to benchmark. (For more information about setting software breakpoints, see section 6.7 on page 6-14.)
- Step 2:** Set a software breakpoint at the statement that marks the end of the section of code that you want to benchmark.
- Step 3:** Enter any run command to execute code up to the first breakpoint.
- Step 4:** From the Target menu, select Run Benchmark.

When the processor halts at the second breakpoint, the value of CLK is valid. To display it, use the ? command or enter it into the Watch window with the Setup→Watch Variable menu option. This value is valid until you enter another run command.

Notes:

- 1) Run Benchmark (or RUNB command) counts CPU clock cycles from the current PC to the breakpoint. This count is not cumulative. You cannot add the number of clock cycles between points A and B to the number of cycles between points B and C to learn the number of cycles between points A and C. This situation occurs because of pipeline filling and flushing.
 - 2) The value in CLK is valid only after using a Run Benchmark command that is terminated by a software breakpoint.
 - 3) When programming in C, avoid using a variable named CLK.
 - 4) The RUNB command accesses the analysis module to count CPU clock cycles. If you have set up an instruction breakpoint, the debugger halts on that breakpoint in addition to your software breakpoints.
-

6.6 Halting Program Execution

Whenever you are running or single-stepping code, program execution halts automatically if the debugger encounters a breakpoint or if it reaches a particular point where you told it to stop (by supplying a *count* or an *address* with the RUN, GO, or any of the single-step commands). If you want to halt program execution explicitly, you can use one of these methods:

- ☐ Click the Halt icon on the toolbar:



- ☐ From the Target menu, select Halt!.
- ☐ Press `(ESC)`.

After halting execution, you can continue program execution from the current PC by reissuing any of the run or single-step commands.

What happens when you halt the emulator

If you are using the emulator version of the debugger, any of the above methods halts the target system after you have commanded the debugger to run code while disconnected from the target (run free).

When you invoke the debugger, it automatically executes a HALT command. Thus, if you use the RUNF command, quit the debugger, and later reinvoke the debugger, you effectively reconnect the emulator to the target system and run the debugger in its normal mode of operation. When you invoke the debugger, use the `-s` option to preserve the current PC and memory contents.

To provide memory visibility, the emulator completes load and store instructions that are already in the pipeline while halting. After halting, the emulator provides visibility to the register file on pipeline cycle boundaries. The memory state provided by the emulator reflects the data that the next memory operation accesses. The emulator saves the data read from the load instructions and restores this data to the pipeline when the emulator resumes execution.

Also, when the emulator halts, all the memory controllers that are internal to the processor halt. This allows the memory controllers' pipeline to remain synchronized with the processor's pipeline.

6.7 Using Software Breakpoints

During the debugging process, you may want to halt execution temporarily so that you can examine the contents of selected variables, registers, and memory locations before continuing with program execution. You can do this by setting *software breakpoints* at critical points in your code. You can set software breakpoints in assembly language code and in C code. A software breakpoint halts any program execution, whether you are running or single-stepping through code.

Software breakpoints are especially useful in combination with conditional execution (described in section 6.4 on page 6-11).

When you set a software breakpoint, the debugger highlights the breakpointed line with this prefix: ●.

If you set a breakpoint in the disassembly, the debugger also highlights the associated C statement if the debugger has access to the C source. If you set a breakpoint in the C source, the debugger also highlights the associated statement in the disassembly. (If more than one assembly language statement is associated with a C statement, the debugger highlights the first of the associated assembly language statements.)

A breakpoint is set at this C statement

A breakpoint is also set at the associated assembly language statement

```

00017  /*****
● 00018  int main()
00019  {
00020      int i, novhd;
00021      clock_t t_overhead, t_start, t_stop, t_stub, t_c;
00022

```

DISASSEMBLY			
Address:			
● 00007c68	01bd14f6	main:	STW.D2 E3.*B15--[8]
00007c6c	053cc2f4		STW.D2 A10.*+B15[6]
00007c70	05bce2f4		STW.D2 A11.*+B15[7]
00007c74	00015c10		B.S1 clock
00007c78	00004000		NOP 3

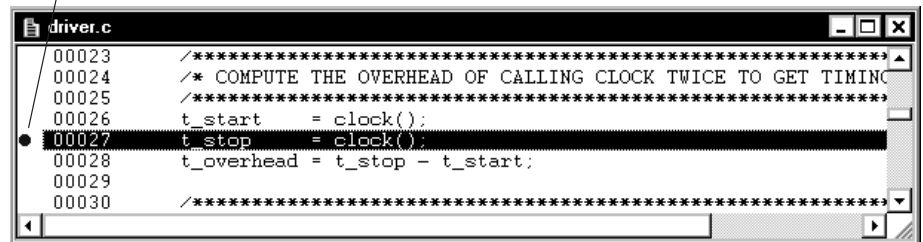
Notes:

- 1) After execution is halted by a breakpoint, you can continue program execution by reissuing any of the run or single-step commands.
- 2) You can set up to 200 breakpoints. If you are using the emulator, you can set only one software breakpoint per set of parallel instructions that execute in the same cycle.
- 3) You cannot set multiple breakpoints at the same statement.

Setting a software breakpoint

To set a breakpoint, click next to the statement in the Disassembly or File window where you want the breakpoint to occur. When you click next to a statement in the Disassembly or File window, a breakpoint symbol is shown:

A breakpoint is set on this statement



Another way to set a breakpoint is to use the context menu for the File or Disassembly window. You can set a breakpoint on the current line in the File or Disassembly window. The current line is highlighted in the display.

To set a breakpoint on the current line in the File or Disassembly window, follow these steps:

- ☐ Open the context menu for the window. (For more information, see page 1-6.)
- ☐ Select Toggle Breakpoint from the context menu.

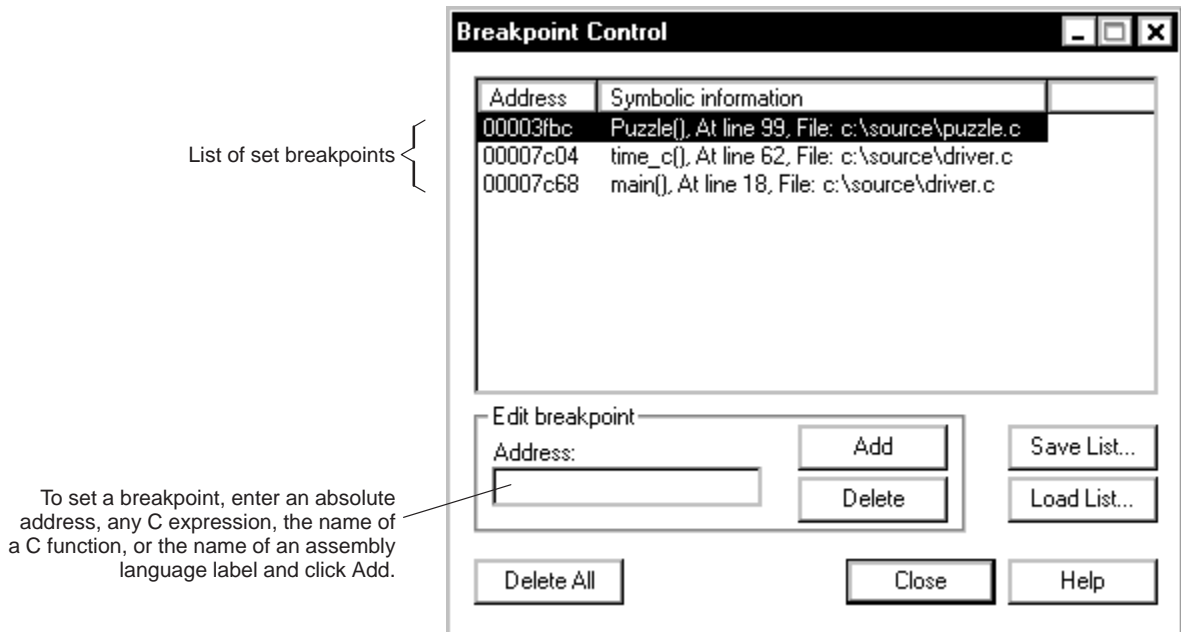
You can also set a breakpoint by using the Breakpoint Control dialog box. To open the Breakpoint Control dialog box, use one of these methods:

- ☐ Click the Breakpoint Dialog icon on the toolbar:



- ☐ From the Setup menu, select Breakpoints.

This displays the Breakpoint Control dialog box:



To set a breakpoint, follow these steps:

- 1) In the Address field of the Breakpoint Control dialog box, enter an absolute address, any C expression, the name of a C function, or the name of an assembly language label. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.
- 2) Click Add. The new breakpoint appears in the breakpoint list.
- 3) Click Close to close the Breakpoint Control dialog box.

Clearing a software breakpoint

There are several ways to clear a software breakpoint. If you clear a breakpoint from an assembly language statement, the breakpoint is also cleared from any associated C statement; if you clear a breakpoint from a C statement, the breakpoint is also cleared from the associated statement in the disassembly.

To clear a breakpoint, click the breakpoint symbol (●) in the File or Disassembly window.

Another way to clear a breakpoint is to use the context menu for the File or Disassembly window:

- 1) Select the line in the File or Disassembly window from which you want to remove the breakpoint.
- 2) From the context menu for the window, select Toggle Breakpoint.

You can also clear a breakpoint by using the Breakpoint Control dialog box (see the illustration on page 6-16):

- 1) Open the Breakpoint Control dialog box by using one of these methods:

- ☐ Click the Breakpoint Dialog icon on the toolbar:



- ☐ From the Setup menu, select Breakpoints.

- 2) Select the address of the breakpoint that you want to clear.
- 3) Click Delete. The breakpoint is removed from the breakpoint list.
- 4) Click Close to close the Breakpoint Control dialog box.

Clearing all software breakpoints

To clear all software breakpoints, follow these steps:

- 1) Open the Breakpoint Control dialog box by using one of these methods:

- ☐ Click the Breakpoint Dialog icon on the toolbar:



- ☐ From the Setup menu, select Breakpoints.

- 2) Click Delete All.
- 3) Click Close to close the Breakpoint Control dialog box.

Saving breakpoint settings

Software breakpoint settings are lost when you exit the debugger. However, you can save the list of breakpoints that you have set by following these steps:

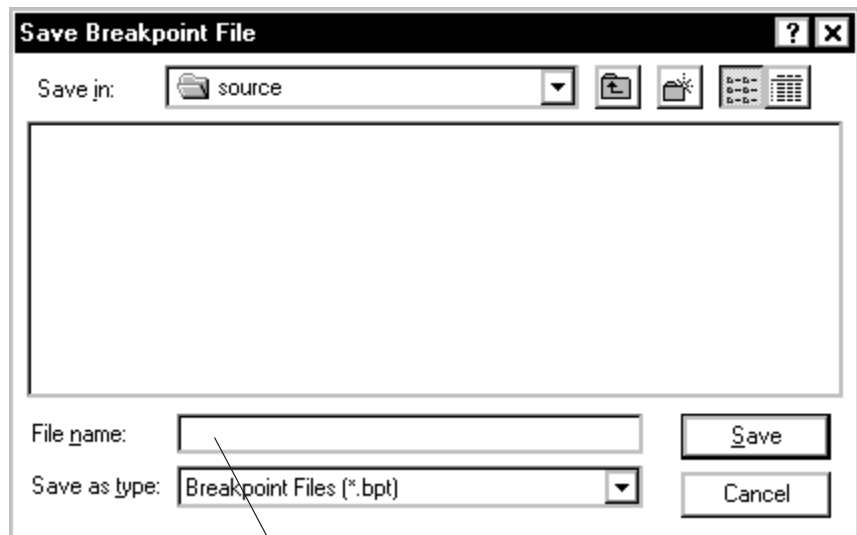
- 1) Open the Breakpoint Control dialog box by using one of these methods:

- ☐ Click the Breakpoint Dialog icon on the toolbar:



- ☐ From the Setup menu, select Breakpoints.

- 2) Click Save List. This displays the Save Breakpoint File dialog box:



Enter a name for the breakpoint file. Use a .bpt extension.

- 3) Select the directory where you want the file to be saved.
- 4) In the File name field, enter a name for the breakpoint file. You can use a .bpt extension to identify the file as a breakpoint file.
- 5) Click Save.
- 6) In the Breakpoint Control dialog box, click Close.

Notes:

- 1) The breakpoint file is editable.
 - 2) You can execute the breakpoint file with the TAKE command to automatically set up the breakpoints that are defined in the file.
 - 3) You can include the breakpoint file in your initialization batch file.
-

Loading saved breakpoint settings

To load a list of saved breakpoints, follow these steps:

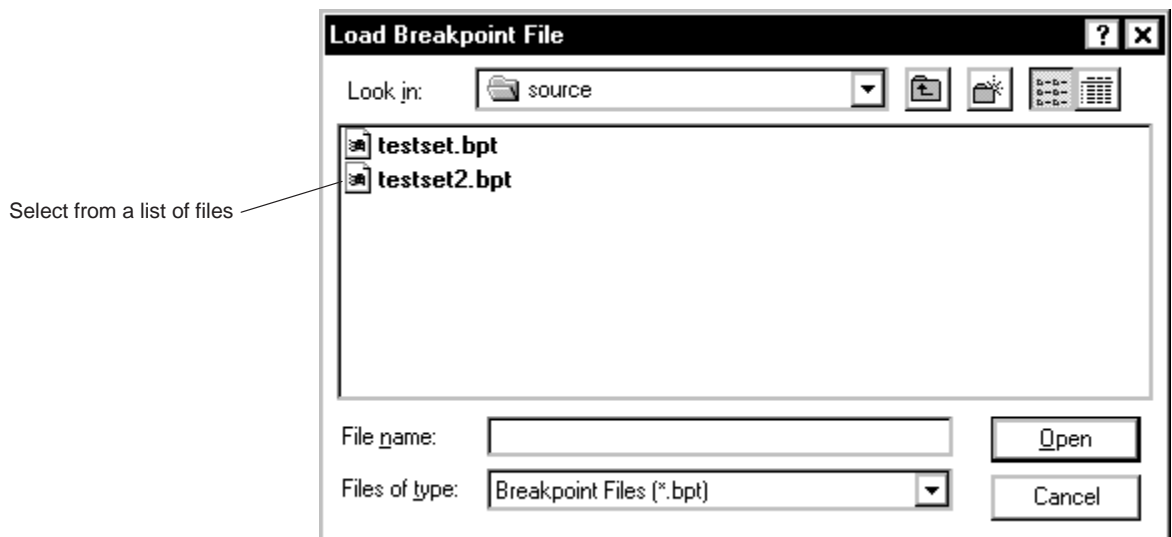
- 1) Open the Breakpoint Control dialog box by using one of these methods:

- ☐ Click the Breakpoint Dialog icon on the toolbar:



- ☐ From the Setup menu, select Breakpoints.

- 2) Click Load List. This displays the Load Breakpoint File dialog box:



- 3) Select the file that you want to open. To do so, you might need to change the working directory.
- 4) Click Open.
- 5) In the Breakpoint Control dialog box, click Close.

Note:

When you load a breakpoint file, breakpoints that you have defined previously in your debugging session are not cleared but remain in effect.

Managing Data

The debugger allows you to examine and modify many types of data related to the 'C6x and to your program. You can display and modify these values:

- ☐ The contents of individual memory locations or a range of memory
- ☐ The contents of 'C6x registers
- ☐ Variables, including scalar types (ints, chars, etc.) and aggregate types (arrays, structures, etc.)

Topic	Page
7.1 Where Data Is Displayed	7-2
7.2 How the Emulator Displays Data for Load and Store Instructions	7-2
7.3 Basic Commands for Managing Data	7-3
7.4 Basic Methods for Changing Data Values	7-5
7.5 Managing Data in Memory	7-7
7.6 Managing Register Data	7-13
7.7 Managing Data in a Watch Window	7-18
7.8 Displaying Data in Alternative Formats	7-22

7.1 Where Data Is Displayed

Various types of data are displayed in one of three dedicated windows.

Type of Data	Window Name	Purpose
Memory locations	Memory window	Displays the contents of a range of memory
Register values	CPU window	Displays the contents of 'C6x registers
Pointer data, variables, aggregate types, and specific memory locations or registers	Watch window	Displays selected data

The three dedicated windows are referred to as *data-display windows*.

7.2 How the Emulator Displays Data for Load and Store Instructions

The 'C6x emulator allows you to view the register file on pipeline cycle boundaries. When the emulator is halting and resuming execution, it maintains pipeline and memory synchronization; however, the CPU and Memory windows do not necessarily reflect this synchronization when displaying the data from load and store instructions.

While halting, the emulator completes the store instructions that have passed the E1 phase of the pipeline. The emulator immediately updates the Memory window with the data from the completed store instructions to reflect what the next load instruction reads from memory.

Unlike store instructions, load instructions shown in the register file displayed in the CPU window reflect the latency of the pipeline. When halting, the emulator completes the load instructions that are currently in the pipeline. The data from these load instructions is saved by the emulator and restored to the pipeline when execution is resumed.

7.3 Basic Commands for Managing Data

The debugger provides special-purpose commands for displaying and modifying data in dedicated windows. The debugger also supports several general-purpose commands that you can use to display or modify any type of data.

Determining the type of a variable

If you want to know the type of a variable or function, use the **WHATIS** command. The syntax for this command is:

whatis *symbol*

The *symbol*'s data type is then listed in the display area of the Command window. The *symbol* can be any variable (local, global, or static), a function name, a structure tag, a typedef name, or an enumeration constant.

Command	Result Displayed in the Command Window
whatis aai	int aai[10][5];
whatis xxx	struct xxx { int a; int b; int c; int f1 : 2; int f2 : 4; struct xxx *f3; int f4[10]; }

Evaluating an expression

The **?** (evaluate expression) command evaluates an expression and shows the result in the display area of the Command window. The syntax for this command is:

? *expression*

The *expression* can be any C expression, including an expression with side effects. However, you cannot use a string constant or function call in the *expression*.

If the result of *expression* is scalar, the debugger displays the result as a decimal value in the Command window. If *expression* is a structure or array, the debugger displays the entire contents of the structure or array; you can halt long listings by pressing **ESC**.

Here are some examples that use the ? command.

Command	Result Displayed in the Command Window
? aai	aai[0][0] 1 aai[0][1] 23 aai[0][2] 45 . . .
? j	4194425
? j=0x5a	90

The EVAL (evaluate expression) command behaves like the ? command *but does not show the result* in the display area of the Command window. The syntax for this command is:

eval *expression*
or
e *expression*

EVAL is useful for assigning values to registers or memory locations in a batch file, where it is not necessary to display the result.

For information about the PDM version of the EVAL command, refer to section 11.9, *Evaluating Expressions*, on page 11-21.

7.4 Basic Methods for Changing Data Values

The debugger provides you with a great deal of flexibility in modifying various types of data. You can use the debugger's overwrite editing capability, which allows you to change a value simply by typing over its displayed value. You can also use the data-management commands for more complex editing.

Editing data displayed in a window

Use overwrite editing to modify data in a data-display window; you can edit:

- ☐ Registers displayed in the CPU window
- ☐ Memory contents displayed in a Memory window
- ☐ Values or elements displayed in a Watch window

To modify data in a data-display window, follow these steps:

- 1) Select the data item that you want to modify. Choose one of these methods:
 - ☐ Double-click the data item that you want to modify.
 - ☐ Select the data item that you want to modify and press **(F9)**.
- 2) Type the new information. If you make a mistake or change your mind, press **(ESC)**; this resets the field to its original value.
- 3) When you finish typing the new information, press **(↵)** or click on another data value. This replaces the original value with the new value.

Editing data using expressions that have side effects

Using the overwrite editing feature to modify data is straightforward. However, data-management methods take advantage of the fact that C expressions are accepted as parameters by most debugger commands and that C expressions can have *side effects*. When an expression has a side effect, the value of some variable in the expression changes as the result of evaluating the expression.

Side effects allow you to coerce many commands into changing values for you. Specifically, it is most helpful to use **?** and **EVAL** to change data as well as display it.

For example, if you want to see what is in register A3, you can enter:

? A3 **(↵)**

? A3++ **(↵)**

Side effect: increments the contents of A3 by 1

eval --A3 **(↵)**

Side effect: decrements the contents of A3 by 1

? A3 = 8 **(↵)**

Side effect: sets A3 to 8

eval A3/=2 **(↵)**

Side effect: divides contents of A3 by 2

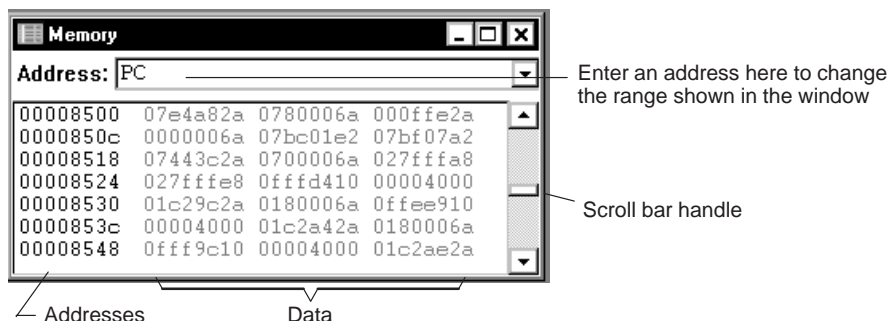
Not all expressions have side effects. For example, if you enter ? **A3+4**, the debugger displays the result of adding 4 to the contents of A3 but does not modify A3's contents. Expressions that have side effects must contain an assignment operator or an operator that implies an assignment. Operators that can cause a side effect are:

=	+=	-=	*=	/=
%=	&=	^=	=	<<=
	>>=	++	--	

7.5 Managing Data in Memory

The main way to observe memory contents is to view the display in a Memory window. In mixed and assembly modes, the debugger displays the default Memory window automatically (labeled Memory). You can open any number of additional Memory windows to display different memory ranges. Figure 7–1 shows the default Memory window.

Figure 7–1. The Default Memory Window



The amount of memory that you can display in a Memory window is limited by the size of the window (which is limited only by your monitor's screen size).

The debugger allows you to change the memory range displayed in the Memory window and to open additional Memory windows. The debugger also allows you to change the values at individual locations; for more information, see section 7.4, *Basic Methods for Changing Data Values*, on page 7-5.

Changing the memory range displayed in a Memory window

To change the memory range displayed in a Memory window, enter a new starting address in the Address field of the Memory window, as shown in Figure 7–1. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.

You can also change the display of any data-display window—including the Memory window—by scrolling through the window's contents. In the Memory window, the scroll bar handle is displayed in the middle of the scroll bar (see Figure 7–1). The middle of memory contents is defined as the most recent starting address that you entered in the Address field of the Memory window or with the MEM command (described on page 12-32). You can scroll up or down to see 1K bytes of memory on either side of the current starting address.

Opening an additional Memory window


To open an additional Memory window, use the MEM command. The syntax for this command is:

mem *expression* [, [*display format*] [, *window name*]]

- The *expression* represents the address of the first entry in the Memory window. The end of the range is defined by the size of the window: to show more memory locations, make the window larger; to show fewer locations, make the window smaller.

The *expression* can be an absolute address, a symbolic address, or any C expression. Here are some examples:

- **Absolute address.** Suppose that you want to display data memory beginning from the very first address. You might enter this command:

```
mem 0x0 
```

Memory window addresses are shown in hexadecimal format. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.

- **Symbolic address.** You can use any defined C symbol as an *expression* parameter. For example, if your program defined a symbol named *SYM*, you could enter this command:

```
mem &SYM 
```

Prefix the symbol with the & operator to use the address of the symbol.

- **C expression.** If you use a C expression as a parameter, the debugger evaluates the expression and uses the result as a memory address:

```
mem SP - A0 + label 
```

```
mem SP - AR0 + label 
```

- The *display format* parameter is optional. When used, the data is displayed in the selected format, as shown in Table 7–3 on page 7-22.
- Use the *window name* parameter to name the additional Memory window. The debugger appends the *window name* to the Memory window label. If you do not supply a name, the debugger does not open a new window; it simply updates the default Memory window to reflect the changes.

Displaying memory contents while you are debugging C

If you are debugging C code in auto mode, you do not see a Memory window—the debugger does not show the Memory window in the C-only display. However, there are several ways to display memory in this situation.

Hint: If you want to use the *contents* of an address as a parameter, be sure to prefix the address with the C indirection operator (*).

- ☐ If you have only a temporary interest in the contents of a specific memory location, you can use the ? command to display the value at this address. For example, if you want to know the contents of memory location 26 (hex), you could enter:

```
? *0x26
```

The debugger displays the memory value in the display area of the Command window.

- ☐ If you want to observe a specific memory location over a longer period of time, you can display it in a Watch window. Use the Setup→Watch Variable... menu option to do this:

- 1) In the Expression field, enter `*0x26`.
- 2) In the Format combo box, enter x-Hexadecimal.
- 3) Click OK.

The debugger displays the memory value in the Watch window.

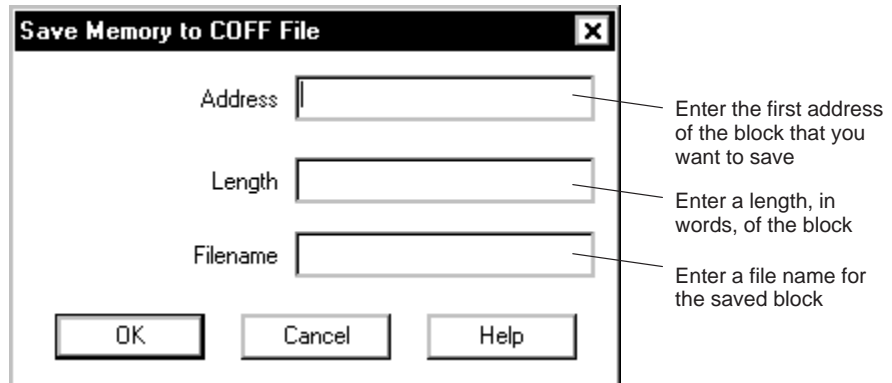
- ☐ You can also use the DISP command to display memory contents in a Watch window. The Watch window shows memory in an array format with the specified address as member [0]. In this situation, you can also use casting to display memory contents in a different numeric format:

```
disp *(float *)0x26
```

Saving memory values to a file

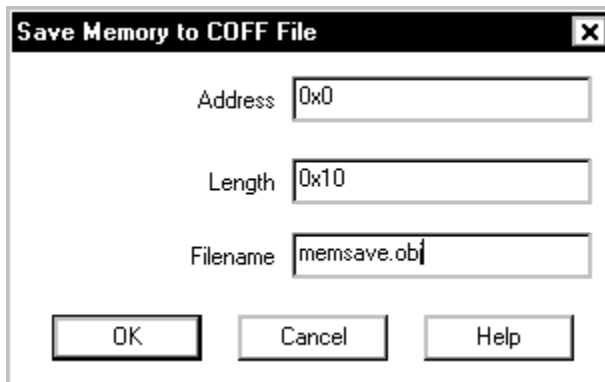
Sometimes it is useful to save a block of memory values to a file. You can use the Memory→Save menu option to do this; the files are saved in COFF format.

- 1) From the Memory menu, select Save. This displays the Save Memory to COFF File dialog box:



- 2) In the Address field, enter the first address in the block that you want to save. To specify a hex address, prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.
- 3) In the Length field, enter a length, in words, of the block. This parameter can be any C expression.
- 4) In the Filename field, enter a name for the saved block of memory. If you do not supply an extension, the debugger adds a .obj extension.
- 5) Click OK.

For example, to save the values in data memory locations 0x0000–0x003F to a file named memsave.obj, you could enter:



The dialog box titled "Save Memory to COFF File" has three input fields: "Address" with the value "0x0", "Length" with the value "0x10", and "Filename" with the value "memsave.obj". At the bottom are three buttons: "OK", "Cancel", and "Help".

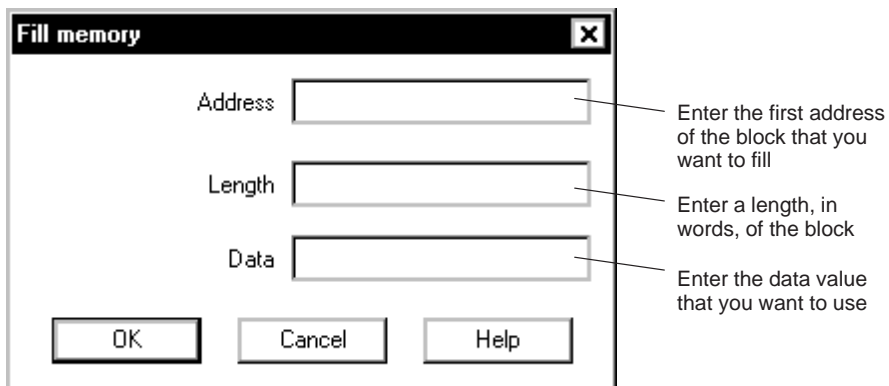
The value of the Length field is measured in words and data memory locations are measured in bytes. For this example, you must enter a length of 0x10 words to equal 0x40 bytes (0x0000–0x0040).

To reload memory values that were saved in a file, use the File→Load Program menu option.

Filling a block of memory

Sometimes it is useful to fill an entire block of memory at once with a particular value. You can fill a block of memory word by word using the Memory→Fill Word command.

- 1) From the Memory menu, select Fill Word. This displays the Fill Memory dialog box:

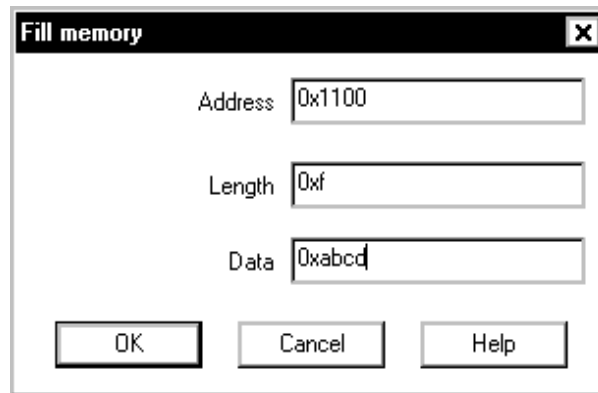


The dialog box titled "Fill memory" has three input fields: "Address", "Length", and "Data". To the right of the dialog box, three lines of text with arrows pointing to the corresponding fields provide instructions: "Enter the first address of the block that you want to fill" points to the Address field, "Enter a length, in words, of the block" points to the Length field, and "Enter the data value that you want to use" points to the Data field. At the bottom are three buttons: "OK", "Cancel", and "Help".

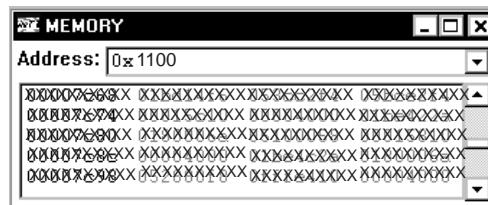
- 2) In the Address field, enter the first address in the block that you want to fill. To specify a hex address, prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.

- 3) In the Length field, enter a length, in words, of the block.
- 4) In the Data field, enter a value that you want placed in each word in the block.
- 5) Click OK.

For example, to fill memory locations 0x1100–0x113B with the value 0xABCD, you could enter:



If you want to check whether memory has been filled correctly, you can change the Memory window display to show the block of memory beginning at memory address 0x1100:



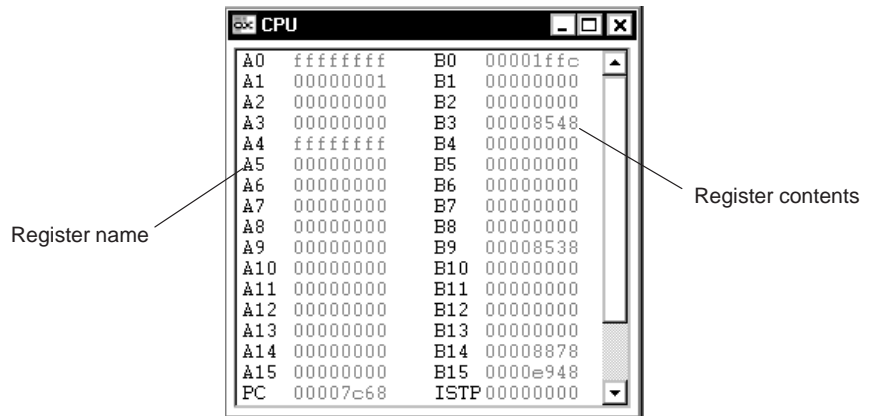
Change the Memory display by entering a new address

You can also use the debugger to fill a block of memory byte by byte by using the Memory→Fill Byte command.

- 1) From the Memory menu, select Fill Byte. This displays the Fill Memory—Byte dialog box.
- 2) In the Address field, enter the first address in the block that you want to fill. To specify a hex address, prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.
- 3) In the Length field, enter a length, in bytes, of the block.
- 4) In the Data field, enter a value that you want placed in each byte in the block.
- 5) Click OK.

7.6 Managing Register Data

In mixed and assembly modes, the debugger maintains a CPU window that displays the contents of individual registers.



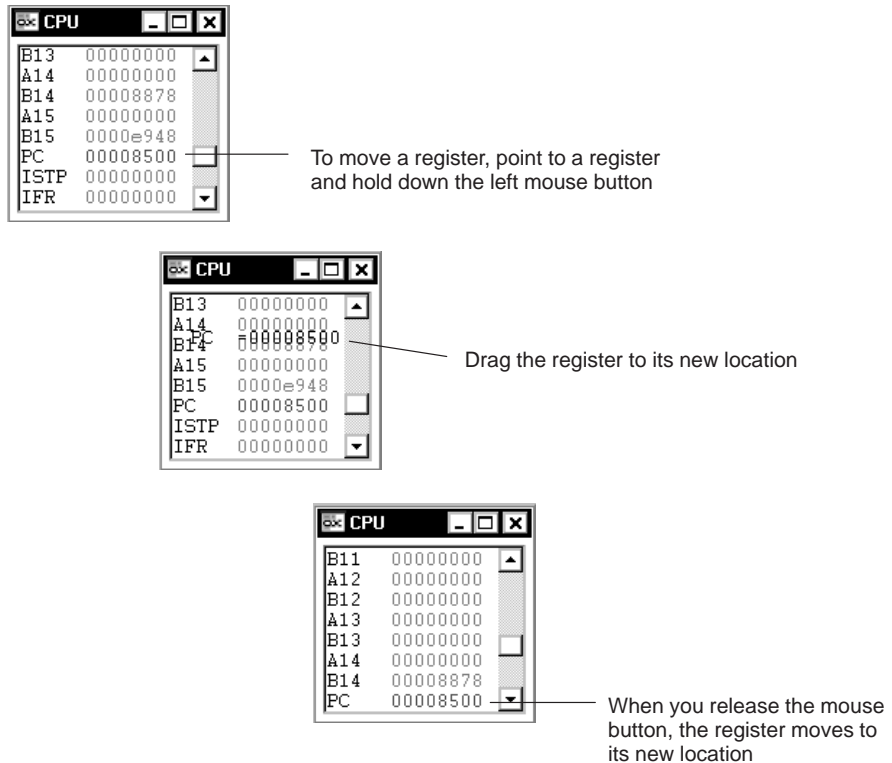
The debugger provides commands that allow you to display and modify the contents of specific registers. You can use the data-management commands or the debugger's overwrite editing capability to modify the contents of any register displayed in the CPU or Watch window. For more information, see section 7.4, *Basic Methods for Changing Data Values*, on page 7-5.

Displaying register contents

The main way to observe register contents is to view the display in the CPU window. However, you may not be interested in all of the registers; if you are interested in only a few registers, you might want to make the CPU window small and use the extra screen space for the Disassembly or File window.


You can also reorder the registers in the CPU window and display the ones that you are most interested in at the top of the CPU window. To do so, use the drag-and-drop method, as shown in Figure 7-2.

Figure 7–2. Reordering Registers in the CPU Window Using the Drag-and-Drop Method



In addition to the CPU window, you can observe the contents of selected registers by using the ? (evaluate expression) command or Setup→Watch Variable menu option:

- ☐ If you have only a temporary interest in the contents of a register, you can use the ? command to display the register's contents. For example, if you want to know the contents of A3, you could enter:

? A3 

The debugger displays A3's current contents in the display area of the Command window.

- ❑ If you want to observe a register over a longer period of time, you can use Setup→Watch Variable to display the register in a Watch window. For example, if you want to observe the control status register (CSR), you could enter:

The screenshot shows the 'Watch Add' dialog box with the following settings:

- Value**
 - Expression:
 - Global variables:
 - Local variables:
 - Static variables:
 - Registers:
- Format:**
- Label:**
- Window name:**
- Buttons:** OK, Apply, Close, Help

This adds the CSR to the Watch window in hexadecimal format and labels it as *Control Status Register*. The register's contents are continuously updated, just as if you were observing the register in the CPU window.

When you are debugging C in auto mode, the ? command and Setup→Watch Variable menu option are useful because the debugger does not show the CPU window in the C-only display.

Accessing single-precision floating-point registers

The debugger allows you to display the registers in the two general-purpose register files (A and B) in single-precision floating-point format. Table 7–1 lists the pseudoregister name for each general-purpose register.

Table 7–1. Pseudoregister Names for Single-Precision Floating-Point Registers

Register Name	Pseudoregister Name	Register Name	Pseudoregister Name
A0	FA0	B0	FB0
A1	FA1	B1	FB1
A2	FA2	B2	FB2
A3	FA3	B3	FB3
A4	FA4	B4	FB4
A5	FA5	B5	FB5
A6	FA6	B6	FB6
A7	FA7	B7	FB7
A8	FA8	B8	FB8
A9	FA9	B9	FB9
A10	FA10	B10	FB10
A11	FA11	B11	FB11
A12	FA12	B12	FB12
A13	FA13	B13	FB13
A14	FA14	B14	FB14
A15	FA15	B15	FB15

You can display the contents of these registers by using the ? (evaluate expression) command or Setup→Watch Variable menu selection.

- ☐ For example, assume B15 = 0xE908. If you want to evaluate the FB15 pseudoregister, enter:

```
?fb15
```

The debugger shows the result in the display area of the Command window:

```
?fb15 8.3595861e-041
```

- ☐ To modify the FA15 pseudoregister and set it equal to 15.75, enter:

```
?fa15 = 15.75
```

The debugger displays the following in the display area of the Command window:

```
?fa15 = 15.75 1.575000e+001
```

Accessing double-precision floating-point registers


The debugger allows you to access double-precision floating-point values in even/odd register pairs. There are 16 sets of general-purpose register pairs. Table 7–2 lists the pseudoregister name for each general-purpose register pair.

Table 7–2. Pseudoregister Names for Double-Precision Floating-Point Registers

Register Pair	Pseudoregister Name	Register Pair	Pseudoregister Name
A1:A0	DA0	B1:B0	DB0
A3:A2	DA2	B3:B2	DB2
A5:A4	DA4	B5:B4	DB4
A7:A6	DA6	B7:B6	DB6
A9:A8	DA8	B9:B8	DB8
A11:A10	DA10	B11:B10	DB10
A13:A12	DA12	B13:B12	DB12
A15:A14	DA14	B15:B14	DB14

You can display the contents of these registers by using the ? (evaluate expression) command or Setup→Watch Variable menu option.


- ☐ For example, if you want to evaluate the A11:A10 register pair, enter the ? command with the DA10 pseudoregister name:

?da10 

The debugger shows the result in the display area of the Command window:

```
?da10 1.0933371e-309
```

- ☐ To modify the A11:A10 register pair (DA10) and set it equal to 25.75, enter:

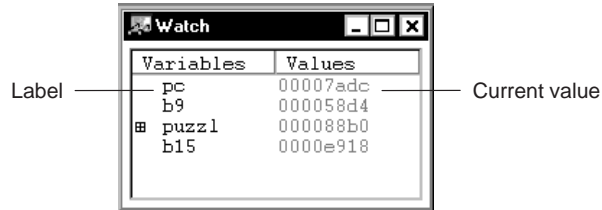
?da10 = 25.75 

The debugger displays the following in the display area of the Command window:

```
?da10 = 25.75 2.575000e+001
```

7.7 Managing Data in a Watch Window

The debugger does not maintain a dedicated window that tells you about the status of all the symbols defined in your program. Such a window might be so large that it would not be useful. Instead, the debugger allows you to open a Watch window that shows you how program execution affects specific expressions, variables, registers, or memory locations. You can choose which ones you want to observe. You can also use the Watch window to display members of complex, aggregate data types, such as arrays and structures.



The debugger displays a Watch window *only when you specifically request a Watch window*.

Remember, you can use the data-management commands or the debugger's overwrite editing capability to modify the contents of any value displayed in the Watch window. For more information, see section 7.4, *Basic Methods for Changing Data Values*, on page 7-5.

Displaying data in a Watch window

To display a value in the Watch window, follow these steps:

- 1) From the Setup menu, select Watch Variable. This displays the Watch Add dialog box:

The screenshot shows the 'Watch Add' dialog box with the following fields and annotations:

- Value Expression:** A text field containing 'csr'. Annotation: 'Enter the item that you want to watch'.
- Global variables:** A dropdown menu. Annotation: 'Select a global variable to watch'.
- Local variables:** A dropdown menu. Annotation: 'Select a local variable to watch'.
- Static variables:** A dropdown menu. Annotation: 'Select a static variable to watch'.
- Registers:** A dropdown menu. Annotation: 'Select a register to watch'.
- Format:** A dropdown menu showing 'x - Hexadecimal'. Annotation: 'Select a data format for the watched item (optional)'.
- Label:** A text field containing 'Control Status Register'. Annotation: 'Assign a label for the watched item (optional)'.
- Window name:** A dropdown menu. Annotation: 'Enter a name to open a new Watch window (optional)'.

Buttons at the bottom: OK, Apply, Close, Help.

- 2) In the Expression field, enter the item that you want to watch. The expression can be any C expression, including an expression that has side effects. Or, you can select a global variable, local variable, static variable, or register to watch.

If you want to use the *contents* of an address as a parameter, be sure to prefix the address with the C indirection operator (*). For example, you could enter this value in the Expression field:

```
*0x26
```

- 3) If you want to change the data format for the watched item, select the format you want to use from the Format drop list. The format field is optional.
- 4) If you want to assign a label for the watched item, use the Label field. If you leave the Label field blank, the debugger displays the expression, variable, or register as the label.
- 5) If you want to open a new Watch window, enter a name for the new Watch window in the Window name field. This field is optional. When you enter

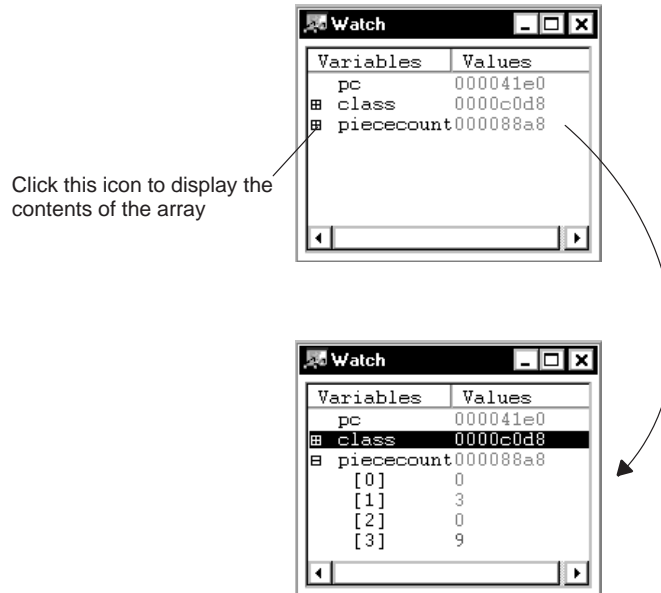
a window name, the debugger appends the window name to the Watch window label. If you do not supply a name, the debugger adds the item to the default Watch window.

- 6) Click Apply. When you have entered the last expression, variable, or register that you want to watch, click OK.

After you open a Watch window, executing Setup→Watch Variable and using the same window name adds additional values to the Watch window. You can open as many Watch windows as you need by using unique window names.

Displaying additional data

When you use the Watch window to view structures, pointers, or arrays, you can display the additional data (the data pointed to or the members of the array or structure) by clicking the box icon next to the watched item:



You can also display additional data by selecting an item and pressing **(SPACE)**.

Deleting watched values

To delete an entry from a Watch window, follow these steps:

- 1) Select the item in the Watch window that you want to delete.
- 2) Press **(DELETE)**.

If you want to close a Watch window and delete all of the items in that window in a single step, use the WR (watch reset) command. The syntax is:

wr [{ * | *window name* }]

The optional *window name* parameter deletes a particular Watch window;
* deletes all Watch windows.

Note:

The debugger automatically closes any Watch windows when you execute File→Load Program, File→Load Symbols, the LOAD command, or the SLOAD command.

7.8 Displaying Data in Alternative Formats

By default, all data is displayed in its natural format. This means that:

- ☐ Integer values are displayed as decimal numbers.
- ☐ Floating-point values are displayed in floating-point format.
- ☐ Pointers are displayed as hexadecimal addresses (with a 0x prefix).
- ☐ Enumerated types are displayed symbolically.

However, any data displayed in the Command, Memory, or Watch window can be displayed in a variety of formats.

Changing the default format for specific data types

To display specific types of data in a different format, use the SETF command. The syntax for this command is:

setf [*data type*, *display format*]

The *display format* parameter identifies the new display format for any data of type *data type*. Table 7–3 lists the available formats and the corresponding characters that can be used as the *display format* parameter.

Table 7–3. Display Formats for Debugger Data

Display Format	Parameter	Display Format	Parameter
Default for the data type	*	Octal	o
ASCII character (bytes)	c	Valid address	p
Decimal	d	ASCII string	s
Exponential floating point	e	Unsigned decimal	u
Decimal floating point	f	Hexadecimal	x

Table 7–4 lists the C data types that can be used for the *data type* parameter. Only a subset of the display formats applies to each data type, so Table 7–4 also shows valid combinations of data types and display formats.

Table 7–4. Data Types for Displaying Debugger Data

Data Type	Valid Display Formats										Default Display Format
	c	d	o	x	e	f	p	s	u		
char	✓	✓	✓	✓						✓	ASCII (c)
uchar	✓	✓	✓	✓						✓	Decimal (d)
short	✓	✓	✓	✓						✓	Decimal (d)
int	✓	✓	✓	✓						✓	Decimal (d)
uint	✓	✓	✓	✓						✓	Decimal (d)
long	✓	✓	✓	✓						✓	Decimal (d)
ulong	✓	✓	✓	✓						✓	Decimal (d)
float				✓	✓	✓	✓				Exponential floating point (e)
double				✓	✓	✓	✓				Exponential floating point (e)
ptr				✓	✓			✓	✓		Address (p)

Here are some examples:


- ☐ To display all data of type short as an unsigned decimal, enter:

```
setf short, u 
```

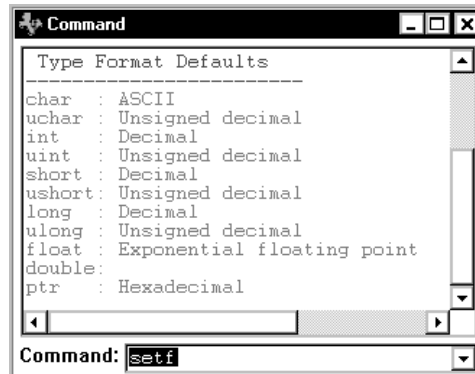
- ☐ To return all data of type short to its default display format, enter:

```
setf short, * 
```

- ☐ To list the current display formats for each data type, enter the SETF command with no parameters:

```
setf 
```

The display should look something like this:



- ☐ To reset all data types back to their default display formats, enter:

`setf *`

Changing the default format with data-management commands

You can also use the Setup→Watch Variable menu option, the Watch window context menu, and the ?, MEM, WA, and DISP commands to show data in alternative display formats. (The ? and DISP commands use alternative formats only for scalar types, arrays of scalar types, and individual members of aggregate types.)

Each of these commands has an optional display format parameter that works in the same way as the display format parameter of the SETF command.

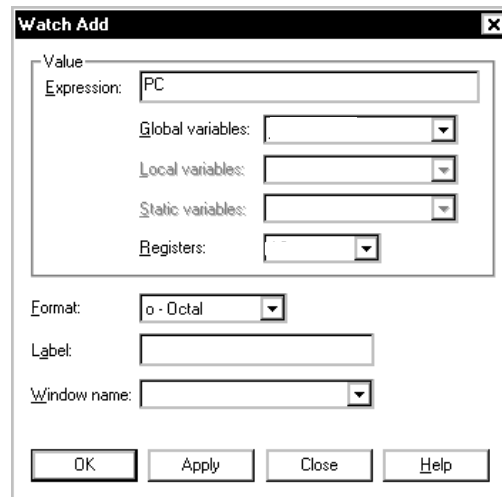
When you do not use a display format parameter, data is shown in its natural format (unless you have changed the format for the data type with SETF).

Here are some examples:

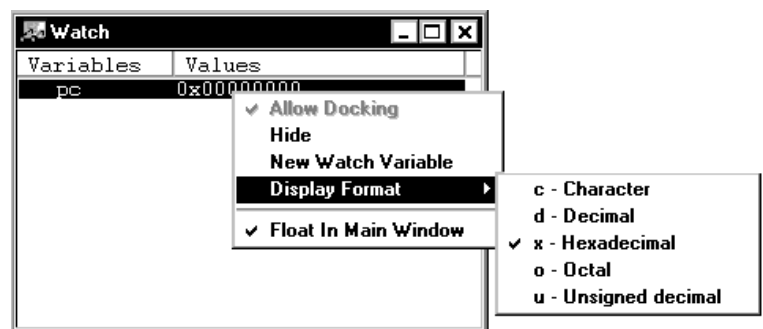
- ☐ To display memory contents in octal, enter:

`mem 0x0,o`

- ☐ To watch the PC in octal, enter:



- ☐ To change the format of the PC in the Watch window:
- 1) In the Watch window, select PC.
 - 2) Right click the mouse to bring up the Watch window context menu.
 - 3) From the context menu, select Display Format. A submenu of data formats appears.
 - 4) From the submenu, select the format in which you want the PC to display.



The valid combinations of data types and display formats listed for SETF also apply to the data displayed with `?`, `MEM`, `Setup→Watch Variable`, `WA`, and `DISP`. For example, if you want to use display format `e` or `f`, the data that you are displaying must be of type float or type double. Additionally, you cannot use the `s` display format parameter with the `MEM` command.

Profiling Code Execution

The profiling environment is a special debugger environment that provides a method for collecting execution statistics about specific areas in your code. These statistics give you immediate feedback on your application's performance.

Topic	Page
8.1 Overview of the Profiling Environment	8-2
8.2 Overview of the Profiling Process	8-3
8.3 Entering the Profiling Environment	8-4
8.4 Defining Areas for Profiling	8-5
8.5 Defining a Stopping Point	8-15
8.6 Running a Profiling Session	8-17
8.7 Viewing Profile Data	8-20
8.8 Saving Profile Data to a File	8-27

8.1 Overview of the Profiling Environment

The profiling environment builds on the same intuitive interface available in the basic debugging environment and has these additional features:

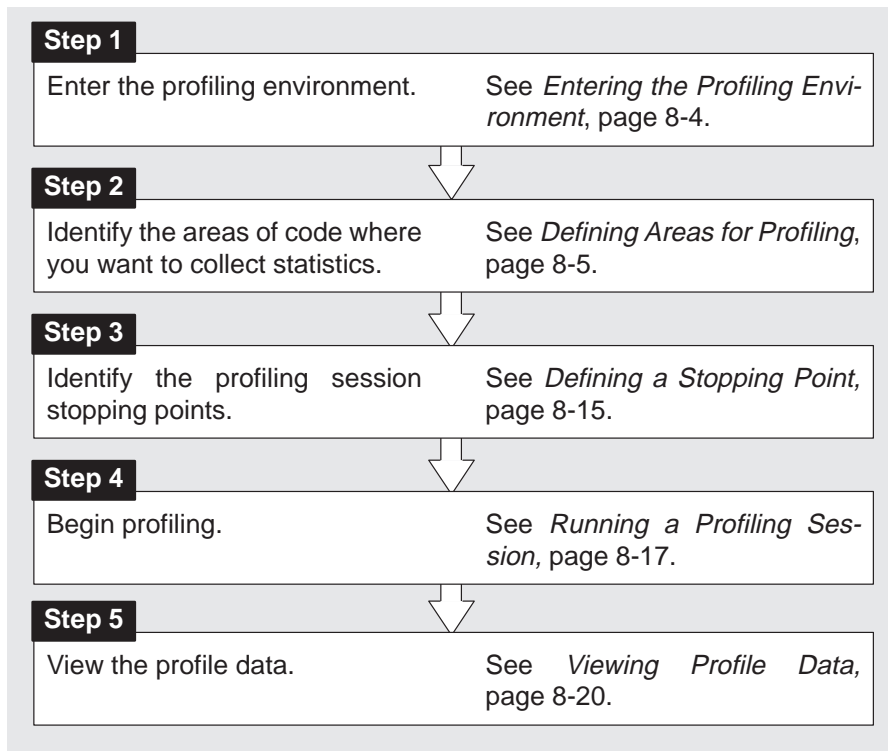
- ❑ **More efficient code.** Within the profiling environment, you can quickly identify busy sections in your programs. This helps you to direct valuable development time toward optimizing the sections of code that most dramatically affect program performance.
- ❑ **Statistics on multiple areas.** You can collect statistics about individual statements in disassembly or C, about ranges in disassembly or C, and about C functions. When you are collecting statistics on many areas, you can choose to view the statistics for all the areas or a subset of the areas.
- ❑ **Comprehensive display of statistics.** The profiler provides all the information you need for identifying bottlenecks in your code:
 - The number of times each area was entered during the profiling session
 - The total execution time of an area, including or excluding the execution time of any subroutines called from within the area
 - The maximum time for one iteration of an area, including or excluding the execution time of any subroutines called from within the area

Statistics may be updated continuously during the profiling session or at selected intervals.

- ❑ **Configurable display of statistics.** Display the entire set of data, or display one type of data at a time. Display all the areas you are profiling, or display a selected subset of the areas.
- ❑ **Visual representation of statistics.** When you choose to display one type of data at a time, the statistics are accompanied by histograms for each area, showing the relationship of each area's statistics to those of the other profiled areas.
- ❑ **Disabled areas.** In addition to identifying areas that you can collect statistics on, you can also identify areas that you do not want to affect the statistics. This removes the timing impact from code such as a standard library function or a fully optimized portion of code.

8.2 Overview of the Profiling Process

Profiling consists of five simple steps:



Note:

When you compile a program that will be profiled, you must use the `-g` and the `-as` compiler shell options. The `-g` option includes symbolic debugging information; the `-as` option ensures that you will be able to include ranges as profile areas. For more information on these options, see the TMS320C6x C Compiler User's Guide.

A profiling strategy

Here is a suggestion for a basic approach to profiling the performance of your program.

- 1) Mark all the functions in your program as profile areas.
- 2) Run a profiling session; find the busiest functions.
- 3) Unmark all the functions.
- 4) Mark the individual lines in the busy functions and run another profiling session.

8.3 Entering the Profiling Environment

To enter the profiling environment, select Profile Mode from the Profile menu.

Some restrictions apply to the profiling environment:

- ☐ The debugger is always in mixed mode.
- ☐ Command, Disassembly, File, and Profile are the only windows available; additional windows, such as a Watch window, cannot be opened.
- ☐ The profiling environment supports only a subset of the debugger commands. Table 8–1 lists the debugger commands that can and cannot be used in the profiling environment.

Table 8–1. Debugger Commands That Can/Cannot Be Used in the Profiling Environment

Can be used	Cannot be used
Data-evaluation commands (such as ? and EVAL)	All run commands
Breakpoint commands	Debugging mode commands (such as ASM, C, and MIX)
Memory-mapping commands	Commands related to the Watch, Memory, or Calls window
System commands (such as SYSTEM, TAKE, and ALIAS)	
Windowing commands (such as SIZE, MOVE, and ZOOM)	

Chapter 12, *Summary of Commands*, summarizes all of the debugger commands and tells you whether a command is valid in the profiling environment.

8.4 Defining Areas for Profiling

Within the profiling environment, you can collect statistics on three types of areas:

- ☐ *Individual lines* in C or disassembly
- ☐ *Ranges* in C or disassembly
- ☐ *Functions* in C only

To identify any of these areas for profiling, mark the line, range, or function. You can disable areas so that they do not affect the profile data, and you can reen-able areas that have been disabled. You can also unmark areas that you are no longer interested in.

Using the mouse is the simplest way to mark, disable, enable, and unmark areas. A dialog box also supports these and more complex tasks.

The following subsections explain how to mark, disable, reen-able, and unmark profile areas by using the mouse or the dialog box. For restrictions on profiling areas, see page 8-12.

Marking an area with a mouse

Marking an area qualifies it for profiling so that the debugger can collect timing statistics about the area.

Remember, to display C code, use the File→Open menu option or the FUNC command; to display disassembly, use the DASM command.

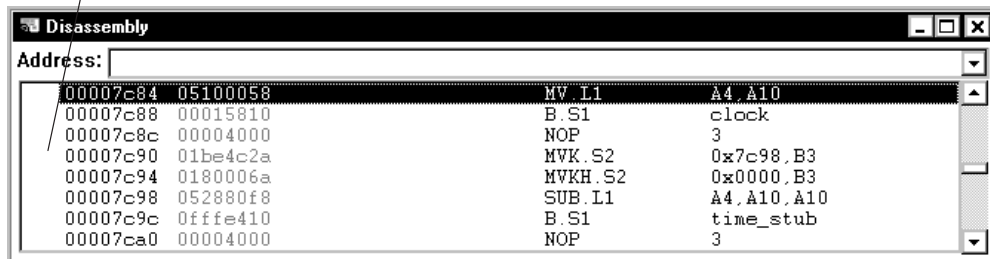
Notes:

- 1) Marking an area in C *does not* mark the associated code in disassembly.
 - 2) Areas can be nested; for example, you can mark a line within a marked range. The debugger reports statistics for both the line and the function.
 - 3) Ranges cannot overlap, and they cannot span function boundaries.
-

To mark an area with the mouse, follow these steps:

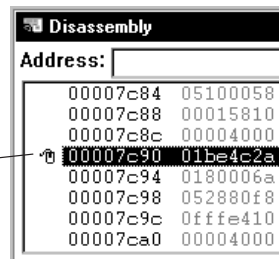
- 1) In the File or Disassembly window, click once to the left of the line that you want to mark or to the left of the first line of the range that you want to mark:

Click next to the line that you want to mark



When you click once next to a line, a mouse icon appears, telling you that you need to click one more time:

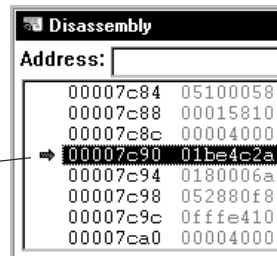
A mouse icon tells you that you need to click one more time



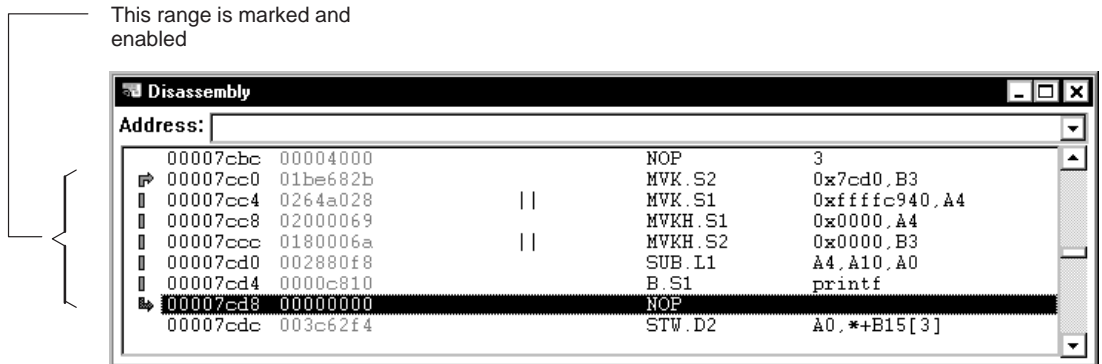
- 2) Choose to mark a single line or a range:

- ☐ To mark a single line, click the mouse icon. This turns the mouse icon into a green right arrow:

A green right arrow tells you that this line is marked and enabled



- ❑ To mark a range, click the last line of the range that you want to mark. This changes the mouse icon on the first line of the range into a green arrow. The entire range is marked with two green right arrows that are connected:



You can also use the mouse to mark a function in C code. To do so, follow these steps:

- 1) In the File window, click next to the statement that declares the function that you want to mark.
- 2) When you see the mouse icon, click again to mark and enable the C function. A green arrow appears, indicating that the function is marked.

Note:

In Profile mode, if you try to mark a line or function by double-clicking next to the statement that you want to mark, the debugger sets a software breakpoint instead of marking the line or function. To mark a function, click once. If you are marking a line and you see the mouse icon, click again.

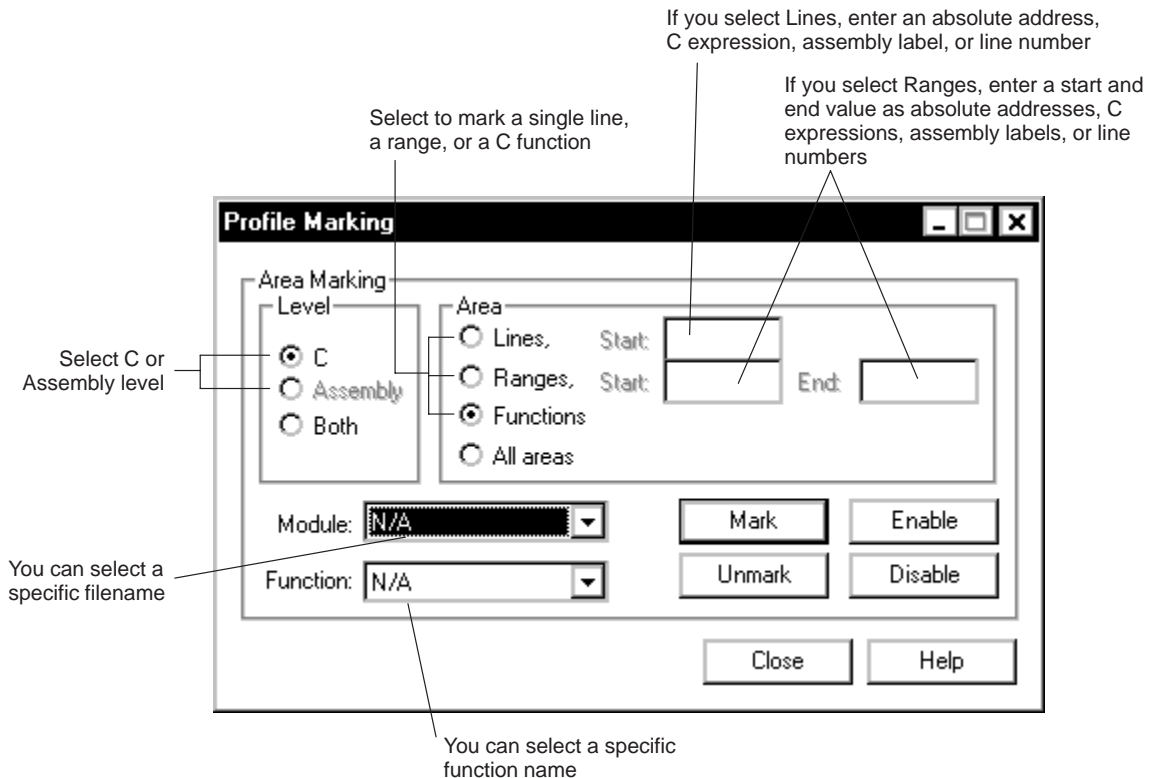
If you are not in profile mode, single-clicking next to a line or function sets a software breakpoint.

Marking an area with a dialog box

You can use a dialog box to mark areas for profiling. To do so, follow these steps:

- 1) Open the Profile Marking dialog box by using one of these methods:
 - ☐ From the Profile menu, select Select Areas.
 - ☐ From the context menu for the Profile window, select Select Areas.

This displays the Profile Marking dialog box:



- 2) In the Level box, select C or Assembly.
- 3) In the Area box, select Lines, Ranges, or Functions. See Table 8–2 for a list of valid combinations.

- 4) Depending on what you select in step 3, do one or more of the following:
- ☐ Next to Lines, enter an absolute address, C expression, assembly label, or line number. If you are entering an absolute address, be sure to prefix it with 0x.
 - ☐ Next to Ranges, enter a Start and an End value as absolute addresses, C expressions, assembly labels, or line numbers.
 - ☐ From the Module combo box, select a specific filename.
 - ☐ From the Function combo box, select a specific function name.
- See Table 8–2 for a list of valid combinations.
- 5) Click Mark.
- 6) Click Close to close the dialog box.

Table 8–2. Using the Profile Marking Dialog Box to Mark Areas

(a) Marking lines

To mark this area...	If C level is selected...	If Assembly level is selected...
By line number, address	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select Lines. <input type="checkbox"/> Next to Lines, specify a line number.	<input type="checkbox"/> In the Area box, select Lines. <input type="checkbox"/> Next to Lines, specify an absolute address, a C expression, or an assembly label
All lines in a function	<input type="checkbox"/> Select a function name. <input type="checkbox"/> In the Area box, select Lines.	<input type="checkbox"/> Select a function name. <input type="checkbox"/> In the Area box, select Lines.

(b) Marking ranges

To mark this area...	If C level is selected...	If Assembly level is selected...
By line numbers, addresses	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select Ranges. <input type="checkbox"/> Next to Ranges, specify a Start line number and an End line number.	<input type="checkbox"/> In the Area box, select Ranges. <input type="checkbox"/> Next to Ranges, specify a Start and an End value. Use an absolute address, a C expression, or an assembly label for each.

(c) Marking functions

To mark this area...	If C level is selected...	If Assembly level is selected...
By function name	<input type="checkbox"/> Select a function name. <input type="checkbox"/> In the Area box, select Functions.	Not applicable
All functions in a module	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select Functions.	Not applicable
All functions everywhere	<input type="checkbox"/> In the Area box, select Functions. <input type="checkbox"/> Be sure that Function and Module are set to N/A.	Not applicable

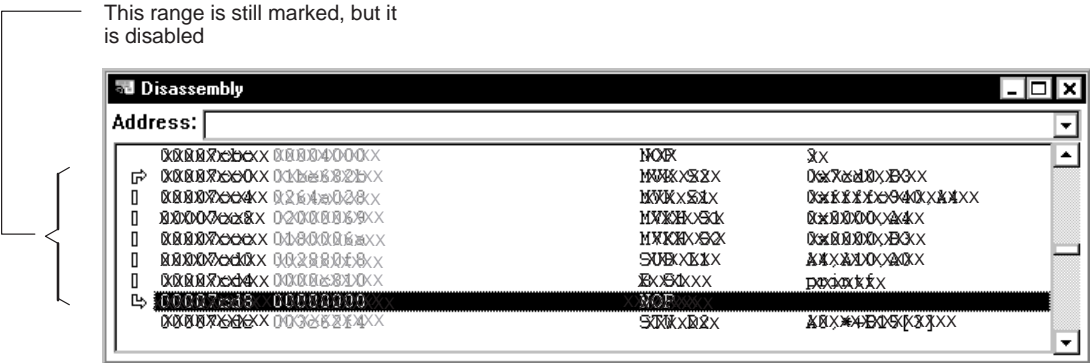
Disabling an area

At times, it is useful to identify areas that you do not want affecting profile statistics. To do this, *disable* the appropriate area. Disabling effectively subtracts the timing information of the disabled area from all profile areas that include or call the disabled area. Areas must be marked before they can be disabled.

For example, if you have marked a function that calls a standard C function such as malloc(), you may not want malloc() to affect the statistics for the calling function. You could mark the line that calls malloc(), and then disable the line. This way, the profile statistics for the function would not include the statistics for malloc().

Note:
If you disable an area after you have already collected statistics on it, that information will be lost.

The easiest way to disable an area is to click the green arrow(s) next to a marked line, range, or function. When you do so, the arrow(s) becomes white:



- You can also disable an area by using the Profile Marking dialog box:
- 1) Open the Profile Marking dialog box by using one of these methods:
 - ☐ From the Profile menu, select Select Areas.
 - ☐ From the context menu for the Profile window, select Select Areas.This displays the Profile Marking dialog box.
 - 2) In the Level box, select C, Assembly, or Both.
 - 3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 8–3 on page 8-13 for a list of valid combinations.

- 4) Depending on what you select in step 3, do one or more of the following:
 - ☐ Next to Lines, enter an absolute address, C expression, assembly label, or line number.
 - ☐ Next to Ranges, enter a Start and an End value as absolute addresses, C expressions, assembly labels, or line numbers.
 - ☐ From the Module combo box, select a specific filename.
 - ☐ From the Function combo box, select a specific function name.

See Table 8–3 for a list of valid combinations.
- 5) Click Disable.
- 6) Click Close to close the dialog box.

Reenabling a disabled area

When an area has been disabled and you would like to profile it once again, you must enable the area. To reenable an area, click the white arrow(s) next to marked line, range, or function; the area will once again be highlighted with a green arrow.

You can also reenable an area by using the Profile Marking dialog box:

- 1) Open the Profile Marking dialog box by using one of these methods:
 - ☐ From the Profile menu, select Select Areas.
 - ☐ From the context menu for the Profile window, select Select Areas.

This displays the Profile Marking dialog box.
- 2) In the Level box, select C, Assembly, or Both.
- 3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 8–3 for a list of valid combinations.
- 4) Depending on what you select in step 3, do one or more of the following:
 - ☐ Next to Lines, enter an absolute address, C expression, assembly label, or line number.
 - ☐ Next to Ranges, enter a Start and an End value as an absolute address, C expression, assembly label, or line number.
 - ☐ From the Module combo box, select a specific filename.
 - ☐ From the Function combo box, select a specific function name.

See Table 8–3 for a list of valid combinations.
- 5) Click Enable.
- 6) Click Close to close the dialog box.

Unmarking an area

If you want to stop collecting information about a specific area, unmark it.

The easiest way to unmark an area is to double-click the green or white arrow(s) next to marked line, range, or function. This unmarks the line, range, or function.

You can also unmark an area by using the Profile Marking dialog box:

- 1) Open the Profile Marking dialog box by using one of these methods:
 - ☐ From the Profile menu, select Select Areas.
 - ☐ From the context menu for the Profile window, select Select Areas.
- 2) In the Level box, select C, Assembly, or Both.
- 3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 8–3 for a list of valid combinations.
- 4) Depending on what you select in step 3, do one or more of the following:
 - ☐ Next to Lines, enter an absolute address, C expression, assembly label, or line number.
 - ☐ Next to Ranges, enter a Start and an End value as absolute addresses, C expressions, assembly labels, or line numbers.
 - ☐ From the Module combo box, select a specific filename.
 - ☐ From the Function combo box, select a specific function name.See Table 8–3 for a list of valid combinations.
- 5) Click Unmark.
- 6) Click Close to close the dialog box.

Restrictions on profiling areas

The following restrictions apply to profiling areas:

- ☐ An area cannot begin or end in the delay slot of a load instruction (emulator only).
- ☐ An area cannot begin in the delay slot of a branch instruction.
- ☐ An area can end in the last delay slot of a branch instruction but cannot end in any other delay slot of a branch instruction.

Table 8–3. Disabling, Enabling, Unmarking, or Viewing Areas

(a) *Disabling, enabling, unmarking, or viewing lines*

To identify this area...	If the C level is selected...	If the Assembly level is selected...	If the Both level is selected...
By line number, address†	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select Lines. <input type="checkbox"/> Next to Lines, specify a line number.	<input type="checkbox"/> In the Area box, select Lines. <input type="checkbox"/> Next to Lines, specify an absolute address, a C expression, or an assembly label.	Not applicable
All lines in a function	<input type="checkbox"/> Select a function name. <input type="checkbox"/> In the Area box, select Lines.	<input type="checkbox"/> Select a function name. <input type="checkbox"/> In the Area box, select Lines.	<input type="checkbox"/> Select a function name. <input type="checkbox"/> In the Area box, select Lines.
All lines in a module	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select Lines.	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select Lines.	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select Lines.
All lines everywhere	<input type="checkbox"/> In the Area box, select Lines. <input type="checkbox"/> Be sure that Function and Module are set to N/A.	<input type="checkbox"/> In the Area box, select Lines. <input type="checkbox"/> Be sure that Function and Module are set to N/A.	<input type="checkbox"/> In the Area box, select Lines. <input type="checkbox"/> Be sure that Function and Module are set to N/A.

† You cannot specify line numbers or addresses when using the Profile View dialog box.

(b) *Disabling, enabling, unmarking, or viewing ranges*

To identify this area...	If the C level is selected...	If the Assembly level is selected...	If the Both level is selected...
By line numbers, addresses†	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select Ranges. <input type="checkbox"/> Next to Ranges, specify a Start line number and an End line number.	<input type="checkbox"/> In the Area box, select Ranges. <input type="checkbox"/> Next to Ranges, specify a Start and an End value as absolute addresses, C expressions, or assembly labels.	Not applicable
All ranges in a function	<input type="checkbox"/> Select a function name. <input type="checkbox"/> In the Area box, select Ranges.	<input type="checkbox"/> Select a function name. <input type="checkbox"/> In the Area box, select Ranges.	<input type="checkbox"/> Select a function name. <input type="checkbox"/> In the Area box, select Ranges.
All ranges in a module	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select Ranges.	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select Ranges.	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select Ranges.
All ranges everywhere	<input type="checkbox"/> In the Area box, select Ranges. <input type="checkbox"/> Be sure that Function and Module are set to N/A.	<input type="checkbox"/> In the Area box, select Ranges. <input type="checkbox"/> Be sure that Function and Module are set to N/A.	<input type="checkbox"/> In the Area box, select Ranges. <input type="checkbox"/> Be sure that Function and Module are set to N/A.

† You cannot specify line numbers or addresses when using the Profile View dialog box.

Table 8–3. Disabling, Enabling, Unmarking, or Viewing Areas (Continued)

(c) Disabling, enabling, unmarking, or viewing functions

To identify this area...	If the C level is selected...	If the Assembly level is selected...	If the Both level is selected...
By function name	<input type="checkbox"/> Select a function name. <input type="checkbox"/> In the Area box, select Functions.	Not applicable	Not applicable
All functions in a module	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select Functions.	Not applicable	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select Functions.
All functions everywhere	<input type="checkbox"/> In the Area box, select Functions. <input type="checkbox"/> Be sure that Function and Module are set to N/A.	Not applicable	<input type="checkbox"/> In the Area box, select Functions. <input type="checkbox"/> Be sure that Function and Module are set to N/A.

(d) Disabling, enabling, unmarking, or viewing all areas

To identify this area...	If the C level is selected	If the Assembly level is selected	If the Both level is selected
All areas in a function	<input type="checkbox"/> Select a function name. <input type="checkbox"/> In the Area box, select All areas.	<input type="checkbox"/> Select a function name. <input type="checkbox"/> In the Area box, select All areas.	<input type="checkbox"/> Select a function name. <input type="checkbox"/> In the Area box, select All areas.
All areas in a module	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select All areas.	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select All areas.	<input type="checkbox"/> Select a module name. <input type="checkbox"/> In the Area box, select All areas.
All areas everywhere	<input type="checkbox"/> In the Area box, select All areas. <input type="checkbox"/> Be sure that Function and Module are set to N/A.	<input type="checkbox"/> In the Area box, select All areas. <input type="checkbox"/> Be sure that Function and Module are set to N/A.	<input type="checkbox"/> In the Area box, select All areas. <input type="checkbox"/> Be sure that Function and Module are set to N/A.

8.5 Defining a Stopping Point

Before you run a profiling session, you must identify the point where the debugger should stop collecting statistics. By default, C programs contain an *exit* label, and this is defined as the default stopping point when you load your program. (You can delete *exit* as a stopping point, if you choose.) If your program does not contain an *exit* label, or if you prefer to stop at a different point, you can use a software breakpoint to define another stopping point. You can set multiple breakpoints; the debugger stops at the first one it finds.

Even though no statistics can be gathered for areas following a breakpoint, the areas will be listed in the Profile window.

Note:

You cannot set a software breakpoint on a statement that has already been defined as a part of a profile area.

Setting and clearing a software breakpoint in the profiling environment is similar to setting and clearing a software breakpoint in the basic debugging environment. For more information about setting and clearing software breakpoints, see section 6.7 on page 6-14.

Setting a software breakpoint

To set a breakpoint, *double-click* next to the statement in the Disassembly or File window where you want the breakpoint to occur.

You can also set a breakpoint using the Breakpoint Control dialog box:

- 1) Open the Breakpoint Control dialog box by using one of these methods:

- ☐ Click the Breakpoint Dialog icon on the toolbar:



- ☐ From the Setup menu, select Breakpoints.

- 2) In the Address field of the Breakpoint Control dialog box, enter an absolute address, any C expression, the name of a C function, or an assembly language label.
- 3) Click Add. The new breakpoint appears in the breakpoint list.
- 4) Click Close to close the Breakpoint Control dialog box.

Clearing a software breakpoint

To clear a breakpoint, *double-click* the breakpoint symbol (●) in the File or Disassembly window.

You can also clear a breakpoint by using the Breakpoint Control dialog box:

- 1) Open the Breakpoint Control dialog box by using one of these methods:

- ☐ Click the Breakpoint Dialog icon on the toolbar:



- ☐ From the Setup menu, select Breakpoints.

- 2) Select the address of the breakpoint that you want to clear.
- 3) Click Delete. The breakpoint is removed from the breakpoint list.
- 4) Click Close to close the Breakpoint Control dialog box.

8.6 Running a Profiling Session

Once you have defined profile areas and a stopping point, you can run a profiling session. You can run two types of profiling sessions:

- ☐ A *full profile* collects a full set of statistics for the defined profile areas.
- ☐ A *quick profile* collects a subset of the available statistics (it does not collect exclusive or exclusive max data, which are described in section 8.7 on page 8-20). This reduces overhead because the debugger does not have to track entering/exiting subroutines within an area.

Running a full or a quick profiling session

To run a profiling session, follow these steps:

- 1) Open the Profile Run dialog box by using one of these methods:

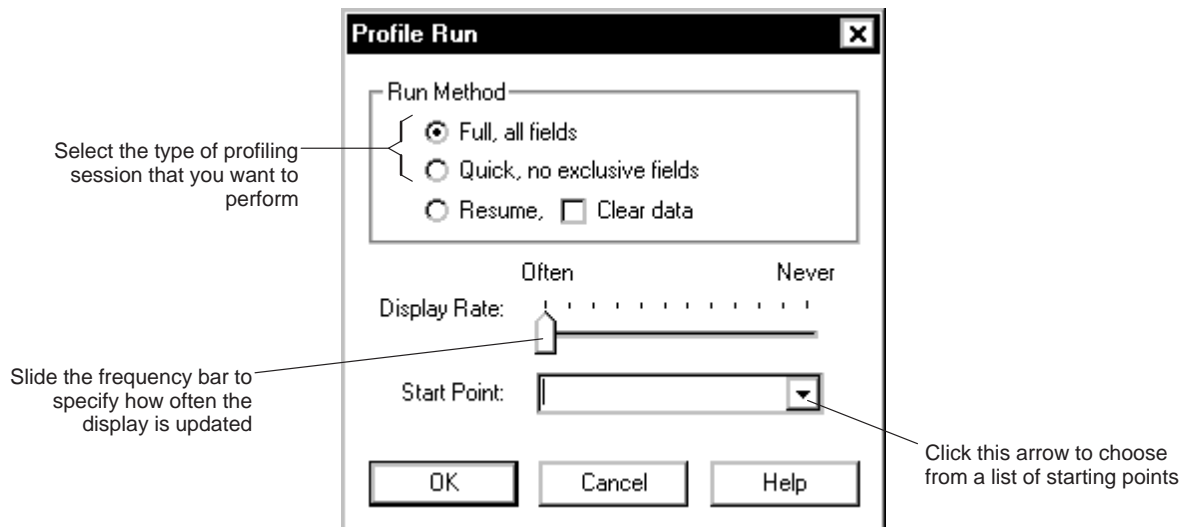
- ☐ Click the Run icon on the toolbar:



- ☐ From the Profile menu, select Run.

- ☐ Press (F5) .

This displays the Profile Run dialog box:



- 2) In the Run Method box, select the type of profiling session that you want to perform: Full or Quick.

- 3) Slide the Display Rate frequency bar to specify how often the display is updated.

You can choose a Display Rate from Often to Never. A Display Rate of Never causes the profiler to display profiling information only when the profiling session is complete.

- 4) In the Start Point field, enter the starting point for the profiling session. The starting point can be a label, a function name, or a memory address. If you specify a memory address, be sure to prefix the address with **0x**.

You can choose from a list of starting points by clicking on the arrow at the end of the Start Point field.

- 5) Click OK.

After you click OK, your program **restarts** and **runs to the defined starting point**. You can tell that the debugger is profiling because the status bar changes to *Target: Profiling*, as shown here.



Profiling begins when the starting point is reached and continues until a stopping point is reached or until you halt the profiling session by doing one of the following:

- ☐ Click the Halt icon on the toolbar:



- ☐ From the Target menu, select Halt!.

- ☐ Press **(ESC)**.

Resuming a profiling session that has halted

To resume a profiling session that has halted, follow these steps:

- 1) Open the Profile Run dialog box by using one of these methods:

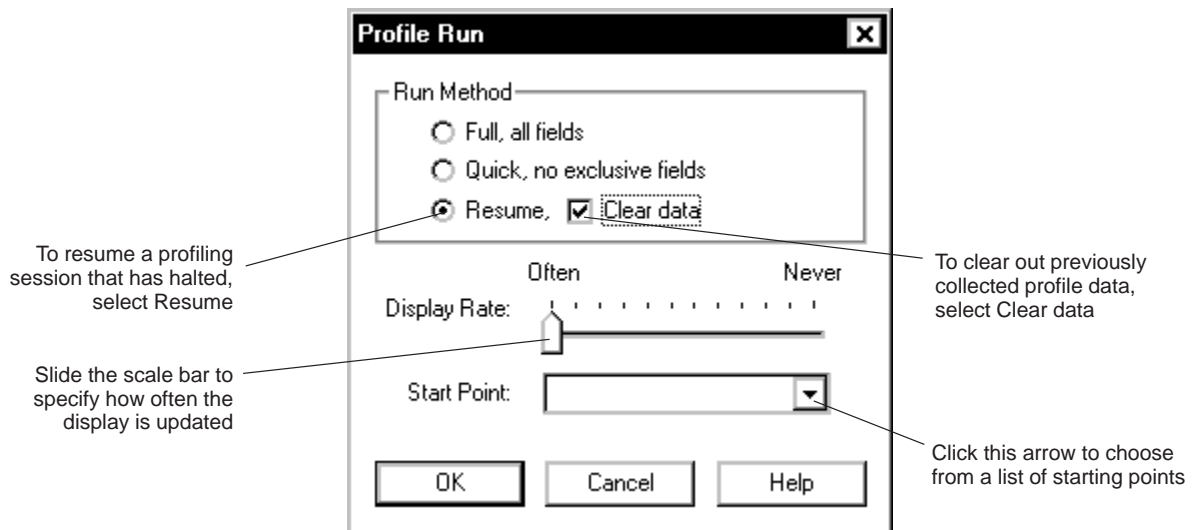
- ☐ Click the Run icon on the toolbar:



- ☐ From the Profile menu, select Run.

- ☐ Press **(F5)**.

This displays the Profile Run dialog box:




- 2) In the Run Method box, select Resume.
- 3) If you want to clear out the previously collected data, select Clear data in the Run Method box.
- 4) Slide the Display Rate scale to specify how often the display is updated. You can choose a Display Rate from Often to Never. A Display Rate of Never causes the profiler to display profiling information only when the profiling session is complete.
- 5) In the Start Point field, enter the starting point for the profiling session. The starting point can be a label, a function name, or a memory address. If you specify a memory address, be sure to prefix the address with **0x**. You can choose from a list of starting points by clicking on the arrow at the end of the Start Point field.
- 6) Click OK.

8.7 Viewing Profile Data

The statistics collected during a profiling session are displayed in the Profile window. Figure 8–1 shows an example of this window.

Figure 8–1. An Example of the Profile Window

Type	Area Name	Count	Inclusive	Incl-Max	Exclusive	Excl-Max
C Function	f1()	4	8638	8638	105	32
C Function	f2()	4	13980	8384	116	32
C Function	f3()	4	13980	8384	116	32
C Function	main()	1	26547	26547	36	36



Profile areas

Profile data

The example in Figure 8–1 shows the Profile window with some default conditions:

- ☐ Column headings show the labels for the default set of profile data, including *Count*, *Inclusive*, *Incl-Max*, *Exclusive*, and *Excl-Max*.
- ☐ The data is sorted on the address of the first line in each area.
- ☐ All marked areas are listed, including disabled areas.

You can modify the Profile window to display selected profile areas or different data; you can also sort the data differently. The following subsections explain how to do these things.

Viewing different profile data

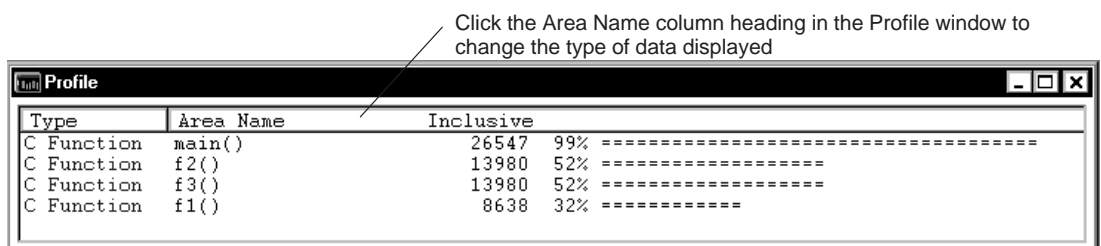
By default, the Profile window shows a set of statistics labeled as Count, Inclusive, Incl-Max, Exclusive, and Excl-Max. The Address field, which is not part of the default statistics, can also be displayed. Table 8–4 describes the statistic that each field represents.

Table 8–4. Types of Data Shown in the Profile Window

Label	Profile Data
Count	The number of times a profile area is entered during a session
Inclusive	The total execution time (cycle count) of a profile area, including the execution time of any subroutines called from within the profile area
Incl-Max (inclusive maximum)	The maximum inclusive time for one iteration of a profile area
Exclusive	The total execution time (cycle count) of a profile area, excluding the execution time of any subroutines called from within the profile area In general, the exclusive data provides the best statistics for comparing the execution time of one profile area to another area.
Excl-Max (exclusive maximum)	The maximum exclusive time for one iteration of a profile area
Address	The memory address of the line. If the area is a function or range, the Address field shows the memory address of the first line in the area.

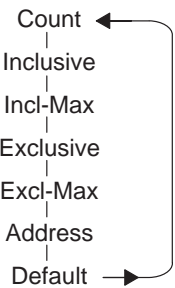
In addition to viewing this data in the default manner, you can view each of these statistics individually. The benefit of viewing them individually is that in addition to a cycle count, you are also supplied with a percentage indication and a histogram.

To view the fields individually, click the Area Name column heading in the Profile window.



When you click the Area Name column heading in the Profile window, fields are displayed individually in the order shown in Figure 8–2.

Figure 8–2. Cycling Through the Profile Window Fields



Note: Exclusive and Excl-Max are shown only when you run a full profile.

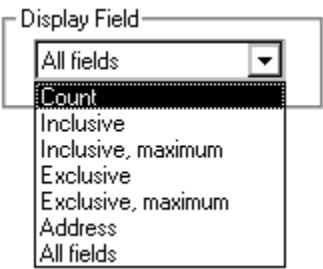
One advantage of using the mouse is that you can change the display while you are profiling.

You can also use the Profile View dialog box to select the field you want to display. To do so, follow these steps:

- 1) Open the Profile View dialog box by using one of these methods:
 - ☐ From the Profile menu, select Change View.
 - ☐ From the context menu for the Profile window, select Change View.

This displays the Profile View dialog box.

- 2) In the Display Field box, select the data field that you want to display:



- 3) Click OK.

Sorting profile data

By default, the data displayed in the Profile window is sorted according to the memory addresses of the displayed areas. The area with the least significant address is listed first, followed by the area with the next least significant address, etc. When you view fields individually, the data is automatically sorted from highest cycle count to lowest (instead of by address).

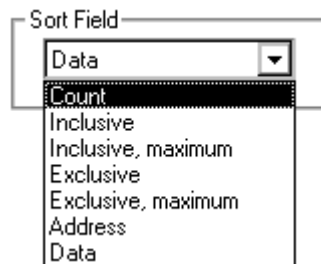
To sort the data on any of the data fields, follow these steps:

- 1) Open the Profile View dialog box by using one of these methods:

- ☐ From the Profile menu, select Change View.
- ☐ From the context menu for the Profile window, select Change View.

This displays the Profile View dialog box.

- 2) In the Sort Field box, select the data field that you want to sort on:



- 3) Click OK.

For example, to sort all the data on the basis of values of the Inclusive field, select Inclusive in the Sort Field box. The area with the highest Inclusive field displays first, and the area with the lowest Inclusive field displays last. This applies even when you are viewing individual fields.

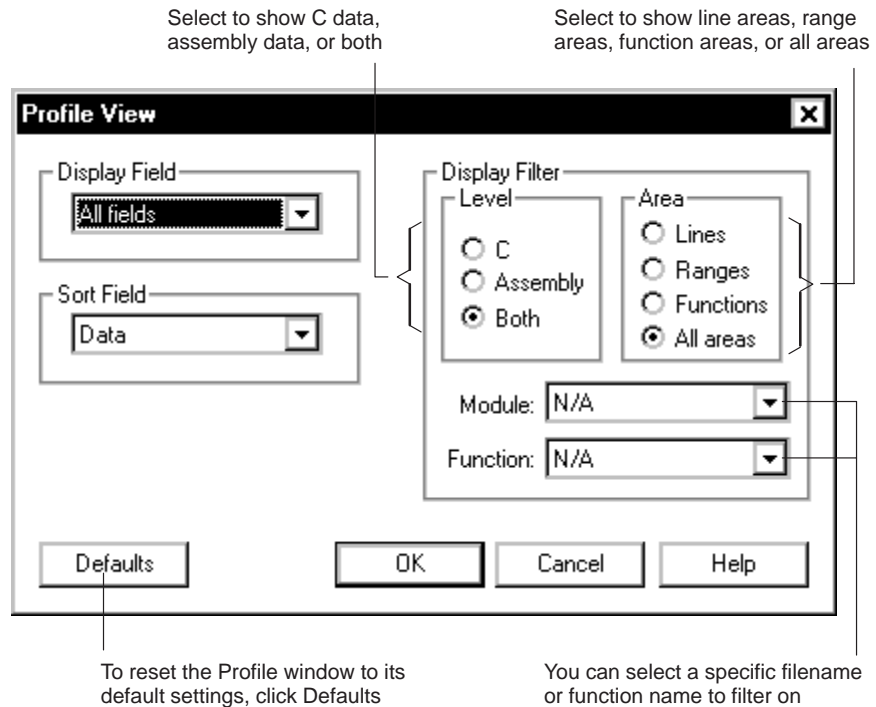
Viewing different profile areas

By default, all marked areas are listed in the Profile window. You can modify the window to display selected areas. To do this, follow these steps:

- 1) Open the Profile View dialog box by using one of these methods:

- ☐ From the Profile menu, select Change View.
- ☐ From the context menu for the Profile window, select Change View.

This displays the Profile View dialog box.



- 2) In the Level box, select C, Assembly, or Both.
- 3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 8–3 on page 8-13 on for a list of valid combinations.
- 4) If you want to view areas within a specific file or function, do one of the following:
 - ☐ From the Module combo box, select a specific filename.
 - ☐ From the Function combo box, select a specific function name.

See Table 8–3 on page 8-13 for a list of valid combinations.

- 5) Click OK.

If you want to reset the Profile window to its default characteristics, use the Profile View dialog box (Profile→Change View). Click the Defaults button, then click OK.

Interpreting session data

General information about a profiling session is displayed in the Command window during and after the session. This information identifies the starting and stopping points. It also lists statistics for three important areas:

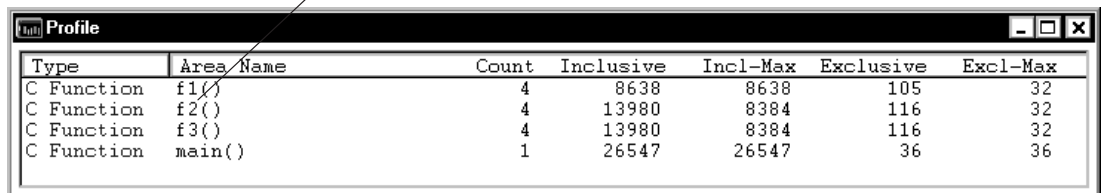
- ☐ *Run cycles* shows the number of execution cycles consumed by the program from the starting point to the stopping point.
- ☐ *Profile cycles* equals the run cycles minus the cycles consumed by disabled areas.
- ☐ *Hits* shows the number of internal breakpoints encountered during the profiling session.

Viewing code associated with a profile area

You can view the code associated with a displayed profile area. The debugger updates the display so that the associated C or disassembly statements are shown in the File or Disassembly window.

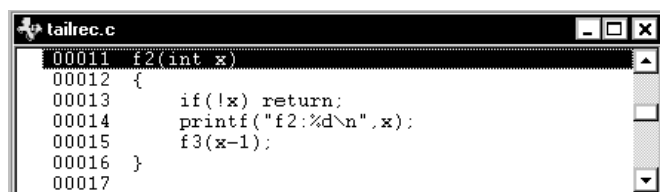
To select the profile area in the Profile window and display the associated code, double-click the area that you want to display:

Double-click an area to display the associated code



Type	Area Name	Count	Inclusive	Incl-Max	Exclusive	Excl-Max
C Function	f1()	4	8638	8638	105	32
C Function	f2()	4	13980	8384	116	32
C Function	f3()	4	13980	8384	116	32
C Function	main()	1	26547	26547	36	36

If the area is a function name, the debugger opens a File window and displays that function:



```

00011 f2(int x)
00012 {
00013     if(!x) return;
00014     printf("f2:%d\n",x);
00015     f3(x-1);
00016 }
00017

```

If the area is in disassembly code, the debugger displays that code in the Disassembly window.

To view the code associated with another area, double-click another area.

If you are attempting to show disassembly, you might need to make several attempts because you can access program memory only when the target is not running.

8.8 Saving Profile Data to a File

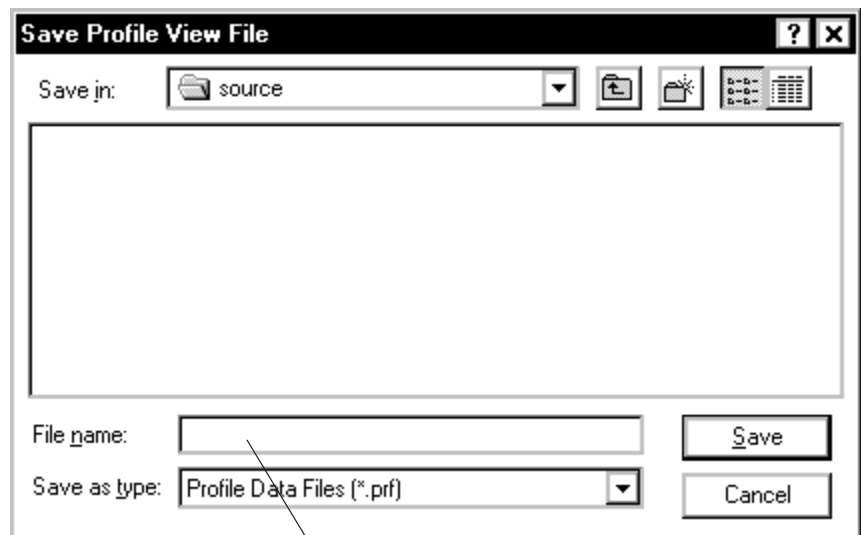
You may want to run several profiling sessions during a debugging session. Whenever you start a new profiling session, the results of the previous session are lost. However, you can save the results of the current profiling session to a system file.

The saved file contents are in ASCII and are formatted in exactly the same manner as they are displayed (or would be displayed) in the Profile window. The general profiling-session information that is displayed in the Command window is also written to the file.

Saving the contents of the Profile window

To save the contents of the Profile window to a system file, follow these steps:

- 1) From the Profile menu, select Save View. This displays the Save Profile View File dialog box:



Enter a name for the file. Use a .prf extension.

- 2) In the File name field, enter a name for the file. You can use a .prf extension to identify the file as a profile data file.
- 3) Click Save.

This saves only the current view; if, for example, you are viewing only the Count field, then only that information is saved. If the file already exists, debugger overwrites the file with the new data.

Saving all data for currently displayed areas

To save all data for the currently displayed areas, follow these steps:

- 1) From the Profile menu, select Save All. This displays the Save Profile File dialog box.
- 2) In the File name field, enter a name for the file. You can use a .prf extension to identify the file as a profile data file.
- 3) Click Save.

This saves all views of the data—including the individual count, inclusive, etc.—with the percentage indications and histograms. If the file already exists, debugger overwrites the file with the new data.

Using Simulator Memory System Analysis

The TMS320C6x simulator/debugger memory system analysis functionality allows you to measure your system performance accurately. You do this by monitoring system events using the Analysis Events dialog box and the Analysis Statistics window.

This chapter explains how to count various CPU events and set up breakpoints on events by using the Analysis menu and the Analysis Events dialog box.

Note:

Memory system analysis is not supported by the fast version of the fixed-point simulator or by the floating-point version of the simulator.

Topic	Page
9.1 Major Functions of Simulator Memory System Analysis	9-2
9.2 Overview of the Analysis Process	9-3
9.3 Enabling Memory System Analysis	9-4
9.4 Defining the Conditions for Analysis	9-5
9.5 Running Your Program	9-8
9.6 Viewing the Analysis Data	9-9
9.7 Summary of Memory System Analysis Commands	9-10
9.8 Entering Analysis Commands Through a Batch File	9-13

9.1 Major Functions of Simulator Memory System Analysis

The 'C6x memory system analysis functionality allows you to set breakpoints on or count any event that is supported by the simulator core. These events are added to the Analysis Events dialog box by polling the simulator core to determine which events it can support.

The 'C6x memory system analysis interface allows you to set breakpoints on or count the following events, which are supported by the 'C6x:

- | | |
|--|---|
| <input type="checkbox"/> Program memory accesses | <input type="checkbox"/> Off-chip program memory accesses |
| <input type="checkbox"/> Program cache hits | <input type="checkbox"/> Off-chip data memory accesses |
| <input type="checkbox"/> Program cache misses | <input type="checkbox"/> Data memory bank conflicts |
| <input type="checkbox"/> Memory stalls | |

You can set breakpoints on or count multiple events.

Set up event breakpoints

'C6x memory system analysis enables you to set breakpoints on system events. These events are called *break events*. You can break on any of the events supported by the 'C6x core, and you can set a breakpoint on one or more of these events.

Break events are automatically counted by the event counter. Each break event is displayed in the Analysis Statistics window, along with the number of times that break event occurred during each execution of the program, the program address on which the last breakpoint occurred, and the name of the routine in which the breakpoint occurred.

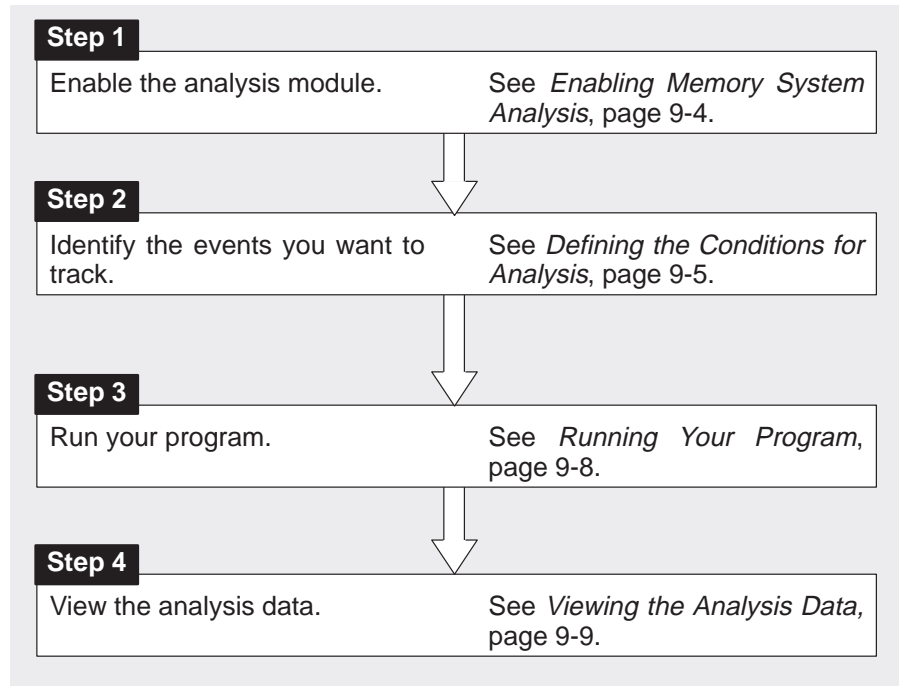
Count system events

You can set up memory system analysis to count occurrences of system events during the execution of your program. You can count any single event or any combination of events supported by the 'C6x. Each event that you select to count is displayed in the Analysis Statistics window, along with the number of times that event occurs during the execution of your program.

You can use all of the basic debugger step and run commands to determine the number of occurrences of each selected event that you want to count. The event counters keep incrementing with each execution of your program unless you select the Reset Counter button on the Analysis Statistics window. When you select the Reset Counter button, each of the event counters resets to 0 immediately.

9.2 Overview of the Analysis Process

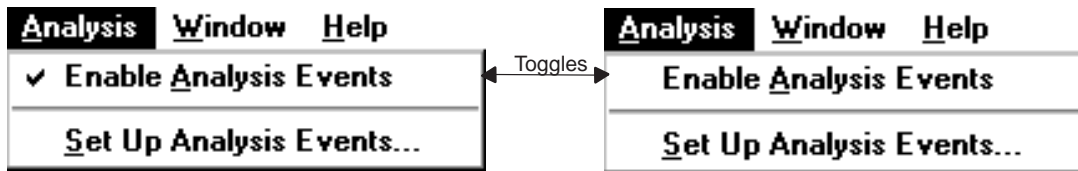
Completing an analysis session consists of four simple steps:



9.3 Enabling Memory System Analysis

When the debugger comes up, analysis is disabled by default. To begin tracking system events, you must explicitly enable analysis by selecting Enable Analysis on the Analysis menu. When you select Enable Analysis, a check mark appears next to the menu item, indicating that analysis is enabled. To disable analysis, select Enable Analysis Events again, and the check mark disappears, indicating that analysis is disabled.

Figure 9–1. Enabling/Disabling the Analysis Interface



When analysis is disabled, all events you previously enabled remain unchanged. You can simply reenable analysis and use the events already defined.

During a single debugging session, you may want to change the analysis parameters several times. For example, you may want to define new parameters such as counting off-chip memory accesses and tracking data memory bank conflicts, etc. To do this, you must open the Analysis Events dialog box, delete any previously defined events that you do not want to monitor, and define the new events you want to track.

Besides the Analysis menu, you can use the Analysis Events dialog box to enable and disable analysis by selecting and deselecting the Enable analysis events check box. For more information about the Analysis Events dialog box, see section 9.4 on page 9-5.

Note:

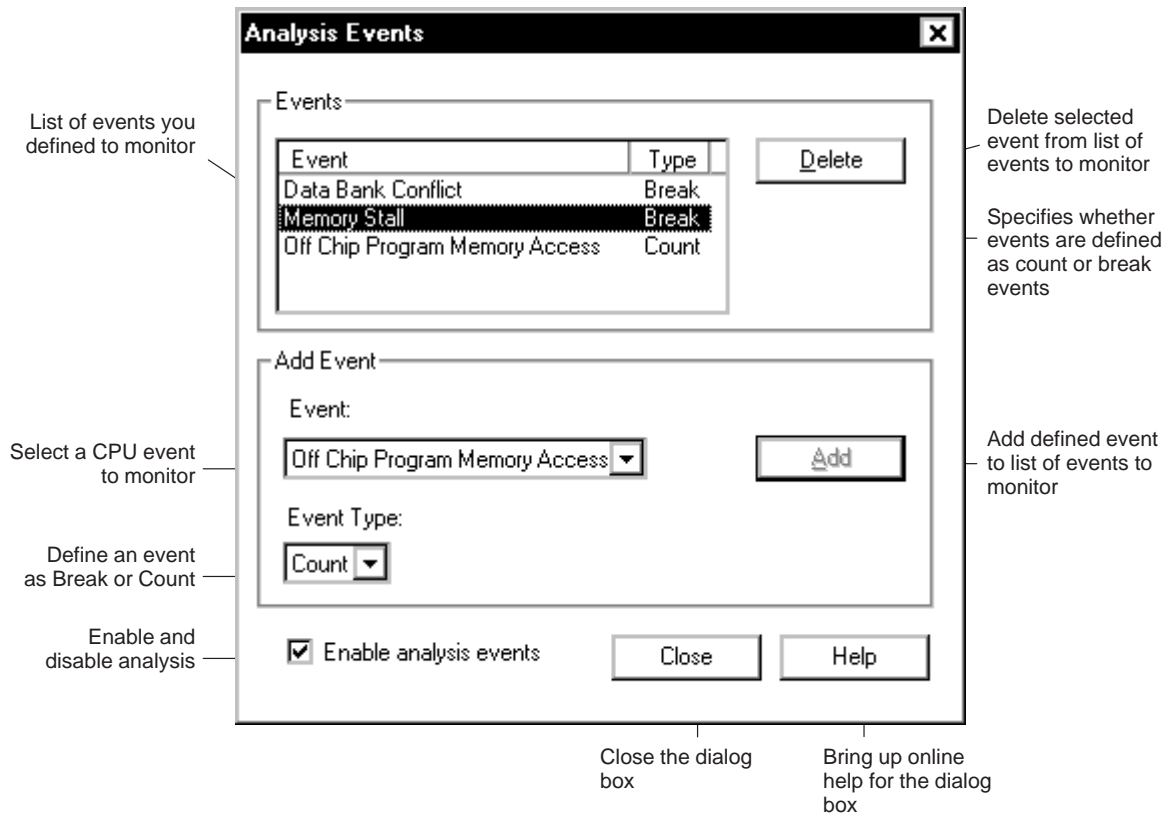
It is not necessary to enable the analysis module each time you run your program.

9.4 Defining the Conditions for Analysis

Memory system analysis detects system events according to the parameters you define for counting events or halting the processor.

To track a particular event, you must define the conditions for analysis. To do this, select the events you want to track by enabling the appropriate conditions in the Analysis Events dialog box. To bring up the Analysis Events dialog box, select the Set Up Analysis Events... option on the Analysis menu. Figure 9–2 illustrates the Analysis Events dialog box.

Figure 9–2. Analysis Events Dialog Box



Description of available system events

Table 9–1 lists and describes the events in the Analysis Events dialog box that are available for the TMS320C6x simulator core.

Table 9–1. Description of Analysis Counter Events

Event	Description
Program memory access	Any program fetch
Program cache hit	Any program memory access resulting in a cache hit. Occurs only when cache is enabled.
Program cache miss	Any program memory access resulting in a cache miss. Occurs only when cache is enabled.
Off-chip program access	A program memory access resulting in an external memory access. A single program fetch from external memory results in eight off-chip memory accesses.
Off-chip data access	A data memory access resulting in an external memory access
Data memory bank conflict	Two data accesses occur in the same memory bank resulting in a CPU stall
Memory stall	The number of cycles lost due to memory stalls. A memory stall occurs when the CPU is stalled because the system is unable to provide the data as soon as it is requested.

Note:

The TMS320C6x simulator core does not fully model the memory system at this time. The host port interface, the timer, and multichannel buffered serial ports are not modeled.

Counting system events

Memory system analysis allows you to count multiple system events at one time. Each event has its own counter. The event count accumulates with each execution of your program. If you do not want the count to accumulate, reset the event counters to 0 by selecting the Reset Counter button before you reexecute your program.

You set up count events in the Analysis Events dialog box. To define system events to count:

- 1) From the Analysis menu, select Set Up Analysis Events.... The Analysis Events dialog box appears.
- 2) In the Add Event section, select the event you want to count from the Event drop list.

- 3) Select Count from the Event Type drop list.
- 4) Click the Add button. The event appears in the event list at the top of the dialog box and in the Analysis Statistics window.
- 5) Define more events to count, or select the Close button.

Setting event breakpoints

You can set a breakpoint on any event that is supported by the simulator core. You can set breakpoints on as many events as you choose. The simulator halts on the first occurrence of any event defined as a break event.

You set up break events in the Analysis Events dialog box. To define system events on which to set a breakpoint:

- 1) From the Analysis menu, select Set Up Analysis Events... The Analysis Events dialog box appears.
- 2) In the Add Event section, select the event on which you want to set a breakpoint from the Event drop list.
- 3) Select Break from the Event Type drop list.
- 4) Click the Add button. The event appears in the event list at the top of the dialog box and also in the Analysis Statistics window.
- 5) Define more events on which to break, or click the Close button.

Removing a defined count or break event

If you decide that you no longer want to count or break on an event, access the Analysis Events dialog box and complete the following steps:

- 1) From the event list in the Event section at the top of the dialog box, click on the event you want to remove.
- 2) Select the Delete button. The event disappears from the event list.
- 3) Add or remove other events, or select the Close button.

9.5 Running Your Program

Once you have defined your parameters, analysis can begin collecting data as soon as you run your program. It stops collecting data when the defined conditions are met. Memory system analysis monitors the progress of the defined events while your program is running.

To run the entire program, use one of these methods:

- ☐ Click the Run icon on the toolbar:



- ☐ From the Target menu, select Run.
- ☐ Press **(F5)**.
- ☐ From the command line, enter the RUN command. The format for this command is:

run [*expression*]

You can use any of the debugger run commands (STEP, CSTEP, NEXT, etc.) described in Chapter 6.

9.6 Viewing the Analysis Data

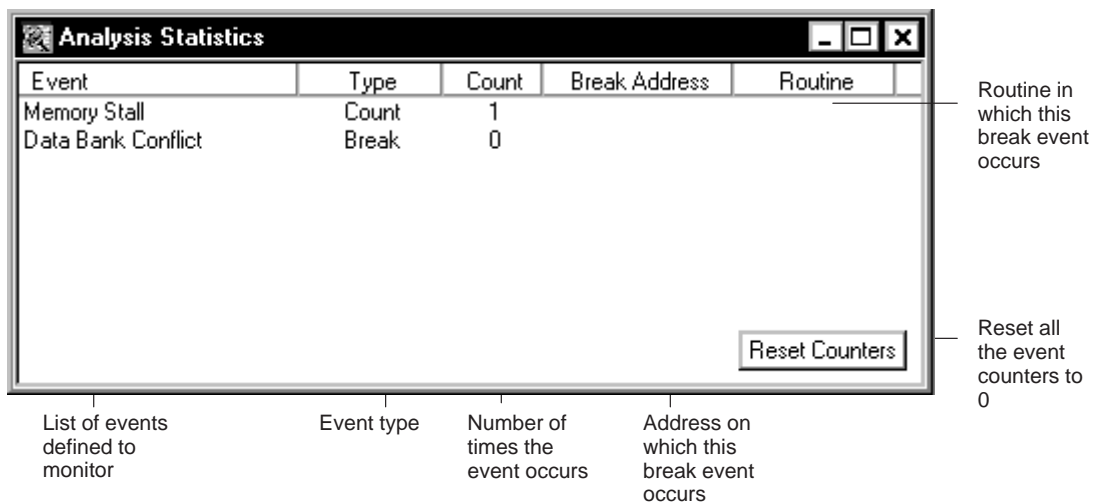
You can monitor the status of the analysis events by checking the Analysis Statistics window. This window displays an ongoing progress report of memory system analysis activity. Through this window, you can monitor the status of the break events and the number of occurrences of each count event that you defined.

If you change any of the analysis options in the Analysis Events dialog box, the Analysis Statistics window updates to reflect the changes you made.

Interpreting the information in the Analysis Statistics window

You can watch the progress of the events that you defined in the Analysis Events dialog box in the Analysis Statistics window. From this window, you can also reset the event counters. Figure 9–3 illustrates the Analysis Statistics window.

Figure 9–3. Analysis Statistics Window Displaying an Ongoing Status Report



Resetting the event counters

The event counters count each occurrence of a defined event until you reset them by selecting the Reset Counter button in the Analysis Statistics window. This means that the counters accumulate occurrences of defined events even if you reexecute your program. If you do not want the event counters to accumulate, select the Reset Counter button on the Analysis Statistics window. When you select the Reset Counter button, the event counters reset to 0 immediately.

9.7 Summary of Memory System Analysis Commands

In addition to the memory system analysis interface, the debugger supports analysis commands that can be entered from the command line or a batch file. For information on setting up a batch file, see section 9.8 on page 9-13.

The debugger supports several analysis commands that allow you to control analysis events. Each of these commands are summarized in Table 9–2 and discussed in detail in the sections following the table.

Table 9–2. Memory System Analysis Command Summary

Command	Alias	Description
event_enable	ee	Enables memory system analysis
event_disable	ed	If no parameter is used, disables memory system analysis. If parameter is used, disables the specified event.
event_break	eb	Configures the specified event as a break event
event_counter_start	ecs	Configures the specified as a count event
event_counter_reset	ecr	If no parameter is used, resets all event counters to 0. If parameter is used, resets the counter for the specified event.
event_list	el	If no parameter is used, lists the configuration for all events in the Command window. If parameter is used, lists the configuration for the specified event.
event_reset	er	Disables analysis and resets all event counters to 0

Note: Determining the number assigned to each event

Some commands have a event_number parameter. To determine what number your simulator core has assigned to each event, use the event_list command discussed on page 9-12.

event_enable (enable specified event)

The event_enable command enables memory system analysis. When memory system analysis is enabled, the processor tracks the events that you define for the processor to monitor. The syntax for the event_enable command is:

event_enable

Or, use the predefined alias:

ee

To enable analysis through the debugger interface, from the Analysis menu, select Enable Analysis.

To enable a particular event through the debugger interface, follow the steps for defining an event as a count event on page 9-6 or defining an event as a break event on page 9-7.

event_disable (disable specified event)

If used without a parameter, the `event_disable` command disables memory system analysis and resets all event counters. If used with the `event_number` parameter, the `event_disable` command disables the specified event, preventing you from counting or halting on that event. The syntax for the `event_disable` command is:

event_disable [*event_number*]

Or, use the predefined alias:

ed [*event_number*]

To disable analysis through the debugger interface, from the Analysis menu, deselect Enable Analysis.

To disable a particular event through the debugger interface, follow the steps for removing a defined event on page 9-7.

event_break (set breakpoint on specified event)

The `event_break` command instructs the processor to halt whenever it encounters the specified event. The syntax for the `event_break` command is:

event_break *event_number*

Or, use the predefined alias:

eb *event_number*

To set a breakpoint on a particular event through the debugger interface, follow the steps for defining an event as a break event on page 9-7.

event_counter_start (count each occurrence of specified event)

The `event_counter_start` command instructs the processor to count each occurrence of the specified event. The syntax for the `event_counter_start` command is:

event_counter_start *event_number*

Or, use the predefined alias:

ecs *event_number*

To count the occurrences of a particular event through the debugger interface, follow the steps for defining an event as a count event on page 9-6.

event_counter_reset (reset counter for specified event)

If used without a parameter, the `event_counter_reset` command resets all event counters to 0. If used with a parameter, the `event_counter_reset` command resets the counter of the specified event to 0. The syntax for the `event_counter_reset` command is:

event_counter_reset *event_number*

Or, use the predefined alias:

ecr *event_number*

You cannot reset the counter for a specific event through the debugger interface; however, you can reset all the event counters for all the events by selecting the Reset Counters button on the Analysis Statistics window. For more information on the Reset Counters button, see page 9-9.

event_reset (disable and clear configuration for all events)

The `event_reset` command disables analysis and resets all event counters to 0. The syntax for the `event_reset` command is:

event_reset

Or, use the predefined alias:

er

To disable all events and reset all event counters through the debugger interface,

- 1) From the Analysis menu, select Enable Analysis Events.
- 2) From the Analysis Statistics window, select the Reset Counters button. For more information on the Reset Counters button, see page 9-9.

event_list (list configuration of all events)

If used without a parameter, the `event_list` command lists in the Command window the configuration of all the events. If used with a parameter, the `event_list` command lists the configuration for the specified event. The configuration includes the event name, event number, whether the event is defined as a count or break event, and the number of occurrences of each count event. The syntax for the `event_list` command is:

event_list [*event_number*]

Or, use the predefined alias:

el [*event_number*]

You also can view the configuration of all analysis events from the debugger interface in the Analysis Statistics window.

9.8 Entering Analysis Commands Through a Batch File

You can use the memory system analysis commands to set up a batch file that automatically sets up your most frequently used analysis settings when the debugger is invoked. Example 9–1 is a sample batch file that loads the program to be analyzed, sets up events to count, enables analysis, runs the program, logs the analysis results to a log file, and lists the analysis results to the Command window. For more information on creating and executing a batch file, see Section 3.3 on page 3-7.

Example 9–1. Sample Memory System Analysis Batch File

```

////////////////////////////////////
; Sample Command File
////////////////////////////////////

load test_prog           ;; Load the program to analyse

;; Perform any other setup

event_counter_start 10    ;; Count program memory accesses
event_counter_start 3     ;; Count program cache hits
event_counter_start 4     ;; Count program cache misses

event_enable             ;; Enable memory system analysis
run                      ;; Run the program

dlog analysis.log        ;; Store output to log file
event_list               ;; List the analysis statistics in
                        ;; Command window

dlog close

;; EOF

```

Monitoring Hardware Functions With the Emulator Analysis Module

The 'C6x has an on-chip analysis module that allows the emulator to monitor hardware functions. Using the analysis module, you can count occurrences of certain hardware functions or set hardware breakpoints on these occurrences.

You access the analysis features through dialog boxes described in this chapter. These dialog boxes provide a transparent means of loading the special set of pseudoregisters that the debugger uses to access the on-chip analysis module.

Topic	Page
10.1 Major Functions of the Analysis Module	10-2
10.2 Overview of the Analysis Process	10-3
10.3 Enabling the Analysis Module	10-4
10.4 Defining the Conditions for Analysis	10-5
10.5 Running Your Program	10-10
10.6 Viewing the Analysis Data	10-11
10.7 Creating Customized Analysis Commands	10-12
10.8 Summary of Analysis Pseudoregisters	10-13

10.1 Major Functions of the Analysis Module

The 'C6x analysis module provides a detailed look into events occurring in hardware, expanding your debugging capabilities beyond software breakpoints. The analysis module examines 'C6x bus cycle information in real time and reacts to this information through actions such as hardware breakpoints and event counting. The analysis module allows you to:

- ☐ **Count events.** The analysis module has an *internal counter* that can count seven types of *events*. You can count the number of times a defined bus event or other internal event occurs during execution of your program. Events that can be counted include:

- | | |
|--------------------|------------------------------|
| ■ CPU clock cycles | ■ Execute packets |
| ■ Pipeline stalls | ■ Interrupt context switches |
| ■ Interrupts taken | ■ Branches taken |
| ■ NOPs | |

You can count only one event at a time.

- ☐ **Set hardware breakpoints.** You can also set up the analysis module to halt the processor during execution of your program. The events that cause the processor to stop are called *break events*. You can define a break event as one or more of the following conditions:

- Any program address
- A low level on the EMU0 pin (EMU0 driven low)
- A low level on the EMU1 pin (EMU1 driven low)

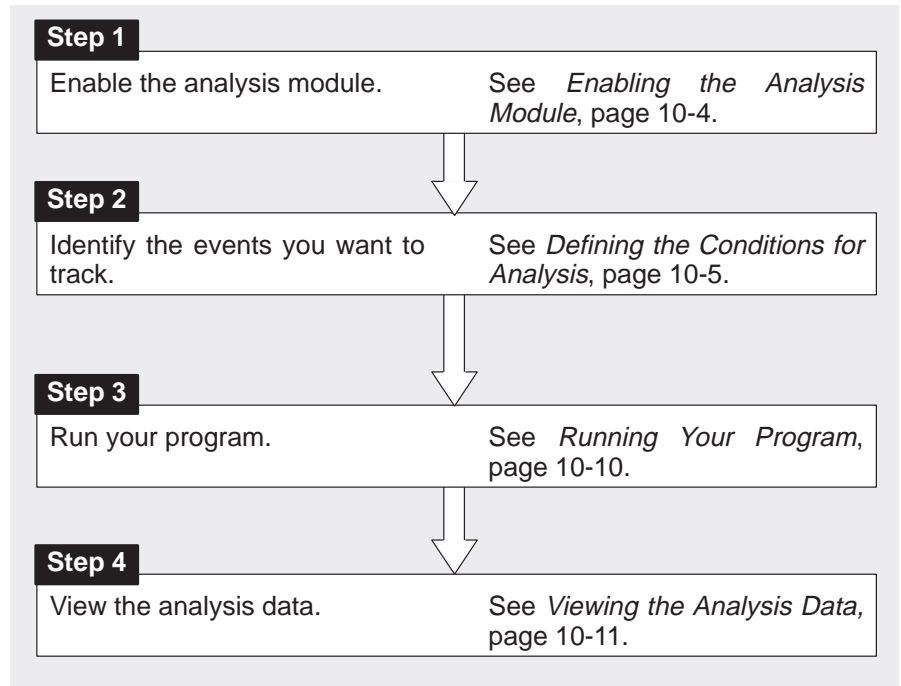
Hardware break events allow you to set breakpoints in ROM and program memory. In addition, any of the debugger's basic features available with software breakpoints can also be used with hardware breakpoints. As a result, you can take advantage of all the step and most of the run commands.

- ☐ **Set global breakpoints with EMU0/1 pins.** In a system of multiple 'C6x processors connected by EMU0/1 (emulation event) pins, setting up the EMU0/1 pins allows you to create global breakpoints. Whenever one processor in your system reaches a breakpoint (software or hardware), *all* processors in the system can be halted.

In addition to setting global breakpoints, you can set up the EMU0 pin to take advantage of the emulator's external counter. Each time the 10-bit internal counter overflows, a signal is sent through the EMU0 pin, incrementing the 32-bit external counter.

10.2 Overview of the Analysis Process

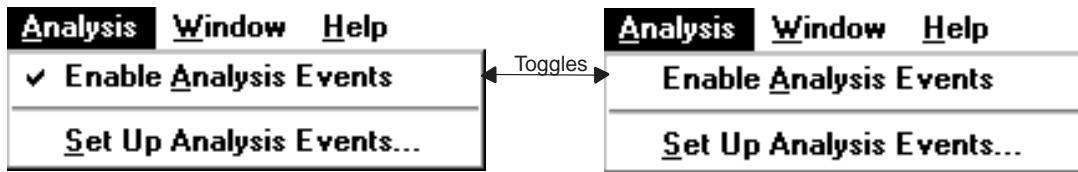
Completing an analysis session consists of four simple steps:



10.3 Enabling the Analysis Module

When the debugger comes up, analysis is disabled by default. To begin tracking system events, you must explicitly enable analysis by selecting Enable Analysis on the Analysis menu. When you select Enable Analysis, a check mark appears next to the menu item, indicating that analysis is enabled. To disable analysis, select Enable Analysis Events again, and the check mark disappears, indicating that analysis is disabled.

Figure 10–1. Enabling/Disabling the Analysis Module



When analysis is disabled, all events you previously enabled remain unchanged. You can simply reenable analysis and use the events already defined.

During a single debugging session, you may want to change the parameters of the analysis module several times. For example, you may want to define new parameters such as counting branches, tracking CPU clock cycles, etc. To do this, you must open the individual dialog boxes, deselect any previous events, and select the new events you want to track.

You also can enable and disable analysis from the Analysis Events dialog boxes by selecting and deselecting the Enable analysis events checkbox. For more information about the Analysis Events dialog boxes, see Section 10.4 on page 10-5.

Note:

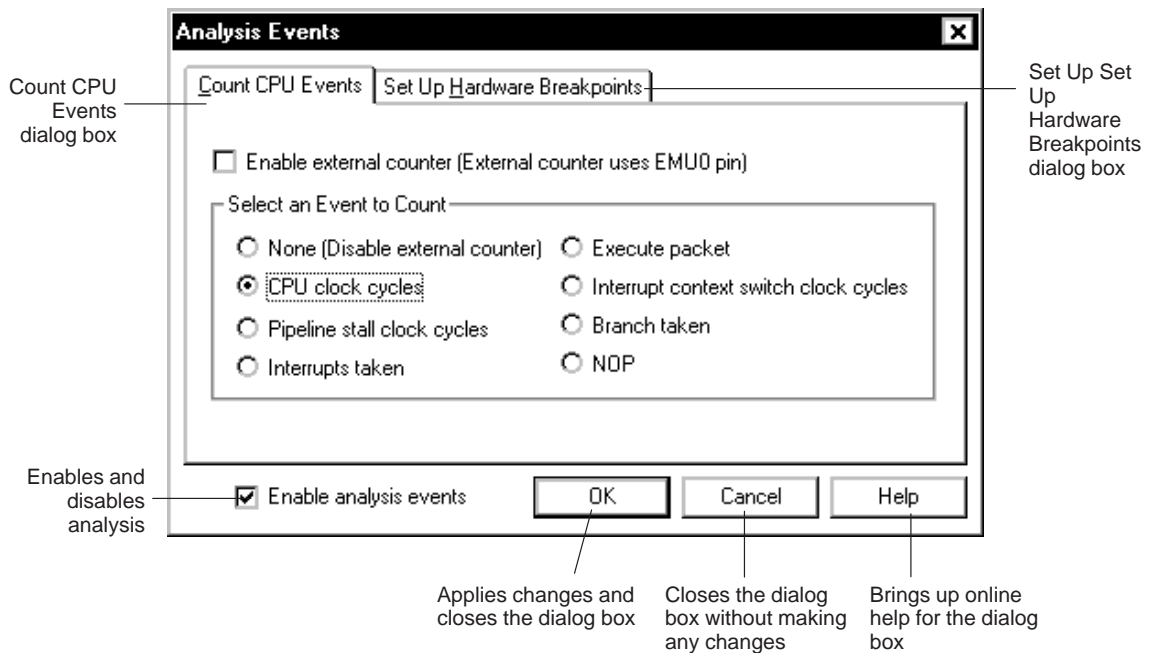
You must enable the analysis module only once during a debugging session. It is not necessary to enable the analysis module each time you run your program.

10.4 Defining the Conditions for Analysis

The analysis module detects hardware events and monitors the internal signals of the processor according to the parameters you define that count events or halt the processor.

You must define the conditions the analysis module must meet to track a particular event. To do this, select the events you want to track by enabling the appropriate conditions in the Analysis Events dialog boxes. To bring up the Analysis Events dialog boxes, select the Set Up Analysis Events... option on the Analysis menu.

Figure 10–2. Analysis Events Dialog Boxes



Counting events

The analysis module's internal counter counts bus events and detects other internal events. This counter keeps track of how many times an event occurs. The Count CPU Events dialog box allows you to count one of the seven types of events until the processor halts. These events are listed and described in Table 10–1.

To count any of the events, simply select that event in the Count CPU Events dialog box. You can count only one event at a time. Table 10–1 describes the available events.

Table 10–1. Description of Analysis Counter Events

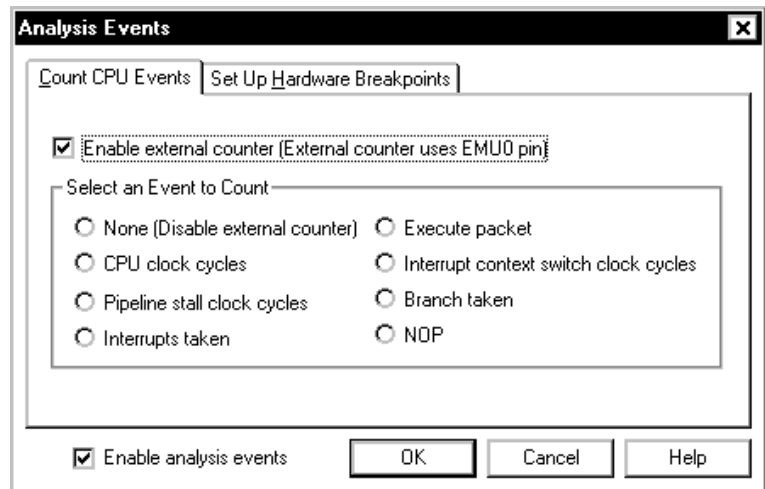
Event	Description
None (disable internal counter)	Do not count any events; disable internal counter
CPU clock cycles	Count the number of CPU clock cycles
Pipeline stall clock cycles	Count the number of CPU clock cycles during a pipeline stall
Interrupt taken	Count the number of interrupts detected
Execute packet	Count the number of execution packets
Interrupt context switch clock cycles	Count the number of CPU clock cycles during an interrupt context switch
Branch taken	Count the number of branches taken
NOP	Count the number of NOPs detected

To watch the progress of the event counter, view the status of the event in the Analysis Statistics window.

If you do not want to count any events, select None to disable the internal counter.

Enabling the external counter

The emulator's *external counter* keeps track of the internal counter. The internal counter is a 10-bit, decremental counter that can keep track of a maximum of 1024 events. The external counter, however, is a 32-bit counter. Each time the internal counter overflows, a signal sent through the EMU0 pin increments the external counter. To enable the emulator's external counter, simply select the external counter checkbox in the Count CPU Events dialog box.



Note:

Enabling the external counter in the Count CPU Events dialog box carries the following restrictions:

- ☐ You can enable only one external counter when you have multiple processors (that are connected by their EMU0/1 pins) in a system.
- ☐ No other external devices can actively drive the EMU0 pin. The EMU0 pin option is disabled in the Set Up Hardware Breakpoints dialog box.

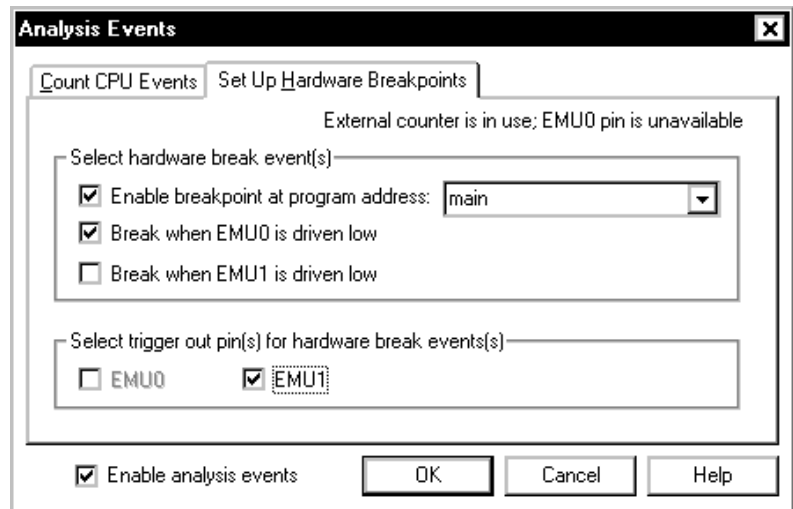
Setting hardware breakpoints

You can set a hardware breakpoint, which halts the processor, on three types of events:

- ☐ A specified program address
- ☐ EMU0 detected low
- ☐ EMU1 detected low

You can select as many events as you want. To specify an event or events on which to halt the processor, follow these steps:

- 1) From the Analysis menu, select Hardware Breakpoints.... The Set Up Hardware Breakpoints dialog box appears.



- 2) Select the event or events on which you want to set a hardware breakpoint by clicking the check box(es) next to that event or events.

If you want to enable a hardware breakpoint at a particular program address, you can enter the program address as a symbol or value in any format. If you choose to enter the program address in hexadecimal format, be sure to begin the address with **0x**. You can also select a previously entered address from the drop-down list.

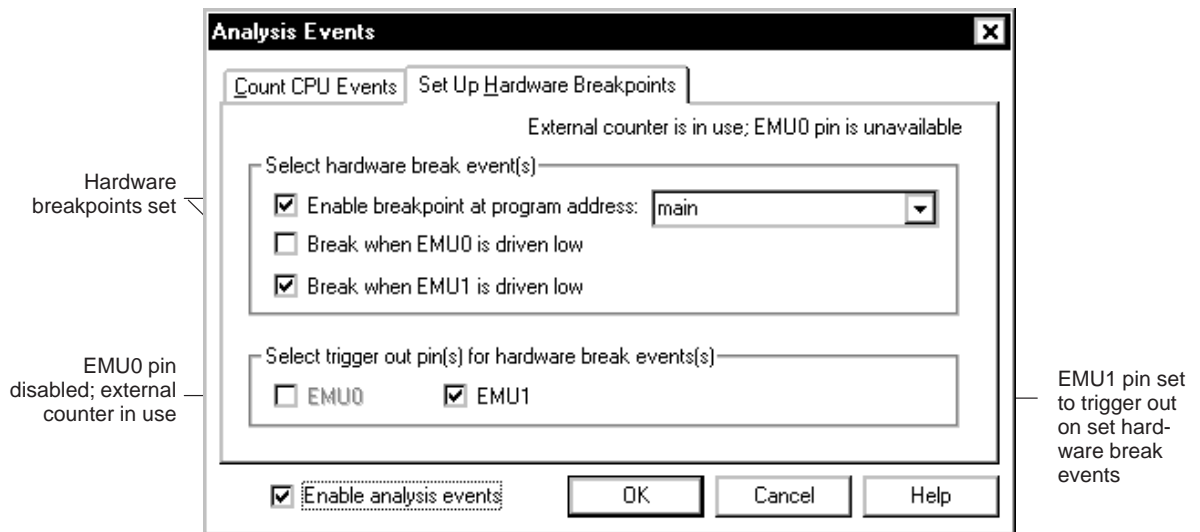
Setting up the EMU0/1 pins to set global breakpoints

By default, the EMU0/1 pins are set up as input signals; however, you can set them up as output signals or *trigger out* whenever the processor is halted by a software or hardware breakpoint. This is extremely useful when you have multiple 'C6x processors in a system connected by their EMU0/1 pins.

To set the EMU0/1 pins to output, select the check box next to the EMU pin on which you want to output in the Select trigger out pin(s) for hardware break event(s) field in the Set Up Hardware Breakpoints dialog box.

Selecting EMU0/1 does not, however, automatically halt all processors in the system. To do so, you must enable the EMU0/1 driven-low condition in the Set Up Hardware Breakpoints dialog box. For example, if you have a system consisting of two processors connected by their EMU1 pins and you want to halt both processors when this pin is driven low, you must enable the EMU1 driven low option in the Set Up Hardware Breakpoints dialog box of one of the processors, as shown in Figure 10–3.

Figure 10–3. EMU1 Pin Set Up to Trigger Out on Hardware Break Events



When processor 1 halts, its EMU1 signal halts processor 2. Setting up each processor in this way creates a global breakpoint so that any processor that reaches a breakpoint halts all other processors in the system.

10.5 Running Your Program

Once you have defined your parameters, the analysis module can begin collecting data as soon as you run your program. It stops collecting data when the defined conditions are met. The analysis module monitors the progress of the defined events while your program is running.

Note:

The conditions for the analysis session must be defined *before* your analysis session begins; you cannot change conditions *during* execution of your program.

How to run the entire program

To run the entire program, use one of these methods:

- ☐ Click the Run icon on the toolbar:



- ☐ From the Target menu, select Run.
- ☐ Press **F5**.
- ☐ From the command line, enter the RUN command. The format for this command is:

run [*expression*]

You can use any of the debugger run commands (STEP, CSTEP, NEXT, etc.) described in Chapter 6 except the RUNB (run benchmarks) or RUNF (run free) command.

How the Run Benchmarks (RUNB) command affects analysis

Running your program by selecting the Run Benchmarks option from the Target menu or entering the RUNB command from the command line disables the current analysis settings and configures the counter to count CPU clock cycles. When the processor is halted after a RUNB, the analysis registers are restored to their original states.

The analysis module provides capabilities in addition to those provided by the RUNB command. With the RUNB command you can count the number of CPU clock cycles only during the execution of a specific section of code. However, the analysis module not only allows you to count CPU clock cycles, it also allows you to count other events.

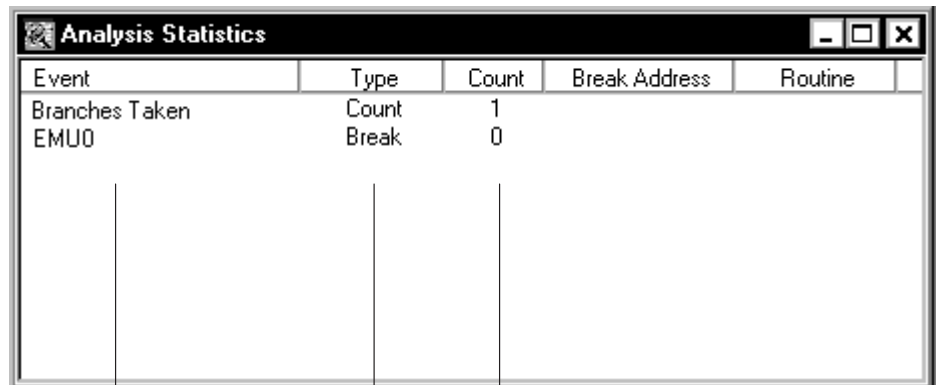
10.6 Viewing the Analysis Data

You can monitor the status of the analysis module by checking the Analysis Statistics window. This window displays an ongoing progress report of the analysis module's activity. Through this window, you can monitor the status of the break events, the value of both the internal and external event counters, and the status of the EMU0/1 events.

Interpreting the information in the Analysis Statistics window

You can watch the progress of the events that you defined in the Analysis Events dialog box in the Analysis Statistics window. From this window, you can also change the display rate of the information in the window. If you change any of the analysis options in the analysis dialog boxes, the Analysis Statistics window updates to reflect the changes you made. Figure 10–4 illustrates the Analysis Statistics window.

Figure 10–4. Analysis Statistics Window Displaying an Ongoing Status Report



Event	Type	Count	Break Address	Routine
Branches Taken	Count	1		
EMU0	Break	0		

Figure 10–4 is a screenshot of the 'Analysis Statistics' window. The window has a title bar with the text 'Analysis Statistics' and standard window controls (minimize, maximize, close). Below the title bar is a table with five columns: 'Event', 'Type', 'Count', 'Break Address', and 'Routine'. The table contains two rows of data. The first row shows 'Branches Taken' as the event, 'Count' as the type, and '1' as the count. The second row shows 'EMU0' as the event, 'Break' as the type, and '0' as the count. Below the table, there are five labels with vertical lines pointing to the corresponding columns: 'List of events defined to monitor' points to the 'Event' column, 'Event type' points to the 'Type' column, 'Number of times the event occurs' points to the 'Count' column, 'Address on which this break event occurs' points to the 'Break Address' column, and 'Routine in which this break event occurs' points to the 'Routine' column.

10.7 Creating Customized Analysis Commands

The interface to the 'C6x emulator analysis module is register based. You can set up hardware breakpoints or counter events through the Analysis menu dialog boxes. In some cases, however, you may want to define more complex conditions for the processor to detect. Or, you may want to write a batch file that defines breakpoint and/or counter conditions. In either case, you can accomplish these tasks by accessing the analysis registers through the debugger.

By manipulating the analysis registers, you can customize commands for more complex instructions that do not exist on the Hardware Breakpoints or Count CPU Events dialog boxes. Use the ALIAS and EVAL commands to create your own commands. The basic syntax for creating customized analysis commands is:

alias *command_name*, "**eval** *register name* = *bit value*"

For example, to create a new command for turning on the analysis module, enter:

```
alias analysis_on, "eval AEN = 3"
```

To create a new command to enable the external emulator counter, enter:

```
alias xcount, "eval ACE = 0x22"
```

10.8 Summary of Analysis Pseudoregisters

To create your own analysis commands, you must familiarize yourself with the seven analysis registers and how they work. The following subsections discuss the analysis registers briefly.

AEN (enable analysis)

You can enable and disable the analysis module and set the EMU0/1 pins to trigger out by using the AEN register. Set the bit to 1 to enable or to 0 to disable.

Bit Number	Bit Field Value (in Hex)	Definition
0:1		Enable/disable analysis module
	0	Disable analysis module
	1	Reserved
	2	Reserved
	3	Enable analysis module
2		Reserved
3		Select EMU0 trigger out
4		Select EMU1 trigger out

When you disable analysis, all registers except AEN retain their previous state.

ABE (configure hardware breakpoints)

The ABE register configures hardware breakpoint events. Set the bit to 1 to enable or to 0 to disable. The hardware breakpoint event bits are defined as follows:

Bit Number	Definition
0	Enable program address breakpoint
1	Enable breakpoint when EMU0 is driven low
2	Enable breakpoint when EMU1 is driven low

Hardware breakpoints do not halt the processor during a step. If the hardware breakpoint is set on the same instruction as a software breakpoint, hardware breakpoints do not halt the processor.

ADR (program address breakpoint value)

The ADR register holds the 32-bit value of the program address breakpoint.

ACE (configure analysis counter events)

The ACE register configures the analysis counter to count a defined event. Set the bit to 1 to enable or to 0 to disable. The counter event bits are defined as follows:

Bit Number	Bit Field Value (in Hex)	Definition
0:1		Defines counter mode
	0	Counter disabled
	1	Reserved
	2	Performance counting
2:4	3	Error generated
		Selects counter event
	0	Clock
	1	Execution packet
	2	Pipeline stall
	3	Interrupt context switch
	4	Interrupt acknowledge
	5	Branch taken
	6	NOP
	7	Non-NOP instructions
5		Enable external emulator counter

ICNT (internal counter value)

The ICNT register holds the 10-bit value of the internal counter.

XCNT (external counter value)

The XCNT register holds the 32-bit value of the internal counter.

AST (analysis status)

The AST register records the occurrence of enabled events. Set the bit to 1 to enable or to 0 to disable. The status bits are defined as follows:

Bit Number	Definition
0	Program address breakpoint
1	EMU0 driven low breakpoint
2	EMU1 driven low breakpoint

Run commands do not interfere with the status bits because the status bits are cleared before command execution.

Using the Parallel Debug Manager

The TMS320C6x emulation system is a true multiprocessing debugging system. It allows you to debug your entire application by using the parallel debug manager (PDM). The PDM is a command shell that controls and coordinates multiple debuggers. This chapter describes the functions that you can perform with the PDM.

See Chapter 2, *Getting Started With the Debugger*, for information about invoking the PDM and debuggers.

Topic	Page
11.1 Identifying Processors and Groups	11-2
11.2 Sending Debugger Commands to One or More Debuggers	11-6
11.3 Running and Halting Code	11-7
11.4 Entering PDM Commands	11-9
11.5 Defining Your Own Command Strings	11-15
11.6 Entering Operating-System Commands	11-16
11.7 Understanding the PDM's Expression Analysis	11-17
11.8 Using System Variables	11-18
11.9 Evaluating Expressions	11-21

11.1 Identifying Processors and Groups

You can send commands to an individual processor or to a group of processors. To do this, you must assign names to the individual processors or to groups of processors. Individual processor names are assigned when you invoke the individual debuggers; you can assign group names with the SET command after the individual processor names have been assigned.

Note:


Each debugger that runs under the PDM must have a unique processor name. The PDM does not keep track of existing processor names. When you send a command to a debugger, the PDM will validate the existence of a debugger invoked with that processor name.

Assigning names to individual processors

You must associate each debugger within the multiprocessing system with a unique name, referred to as a *processor name*. The processor name is used for:

- ☐ Identifying a processor to send commands to
- ☐ Assigning a processor to a group
- ☐ Setting the default prompts for the associated debuggers. For example, if you invoke a debugger with the processor name CPU_A, that debugger's prompt will be CPU_A>.
- ☐ Identifying the individual debuggers on the screen (Sun systems only). The processor name that you assign will appear at the top of the operating-system window that contains the debugger. Additionally, if you turn one of the windows into an icon, the icon name is the same as the processor name that you assigned.

To assign a processor name, you can use the `-n` option when you invoke a debugger. For example, to name one of the 'C6x processors CPU_B, you would use the following command to invoke the debugger:

```
spawn emu6x -n CPU_B 
```

From this point on, whenever you needed to identify this debugger, you could identify it by its processor name, CPU_B.

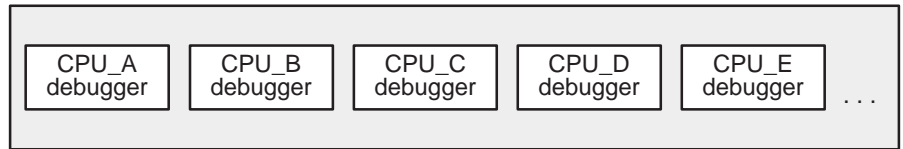
The processor name that you supply can consist of up to eight alphanumeric characters or underscore characters and must begin with an alphabetic character. The name is not case sensitive. The processor name must match one of the names defined in your board configuration file (refer to Appendix B, *Describing Your Target System to the Debugger*).

Organizing processors into groups

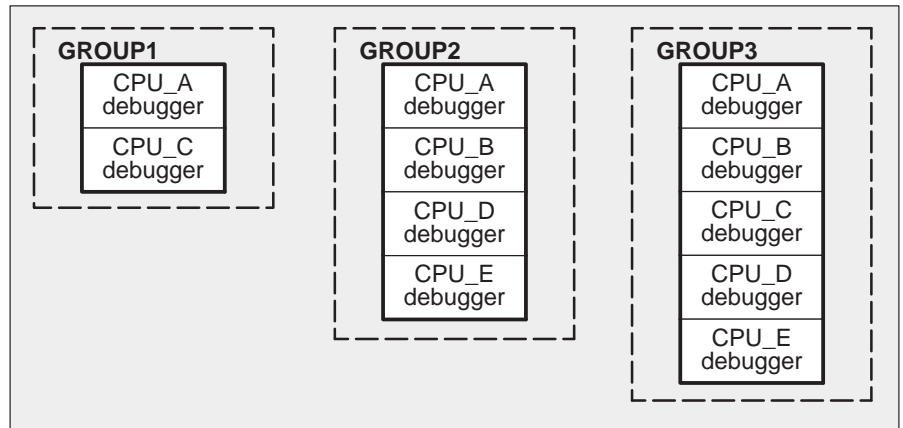
You can organize processors into groups by using the SET command to group processors under one name. Each processor can belong to any group, all groups, or a group of its own. Figure 11–1 (a) shows an example of processors in a system, and Figure 11–1 (b) illustrates three examples of named groups. GROUP1 contains two processors, GROUP2 contains four processors, and GROUP3 contains five processors.

Figure 11–1. Grouping Processors

(a) All possible processors in a system



(b) Examples of how processors could be grouped



To define and manipulate software groupings of named processors, use the SET and UNSET commands.

❑ **Defining a group of processors**

To define a group, use the SET command. The format for this command is:

```
set [group name [= list of processor names] ]
```

This command allows you to specify a group name and the list of processors you want in the group. The *group name* can consist of up to 128 alphanumeric characters or underscore characters.

For example, to create the GROUP1 group illustrated in Figure 11–1 (b), you could enter the following on the PDM command line:

```
set GROUP1 = CPU_A CPU_C 
```

The result is a group called GROUP1 that contains the processors named CPU_A and CPU_C. The order in which you add processors to a group is the same order in which commands will be sent to the members of that group.

❑ **Setting the default group**

Many of the PDM commands can be sent to groups; if you often send commands to the same group and you want to avoid typing the group name each time, you can assign a default group.

To set the default group, use the SET command with a special group name called dgroup. For example, if you want the default group to contain the processors called CPU_B, CPU_D, and CPU_E, enter:

```
set dgroup = CPU_B CPU_D CPU_E 
```

The PDM automatically sends commands to the default group when you do not specify a group name.

❑ **Modifying an existing group or creating a group based on another group**

Once you have created a group, you can add processors to it by using the SET command and preceding the existing *group name* with a dollar sign (\$) in the list of processors. You can also use a group as part of another group by preceding the existing group's name with a dollar sign. The dollar sign tells the PDM to use the processors listed previously in the group as part of the new list of processors.

Suppose GROUPA contained CPU_C and CPU_D. If you wanted to add CPU_E to the group, you would enter:

```
set GROUPA = $GROUPA CPU_E 
```


After entering this command, GROUPA would contain CPU_C, CPU_D, and CPU_E.

If you decided to send numerous commands to GROUPA, you could make it the default group:

```
set dgroup = $GROUPA
```

□ Listing all groups of processors

To list all groups of processors in the system, use the SET command without any parameters:

```
set
```

The PDM lists all of the groups and the processors associated with them:

```
GROUP1  "CPU_A CPU_C"
GROUPA  "CPU_C CPU_D CPU_E"
dgroup  "CPU_C CPU_D CPU_E"
```

You can also list all of the processors associated with a particular group by supplying a group name:

```
set dgroup
dgroup  "CPU_C CPU_D CPU_E"
```

□ Deleting a group

To delete a group, use the UNSET command. The format for this command is:

```
unset group name
```

You can use this command in conjunction with the SET command to remove a particular processor from a group. For example, suppose GROUPB contained CPU_A, CPU_C, CPU_D, and CPU_E. If you wanted to remove CPU_E, you could enter:

```
unset GROUPB
set GROUPB = CPU_A CPU_C CPU_D
```

If you want to delete all of the groups you have created, use the UNSET command with an asterisk instead of a group name:

```
unset *
```

The asterisk *does not* work as a wild card.

Note:

When you use UNSET * to delete all of your groups, the default group (dgroup) is also deleted. As a result, if you issue a command such as PRUN and do not specify a group or processor, the command will fail because the PDM cannot find the default group name (dgroup).

11.2 Sending Debugger Commands to One or More Debuggers


The SEND command sends a debugger command to an individual processor or to a group of processors. The command is sent directly to the command interpreter of the individual debuggers. You can send any valid debugger command string.

The syntax for the SEND command is:

send [-r] [-g {group | processor name}] *debugger command*

- ☐ The **-g** option specifies the group or processor that the debugger command should be sent to. If you do not use this option, the command is sent to the default group (dgroup).
- ☐ The **-r** (return) option determines when control returns to the PDM command line:

- ☒ **Without -r**, control is not returned to the command line until each debugger in the group finishes running code. Any results that would be printed in the COMMAND window of the individual debuggers will also be echoed in the PDM command window. These results will be displayed by the processor. For example:

```
send ?pc   
[CPU_C] 0x200A  
[CPU_D] 0x2008
```

If you want to break out of a synchronous command and regain control of the PDM command line, press **CONTROL C** in the PDM window. This will return control to the PDM command line. However, no debugger executing the command will be interrupted.

- ☒ **With -r**, control is returned to the command line immediately, even if a debugger is still executing a command. When you use **-r**, you *do not* see the results of the commands that the debuggers are executing.

The **-r** option is useful when you want to exit from a debugger but not from the PDM. When you send the QUIT command to a debugger or group of debuggers without using the **-r** command, you will not be able to enter another PDM command until all debuggers that QUIT was sent to finish quitting; the PDM waits for a response from all of the debuggers that are quitting. By using **-r**, you can gain immediate control of the PDM and continue sending commands to the remaining debuggers.

The SEND command is useful for loading a common object file into a group of debuggers. For example, to load a file called test.out into the debuggers contained in GROUP_A, you could use the following command:

```
send -g GROUP_A load test.out 
```

11.3 Running and Halting Code

The PRUN, PRUNF, and PSTEP commands synchronize the debuggers to cause the processors to begin execution at the same real time.

- ☐ PRUNF starts the processors running free, which means they are disconnected from the emulator.
- ☐ PRUN starts the processors running under the control of the emulator.
- ☐ PSTEP causes the processors to single-step synchronously through assembly language code with interrupts disabled.

The formats for these commands are:

prunf `[-g {group | processor name}]`

prun `[-r] [-g {group | processor name}]`

pstep `[-g {group | processor name}] [count]`

- ☐ The **-g** option identifies the group or processor that the command should be sent to. If you do not use this option, the command is sent to the default group (dgroup).
- ☐ The **-r** (return) option for the PRUN command determines when control returns to the PDM command line:
 - **Without -r**, control is not returned to the command line until each debugger in the group finishes running code. If you want to break out of a synchronous command and regain control of the PDM command line, press `(CONTROL) (C)` in the PDM window. This will return control to the PDM command line. However, no debugger executing the command will be interrupted.
 - **With -r**, control is returned to the command line immediately, even if a debugger is still executing a command. You can type new commands, but the processors cannot execute the commands until they finish with the current command; however, you can perform PHALT, PESCE, and STAT commands when the processors are still executing.
- ☐ You can specify a *count* for the PSTEP command so that each processor in the group will step for *count* number of times.

Note:

If the current statement that a processor is pointing to has a breakpoint, that processor will not step synchronously with the other processors when you use the PSTEP command. However, that processor will still single-step.

Halting processors at the same time

You can use the PHALT command after you enter a PRUNF command to stop an individual processor or a group of processors (global halt). Each processor in the group is halted at the same real time. The syntax for the PHALT command is:

```
phalt [-g {group | processor name}]
```

Sending ESCAPE to all processors

Use the PESC command to send the escape key to an individual processor or to a group of processors after you execute a PRUN command. Entering PESC is essentially like typing an escape key in all of the individual debuggers. However, the PESC command is *asynchronous*; the processors do not halt at the same real time. When you halt a group of processors, the individual processors are halted in the order in which they were added to the group.

The syntax for this command is:

```
pesc [-g {group | processor name}]
```

Finding the execution status of a processor or a group of processors

The STAT command tells you whether a processor is running or halted. If a processor is halted when you execute this command, then the PDM also lists the current PC value for that processor. The syntax for the command is:

```
stat [-g {group | processor name}]
```

For example, to find the execution status of all of the processors in GROUP_A after you have executed a global PRUN, enter:

```
stat -g GROUP_A 
```

After entering this command, you will see something similar to this in the PDM window:

```
[CPU_C] Running  
[CPU_D] Halted   PC=201A  
[CPU_E] Running
```

11.4 Entering PDM Commands

The PDM provides a flexible command-entry interface that allows you to:

- ☐ Execute PDM commands from a batch file
- ☐ Record the information shown in the PDM display area
- ☐ Conditionally execute or loop through PDM commands
- ☐ Echo strings to the PDM display area
- ☐ Pause command execution
- ☐ Repeat previously entered commands (use the command history)

This section describes the PDM commands that you can use to perform these tasks.

Executing PDM commands from a batch file

The TAKE command tells the PDM to execute commands from a batch file. The syntax for the PDM version of this command is:

take *batch filename*

The *batch filename* **must** have a .pdm extension, or the PDM will not be able to read the file. If you do not supply a pathname as part of the filename, the PDM first looks in the current directory and then searches directories named with the D_DIR environment variable.

The TAKE command is similar to the debugger version of this command (described on page 12-55). However, there are some differences when you enter TAKE as a PDM command instead of a debugger command.

- ☐ **Similarities.** As with the debugger version of the TAKE command, you can nest batch files up to 10 deep.
- ☐ **Differences.** Unlike the debugger version of the TAKE command:
 - There is no suppress-echo-flag parameter. Therefore, all command output is echoed to the PDM window, and this behavior cannot be changed.
 - To halt batch-file execution, you must press **CONTROL C** instead of **ESC**.
 - The batch file must contain only PDM commands (no debugger commands).

The TAKE command is advantageous for executing a batch file in which you have defined often-used aliases. Additionally, you can use the SET command in a batch file to set up group configurations that you use frequently, and then execute that file with the TAKE command. You can also put your flow-control commands (described in *Controlling PDM command execution on page 11-10*) in a batch file and execute the file with the TAKE command.

Recording information from the PDM display area

By using the DLOG command, you can record the information shown in the PDM display area into a log file. This command is identical to the debugger DLOG command described on page 12-20.

- ☐ To begin recording the information shown in the PDM display area, use:

dlog *filename*

This command opens a log file called *filename* that the information is recorded into. If you plan to execute the log file with the TAKE command, the filename *must* have a .pdm extension.

- ☐ To end the recording session, enter:

dlog close 

If necessary, you can write over existing log files or append additional information to existing files. The extended format for the DLOG command is:

dlog *filename* [{**a** | **w**}]

The optional parameters control how the log file is created and/or used:

- ☐ **Appending to an existing file.** Use the **a** parameter to open an existing file and append the information in the display area.
- ☐ **Writing over an existing file.** Use the **w** parameter to open an existing file and write over the current contents of the file. This is the default action if you specify an existing filename without using either the **a** or **w** options; you will lose the contents of an existing file if you do not use the append (a) option.

Controlling PDM command execution

You can control the flow of PDM commands in a batch file or interactively. With the IF/ELIF/ELSE/ENDIF or LOOP/BREAK/CONTINUE/ENDLOOP flow-control commands, you can conditionally execute debugger commands or set up a looping situation, respectively.

- ☐ To conditionally execute PDM commands, use the IF/ELIF/ELSE/ENDIF commands. The syntax is:

```

if expression
  PDM commands
[elif expression
  PDM commands]
[else
  PDM commands]
endif
    
```




- If the expression for the IF is nonzero, the PDM executes all commands between the IF and the ELIF, ELSE, or ENDIF.
 - The ELIF is optional. If the expression for the ELIF is nonzero, the PDM executes all commands between the ELIF and the ELSE or ENDIF.
 - The ELSE is optional. If the expressions for the IF and ELIF (if present) are false (zero), the PDM executes the commands between the ELSE and the ENDIF.
- To set up a looping situation to execute PDM commands, use the LOOP/BREAK/CONTINUE/ENDLOOP commands. The syntax is:

```
loop   Boolean expression
        PDM commands
[break]
[continue]
endloop
```

The PDM version of the LOOP command is different from the debugger version of this command (described on page 12-29). Instead of accepting any expression, the PDM version of the LOOP command evaluates only Boolean expressions. If the Boolean expression evaluates to true (1), the PDM executes all commands between the LOOP and the BREAK, CONTINUE, or ENDLOOP. If the Boolean expression evaluates to false (0), the loop is not entered.

- The optional BREAK command allows you to exit the loop without having to reach the ENDLOOP. This is helpful when you are testing a group of processors and want to exit if an error is detected.
- The CONTINUE command, which is also optional, acts as a goto and returns command flow to the enclosing LOOP command. CONTINUE is useful when the part of the loop that follows is complicated, and returning to the top of the loop avoids further nesting.

You can enter the flow-control commands interactively or include the commands in a batch file that is executed by the TAKE command. When you enter LOOP or IF from the PDM command line, a question mark (?) prompts you for the next entry:

```
PDM:11>>if $i > 10 
?echo ERROR IN TEST CASE 
?endif 
ERROR IN TEST CASE

PDM:12>>
```

The PDM continues to prompt you for input using the ? until you enter ENDIF (for an IF command) or ENDLOOP (for a LOOP command). After you enter ENDIF or ENDLOOP, the PDM immediately executes the IF or LOOP command.

If you are in the middle of interactively entering a LOOP or IF statement and want to abort it, type (CONTROL) (C).

You can use the IF/ENDIF and LOOP/ENDLOOP commands together to perform a series of tests. For example, within a batch file, you can create a loop like the following (the SET and @ commands are described in section 11.8, beginning on page 11-18):

set i = 10	<i>Set the counter (i) to 10.</i>
loop \$i > 0	<i>Loop while i is greater than 0.</i>
·	
test commands	
·	
if \$k > 500	<i>Test for error condition.</i>
echo ERROR ON TEST CASE 8	<i>Display an error message.</i>
endif	
·	
@ i = \$i - 1	<i>Decrement the counter.</i>
endloop	

You can record the results of this loop in a log file (refer to page 11-10) to examine which test cases failed during the testing session.


Echoing strings to the PDM display area

You can display a string in the PDM display area by using the ECHO command. This command is especially useful when you are executing a batch file or running a flow-control command such as IF or LOOP. The syntax for the command is:

echo *string*

This displays the *string* in the PDM display area.

You can also use ECHO to show the contents of a system variable (system variables are described in section 11.8):

```
echo $var_proc1 
```

34

The PDM version of the ECHO command works in exactly the same way as the debugger version described on page 12-21 works, except that you can use the PDM version outside of a batch file.

Pausing command execution

Sometimes you may want the PDM to pause while it is running a batch file or when it is executing a flow control command such as LOOP/ENDLOOP. Pausing is especially helpful in debugging the commands in a batch file.

The syntax for the PAUSE command is:

pause

When the PDM reads this command in a batch file or during a flow control command segment, the PDM stops execution and displays the following message:

```
<< pause - type return >>
```

To continue processing, press .

Using the command history

The PDM supports a command history that is similar to the UNIX command history. The PDM prompt identifies the number of the current command. This number is incremented with every command. For example, PDM:12>> indicates that eleven commands have previously been entered, and the PDM is now ready to accept the twelfth command.

The PDM command history allows you to re-enter any of the last twenty commands:


- ☐ To repeat the last command that you entered, type:

```
!! 
```

- ☐ To repeat any of the last twenty commands, use the following command:

!number

number is the number of the PDM prompt that contains the command that you want to re-enter. For example,

```
PDM:100>>echo hello 
hello
PDM:101>>echo goodbye 
goodbye
PDM:102>>!100 
echo hello
hello
```

Notice that the PDM displays the command that you are re-entering.

- ❑ An alternate way to repeat any of the last twenty commands is to use:

!string

This command tells the PDM to execute the last command that began with *string*. For example,

```
PDM:103>>pstep -g GROUPA
PDM:104>>send -g GROUPA ?pc
[CPU_C] 0x2000
[CPU_D] 0x2004
PDM:103>>pstep -g GROUPB
PDM:104>>send -g GROUPB ?pc
[CPU_A] 0x201A
[CPU_E] 0x2014
PDM:105>>!p
pstep -g GROUPB
```

- ❑ To see a list of the last twenty commands that you entered, type:

history

The command history for the PDM works differently from that of the debugger; the **TAB** and **F2** keys have no command-history meaning for the PDM.

11.5 Defining Your Own Command Strings

The ALIAS command provides a shorthand method of entering often-used commands or command sequences. The UNALIAS command deletes one or more ALIAS definitions. The syntax for the PDM version of each of these commands is:

```
alias [alias name [, "command string"]]
unalias {alias name | *}
```

The PDM versions of the ALIAS and UNALIAS commands are similar to the debugger versions of these commands. You can:

- ☐ Include several commands in the command string by separating the individual commands with semicolons
- ☐ Define parameters in the command string by using a percent sign and a number (%1, %2, etc.) to represent a parameter whose value will be supplied when you execute the aliased command
- ☐ List all currently defined PDM aliases by entering ALIAS with no parameters
- ☐ Find the definition of a PDM alias by entering ALIAS with only an alias-name parameter
- ☐ Nest alias definitions
- ☐ Redefine an alias
- ☐ Delete a single PDM alias by supplying the UNALIAS command with an alias name or delete all PDM aliases by entering UNALIAS *

Like debugger aliases, PDM alias definitions are lost when you exit the PDM. However, individual commands within a PDM command string do not have an expanded-length limit.

For more information about these features, see section 3.1, *Defining Your Own Command Strings*.

The PDM version of the ALIAS command is especially useful for aliasing often-used command strings involving the SEND and SET commands.

- ☐ You can use the ALIAS command to create PDM versions of debugger commands. For example, the ML debugger command lists the memory ranges that are currently defined. To make a PDM version of the ML command to list the memory ranges of all the debuggers in a particular group, enter:

```
alias ml, "send -g %1 ml" 
```

You could then list the memory maps of a group of processors such as those in group GROUPA:

```
m1 GROUPA 
```

- ☐ The ALIAS command can be helpful if you frequently change the default group. For example, suppose you plan to switch between two groups. You can set up the following alias:

```
alias switch, "set dgroup %1; set prompt %1" 
```

The %1 parameter will be filled in with the group information that you enter when you execute SWITCH. Notice that the %1 parameter is preceded by a dollar sign (\$) to set up the default group. The dollar sign tells the PDM to evaluate (take the list of processor names defined in the group instead of the actual group name). However, to change the prompt, you do not want the PDM to evaluate (use the processors associated with the group name as the prompt)—you just want the group name. As a result, you do not need to use the dollar sign when you want to use only the group name.

Assume that GROUP3 contains CPU_A, CPU_B, and CPU_D. To make GROUP3 the current default group and make the PDM prompt the same name as your default group, enter:

```
switch GROUP3 
```

This causes the default group (dgroup) to contain CPU_A, CPU_B, and CPU_D, and it changes the PDM prompt to GROUP3:x>>.

11.6 Entering Operating-System Commands

The SYSTEM command provides you with a method of entering operating-system commands. The format for the PDM version of this command is:

```
system operating-system command
```

The SYSTEM command is similar to the debugger's SYSTEM command (described on page 3-5), but there are some differences.

- ☐ **Similarities.** You can enter operating-system commands without having to leave the primary environment (in this case, the PDM) and without having to open another operating-system window.
- ☐ **Differences.** Unlike the debugger version of the SYSTEM command:
 - The PDM version of the SYSTEM command cannot be entered without an operating-system command parameter. Therefore, you cannot use the command to open a shell.
 - There is no flag parameter; command output is always displayed in the PDM window.

11.7 Understanding the PDM's Expression Analysis

The PDM analyzes expressions differently than individual debuggers do (expression analysis for the debugger is described in Chapter 13, *Basic Information About C Expressions*). The PDM uses a simple integral expression analyzer. You can use expressions to cause the PDM to make decisions as part of the @ command and the flow control commands (described on pages 11-19 and 11-10, respectively).

You cannot evaluate string variables with the PDM expression analyzer. You can evaluate only constant expressions.

Table 11-1 summarizes the PDM operators. The PDM interprets the operators in the order in which they are listed in Table 11-1 (left to right, top to bottom).

Table 11-1. PDM Operators

Operator	Definition	Operator	Definition
()	take highest precedence	*	multiplication
/	division	%	modulo
+	addition (binary)	—	subtraction (binary)
< <	left shift	~	complement
<	less than	> >	right shift
>	greater than	< =	less than or equal to
= =	is equal to	> =	greater than or equal to
&	bitwise AND	! =	is not equal to
	bitwise OR	^	bitwise exclusive-OR
	logical OR	&&	logical AND

11.8 Using System Variables

You can use the SET, @, and UNSET commands to create, modify, and delete system variables. In addition, you can use the SET command with system-defined variables.

Creating your own system variables

The SET command lets you create system variables that you can use with PDM commands. The syntax for the SET command is:

```
set [variable name [= string] ]
```


The *variable name* can consist of up to 128 alphanumeric characters or underscore characters.

For example, suppose you have an array that you want to examine frequently. You can use the SET command to define a system variable that represents that array value:

```
set result = ar1[0] + 100 
```

In this case, result is the variable name, and ar1[0] + 100 is the expression that will be evaluated whenever you use the variable result.

Once you have defined result, you can use it with other PDM commands, such as the SEND command:


```
send CPU_D ? $result 
```

The dollar sign (\$) tells the PDM to replace result with ar1[0] + 100 (the string defined in result) as the expression parameter for the ? command. You *must* precede the name of a system variable with a \$ when you want to use the string value you defined with the variable as a parameter.


You can also use the SET command to concatenate and substitute strings.

Concatenating strings


The dollar sign followed by a system variable name enclosed in braces ({ and }) tells the PDM to append the contents of the variable name to a string that precedes or follows the braces. For example:

```
set k = Hel 
```

Set k to the string Hel.

```
set i = ${k}lo ${k}en 
```

Concatenate the contents of k before lo and en, and set the result to i.

```
echo $i   
Hello Helen
```

Show the contents of i.

❑ Substituting strings

You can substitute defined system variables for parts of variable names or strings. This series of commands illustrates the substitution feature:

```
set err0 = 25  ⓘ           Set err0 to 25.
set j = 0  ⓘ             Set j to 0.
echo $err$j  ⓘ           Show the value of $err$j → $err0 → 25.
25
```

Substitution stops when the PDM detects recursion (for example, \$k = k).

Assigning a variable to the result of an expression

The @ (substitute) command is similar to the SET command. You can use the @ command to assign the result of an expression to a variable. The syntax for the @ command is:

@ variable name = expression

The following series of commands illustrates the differences between the @ command and the SET command. Assume that mask1 equals 36 and mask2 equals 47.

```
set mask3 = $mask1+$mask2  ⓘ   Set mask3 to the contents of mask1
                                plus the contents of mask2.
echo $mask3  ⓘ               Show the contents of mask3.
36+47

@ mask3 = $mask1+$mask2  ⓘ     Set mask3 to the result of the
                                expression $mask1+$mask2.
echo $mask3  ⓘ               Show the contents of mask3.
83
```

Notice the difference between the two commands. The SET command lets you create system variables that you can use with PDM commands. The @ command evaluates the expression and assigns the result to the variable name.

The @ command is useful in setting loop counters. For example, you can initialize a counter with the following command:

```
@ j = 0  ⓘ
```

Inside the loop, you can increment the counter with the following statement:

```
@ j = $j + 1  ⓘ
```

Changing the PDM prompt

The PDM recognizes a system variable called prompt. You can change the PDM prompt by setting the prompt variable to a string. For example, to change the PDM prompt to 3PROC's, enter:

```
set prompt = 3PROC's  ⓘ
```

After entering this command, the PDM prompt will look like this: 3PROC's:x>>.

Checking the execution status of the processors

In addition to displaying the execution status of a processor or group of processors, the STAT command (described on page 11-8) sets a system variable called status.

- ☐ If *all* of the processors in the specified group are running, the status variable is set to 1.
- ☐ If one or more of the processors in the group is halted, the status variable is set to 0.

You can use this variable when you want an instruction loop to execute until a processor halts:


```
loop stat == 1
send ?pc
.
.
```

Listing system variables

To list all system variables, use the SET command without parameters:

```
set 
```

You can also list the contents of a single variable. For example,

```
set j 
j "100"
```

Deleting system variables

To delete a system variable, use the UNSET command. The format for this command is:

```
unset variable name
```

If you want to delete all of the variables you have created and any groups you have defined (as described on page 11-4), use the UNSET command with an asterisk instead of a variable name:

```
unset * 
```

Note:

When you use UNSET * to delete all of your system variables and processor groups, variables such as prompt, status, and dgroup are also deleted.

11.9 Evaluating Expressions

The debugger includes an EVAL command that evaluates an expression (see section 7.3, *Basic Commands for Managing Data*, for more information about the debugger version of the EVAL command). The PDM has a similar command called EVAL that you can send to a processor or a group of processors. The EVAL command evaluates an expression in a debugger and sets a variable to the result of the expression. The syntax for the PDM version of the EVAL command is:

eval **[-g {group | processor name}]** *variable name*=*expression*[, *format*]

- ☐ The **-g** option specifies the group or processor that EVAL should be sent to. If you do not use this option, the command is sent to the default group (dgroup).
- ☐ When you send the EVAL command to more than one processor, the PDM takes the *variable name* that you supply and appends a suffix for each processor. The suffix consists of the underscore character (**_**) followed by the name that you assigned the processor. That way, you can differentiate between the resulting variables.
- ☐ The *expression* can be any expression that uses the symbols described in section 11.7.
- ☐ When you use the optional *format* parameter, the value that the variable is set to will be in one of the following formats:

Parameter	Format	Parameter	Format
*	Default for the data type	o	Octal
c	ASCII character (bytes)	p	Valid address
d	Decimal	s	ASCII string
e	Exponential floating point	u	Unsigned decimal
f	Decimal floating point	x	Hexadecimal

Suppose the program that CPU_A is running has two variables defined: j is equal to 5, and k is equal to 17. Also assume that the program that CPU_B is running contains variables j and k: j is equal to 12, and k is equal to 22.

```
set dgroup = CPU_A CPU_B
eval val = j + k
set
dgroup      "CPU_A CPU_B"
val_CPU_A  "22"
val_CPU_B  "34"
```

Notice that the PDM created a system variable for each processor: val_CPU_A for CPU_A and val_CPU_B for CPU_B.

Summary of Commands

This chapter describes the basic debugger and PDM commands and profiling commands.

Topic	Page
12.1 Functional Summary of Debugger Commands	12-2
12.2 Alphabetical Summary of Debugger and PDM Commands	12-11
12.3 Summary of Profiling Commands	12-62

12.1 Functional Summary of Debugger Commands

This section summarizes the debugger commands according to these categories:

- ☐ **Managing multiple debuggers.** These commands allow you to group debuggers, run code on multiple processors, and send commands to a group of debuggers.
- ☐ **Changing modes.** These commands (listed on page 12-4) allow you to switch freely between the debugging modes (auto, mixed, and assembly).
- ☐ **Managing windows.** These commands (listed on page 12-4) allow you to make a window active and move or resize the active window.
- ☐ **Displaying and changing data.** These commands (listed on page 12-5) allow you to display and evaluate a variety of data items.
- ☐ **Performing system tasks.** These commands (listed on page 12-6) allow you to perform several system functions and provide you with some control over the target system.
- ☐ **Managing breakpoints.** These commands (listed on page 12-7) provide you with a command line method for controlling software breakpoints.
- ☐ **Displaying files and loading programs.** These commands (listed on page 12-4) allow you to change the displays in the File and Disassembly windows and to load object files into memory.
- ☐ **Customizing the screen.** These commands (listed on page 12-4) allow you to customize the debugger display, then save and later reuse the customized displays.
- ☐ **Memory mapping.** These commands (listed on page 12-7) allow you to define the areas of target memory that the debugger can access.
- ☐ **Running programs.** These commands (listed on page 12-8) provide you with a variety of methods for running your programs in the debugger environment.
- ☐ **Profiling commands.** These commands (listed on page 12-9) allow you to collect execution statistics for your code.
- ☐ **Memory system analysis commands (simulator only).** These commands (listed on page 12-10) allow you to set up analysis for events supported by the simulator core.

Managing multiple debuggers

To do this...	Use this command...	See page...
Use the command history	!	12-12
Assign a variable to the result of an expression	@	12-13
Define a custom command string	alias	12-14
Record the information shown in the PDM display area	dlog	12-20
Display a string to the PDM display area	echo	12-21
Evaluate an expression in a debugger or group of debuggers and set a variable to the result	eval	12-22
List available PDM commands	help	12-25
View the description of a PDM command	help	12-25
List the last twenty commands	history	12-26
Conditionally execute PDM commands	if/elif/else/endif	12-26
Loop through PDM commands	loop/break/continue/endloop	12-28
Pause the PDM	pause	12-36
Halt code execution	pesc	12-36
Global halt	phalt	12-37
Run code globally	prun	12-40
Run free globally	prunf	12-41
Single-step globally	pstep	12-41
Exit any debugger and/or the PDM	quit	12-42
Send a command to an individual processor or a group of processors	send	12-46
Change the PDM prompt	set	12-47
Create your own system variables	set	12-47
Define or modify a group of processors	set	12-47
List all system variables or groups of processors	set	12-47
Set the default group	set	12-47
Invoke an individual debugger	spawn	12-51
Find the execution status of a processor or a group of processors	stat	12-53
Enter an operating-system command	system	12-54
Execute a batch file	take	12-55
Delete an alias definition	unalias	12-56
Delete a group or system variable	unset	12-56

Changing modes

To put the debugger in...	Use this command...	See page...
Assembly mode	asm	12-14
Auto mode for debugging C code	c	12-16
Mixed mode	mix	12-33

Managing windows

To do this...	Use this command...	See page...
Reposition a window	move	12-34
Resize a window	size	12-50
Make a window active	win	12-60
Make a window as large as possible	zoom	12-61

Customizing the screen

To do this...	Use this command...	See page...
Change the command-line prompt	prompt	12-40
Load and use a previously saved custom screen configuration	sconfig	12-46
Save a custom screen configuration	ssave	12-52

Displaying files and loading programs

To do this...	Use this command...	See page...
Display a text file in a File window	file	12-23
Load an object file and its symbol table	load	12-28
Load only the object-code portion of an object file	reload	12-42
Load only the symbol-table portion of an object file	sload	12-50

Displaying and changing data

To do this...	Use this command...	See page...
Evaluate and display the result of a C expression	?	12-11
Display C and/or assembly language code at a specific point	addr	12-13
Display the Calls window	calls	12-16
Display assembly language code at a specific address	dasm	12-19
Display the values in an array or structure, or display the value that a pointer is pointing to	disp	12-19
Evaluate a C expression without displaying the results	eval	12-22
Display a specific line in the File window	line	12-28
Display a specific C function	func	12-24
Change the range of memory displayed in the Memory window or display an additional Memory window	mem	12-32
Change the format for displaying data values	setf	12-49
Display the current debugger version	version	12-58
Continuously display the value of a variable, register, or memory location within the Watch window	wa	12-58
Delete a data item from the Watch window	wd	12-59
Show the type of a data item	whatis	12-60
Delete all data items from the Watch window	wr	12-61

Performing system tasks

To do this...	Use this command...	See page...
Define your own command string	alias	12-14
Change the current working directory from within the debugger environment	cd, chdir	12-17
Clear all displayed information from the display area of the Command window	cls	12-17
List the contents of the current directory or any other directory	dir	12-19
Record the information shown in the display area of the Command window	dlog	12-20
Display a string to the Command window while executing a batch file	echo	12-21
Display a help topic for a debugger command	help	12-25
Conditionally execute debugger commands in a batch file	if/else/endif	12-27
Loop debugger commands in a batch file	loop/endloop	12-29
Pause the execution of a batch file	pause	12-36
Exit the debugger	quit	12-42
Reset communication with the emulator	reconnect	12-42
Reset the target system	reset	12-43
Associate a beeping sound with the display of error messages	sound	12-51
Enter any operating-system command or exit to a system shell	system	12-54
Execute commands from a batch file	take	12-55
Delete an alias definition	unalias	12-56
Name additional directories that can be searched when you load source files	use	12-57

Managing breakpoints

To do this...	Use this command...	See page...
Add a software breakpoint	ba	12-15
Delete a software breakpoint	bd	12-15
Display a list of all the software breakpoints that are set	bl	12-15
Reset (delete) all software breakpoints	br	12-16

Memory mapping

To do this...	Use this command...	See page...
Initialize a block of memory word by word	fill	12-24
Initialize a block of memory byte by byte	fillb	12-24
Add an address range to the memory map	ma	12-30
Enable or disable memory mapping	map	12-31
Connect a simulated I/O port to an input or output file (simulator only)	mc	12-31
Delete an address range from the memory map	md	12-32
Disconnect a simulated I/O port (simulator only)	mi	12-33
Display a list of the current memory map settings	ml	12-34
Reset the memory map (delete all range definitions)	mr	12-34
Save a block of memory to a system file	ms	12-35
Connect an input file to the pin (simulator only)	pinc	12-37
Disconnect the input file from the pin (simulator only)	pind	12-38
List the pins that are connected to the input files (simulator only)	pinl	12-38

Running programs

To do this..	Use this command...	See page...
Single-step through assembly language or C code, one C statement at a time; step over function calls	cnext	12-18
Single-step through assembly language or C code, one C statement at a time	cstep	12-18
Run a program up to a certain point	go	12-25
Halt the target system	halt	12-25
Single-step through assembly language or C code; step over function calls	next	12-35
Reset the target system	reset	12-43
Reset the program to its entry point	restart	12-43
Execute code in a function and return to the function's caller	return	12-43
Run a program	run	12-44
Run a program with benchmarking—count the number of CPU clock cycles consumed by the executing portion of code	runb	12-45
Disconnect the emulator from the target system and run free	runf	12-45
Single-step through assembly language or C code	step	12-53
Execute commands from a batch file	take	12-55

Profiling commands

All of the profiling commands can be entered from the Profile menu and associated dialog boxes. In many cases, using the Profile menu and dialog boxes is the easiest way to enter some of these commands. For this reason and also because there are over 100 profiling commands, most of these commands are not described individually in this chapter (as the basic debugger commands are).

Listed below are some of the profiling commands that you might choose to enter from the command line; these commands are also described in the alphabetical command summary. The remaining profiling commands are summarized in section 12.3, *Summary of Profiling Commands*, on page 12-62.

To do this...	Use this command...	See page...
Run a full profiling session	pf	12-36
Run a quick profiling session	pq	12-38
Resume a profiling session	pr	12-39
Switch to profiling environment	profile	12-39
Add a stopping point	sa	12-45
Delete a stopping point	sd	12-46
List all the stopping points	sl	12-50
Delete all the stopping points	sr	12-52
Save all the profile data to a file	vaa	12-57
Save currently displayed profile data to a file	vac	12-57
Reset the display in the Profile window to show all areas and the default set of data	vr	12-58

Memory system analysis commands (simulator only)

Most of the memory system analysis commands can be entered from the analysis menu and dialog box. However, you might want to create a batch file that sets up your most frequently used analysis settings when the debugger is invoked.

Listed below are the memory system analysis commands that are available for you to enter either on the command line or in a batch file.

To do this...	Use this command...	See page...
Enable memory system analysis	ee	9-10
Disable memory system analysis or a specified event	ed	9-11
Configure an event as a break event	eb	9-11
Configure an event as a count event	ecs	9-11
Reset the counters for all events or for a specified event	ecr	9-12
List the configuration for a specified event or all events in the Command window	el	9-12
Disable all events and remove any event configurations that are set	er	9-12

12.2 Alphabetical Summary of Debugger and PDM Commands

There are three types of debugger commands:

- ☐ Basic debugger commands
- ☐ Parallel Debug Manager (PDM) commands that allow you to control multiple debuggers
- ☐ Profiler commands that allow you to control the debugger profiling environment

Most of the commands can be used in the basic debugger environment and/or the profiling environment. Other commands can be used only by the parallel debug manager (PDM). Some commands can be used in more than one environment; other commands can be used in only one of the environments. Each command description identifies the applicable environments for the command.

Commands are not case sensitive; to emphasize this, command names are shown in both uppercase and lowercase throughout this book.

?	<i>Evaluate Expression</i>
Syntax	? <i>expression</i> [, <i>display format</i>]
Menu selection	none
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The ? (evaluate expression) command evaluates an expression and shows the result in the display area of the Command window. The <i>expression</i> can be any C expression, including an expression with side effects; however, you cannot use a string constant or function call in the <i>expression</i>.</p> <p>If the result of <i>expression</i> is not an array or structure, then the debugger displays the results in the Command window. If <i>expression</i> is a structure or array, ? displays the entire contents of the structure or array; you can halt long listings by pressing ESC .</p>

When you use the optional *display format* parameter, data is displayed in one of the following formats:

Parameter	Result is displayed in...	Parameter	Result is displayed in...
*	Default for the data type	o	Octal
c	ASCII character (bytes)	p	Valid address
d	Decimal	s	ASCII string
e	Exponential floating point	u	Unsigned decimal
f	Decimal floating point	x	Hexadecimal

Use the PDM Command History

Syntax

!*{prompt number | string}*
!!

Menu selection

none

Toolbar selection

none

Environments

☐ basic debugger ☒ PDM ☐ profiling

Description

The PDM supports a command history that is similar to the UNIX command history. The PDM prompt identifies the number of the current command. This number is incremented with every command. The PDM command history allows you to reenter any of the last twenty commands.

- ☐ The *number* parameter is the number of the PDM prompt that contains the command that you want to reenter.
- ☐ The *string* parameter tells the PDM to execute the last command that began with *string*.
- ☐ The !! command tells the PDM to execute the last command that you entered.

@*Substitute Result of an Expression***Syntax****@** *variable name* = *expression***Menu selection**

none

Toolbar selection

none

Environments
☐ basic debugger

 ☒ PDM

 ☐ profiling
Description

Unlike the SET command, the @ command first evaluates the *expression*, and then sets the *variable name* to the result. The *expression* can be any expression that uses the symbols described in section 11.7, *Understanding the PDM's Expression Analysis*, on page 11-17. The *variable name* can consist of up to 128 alphanumeric characters or underscore characters.

addr*Display Code at Specified Address***Syntax****addr** {*address* | *function name*}**Menu selection**

none

Toolbar selection

none

Environments
☒ basic debugger

 ☐ PDM

 ☐ profiling
Description

Use the ADDR command to display C code or the disassembly at a specific point. ADDR's behavior changes depending on the current debugging mode:

- ☐ In assembly mode, ADDR works like the DASM command, positioning the code starting at *address* or at *function name* as the first line of code in the Disassembly window.
- ☐ In a C display, ADDR works like the FUNC command, displaying the code starting at *address* or at *function name* as the first line of code in the File window.
- ☐ In mixed mode, ADDR affects both the Disassembly and File windows by displaying code starting at *address* or at *function name* as the first line of code in the Disassembly and File window.

Note:

ADDR affects the File window only if the specified *address* is in a C function.

alias*Define Custom Command String*

Syntax**alias** [*alias name* [, "*command string*"]]**Menu selection**Setup→Alias Commands**Toolbar selection**

none

Environments☒ basic debugger ☒ PDM ☒ profiling**Description**

You can use the ALIAS command to associate one or more debugger or PDM commands with a single *alias name*.

- ☐ The debugger version of the ALIAS command allows you to associate one or more debugger commands with a single *alias name*.
- ☐ The PDM version of the ALIAS command allows you to associate one or more PDM commands with a single *alias name* *or* associate one or more debugger commands with a single *alias name*.

You can include as many commands in the *command string* as you like, as long you separate them with semicolons and enclose the entire string of commands in quotation marks. Also, you can identify command parameters by a percent sign followed by a number (%1, %2, etc.). The total number of characters for an individual command (expanded to include parameter values) is limited to 132. (This restriction applies to the debugger version of the ALIAS command only.)

Previously defined alias names can be included as part of the definition for a new alias.

You can find the current definition of an alias by entering the ALIAS command with the *alias name* only. To see a list of all defined aliases, enter the ALIAS command with no parameters.

asm*Enter Assembly Mode*

Syntax**asm****Menu selection**Mode→Assembly**Toolbar selection**

none

Environments☒ basic debugger ☐ PDM ☐ profiling**Description**

The ASM command changes from the current debugging mode to assembly mode. If you are already in assembly mode, the ASM command has no effect.

ba*Add Software Breakpoint*

Syntax**ba** *address***Menu selection**

Setup→Breakpoints

Toolbar selection**Environments**
☒ basic debugger

 ☐ PDM

 ☒ profiling
Description

The BA command sets a software breakpoint at a specific *address*. The *address* can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label.

You can set breakpoints in program memory (RAM) only; the *address* parameter is treated as a program-memory address.

bd*Delete Software Breakpoint*

Syntax**bd** *address***Menu selection**

Setup→Breakpoints

Toolbar selection**Environments**
☒ basic debugger

 ☐ PDM

 ☒ profiling
Description

The BD command clears a software breakpoint at a specific *address*. The *address* can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label.

bl*List Software Breakpoints*

Syntax**bl****Menu selection**

Setup→Breakpoints

Toolbar selection**Environments**
☒ basic debugger

 ☐ PDM

 ☒ profiling
Description

The BL command lists all the software breakpoints that are currently set in your program. It displays a table of breakpoints in the display area of the Command window. BL lists all the breakpoints that are set in the order in which you set them.

br*Reset Software Breakpoint*

Syntax**br****Menu selection**Setup→Breakpoints**Toolbar selection****Environments**

basic debugger



PDM



profiling

Description

The BR command clears all software breakpoints that are set.

c*Enter Auto Mode*

Syntax**c****Menu selection**Mode→C (Auto)**Toolbar selection**

none

Environments

basic debugger



PDM



profiling

Description

The C command changes from the current debugging mode to auto mode. If you are already in auto mode, the C command has no effect.

calls*Opens Calls Window*

Syntax**calls****Menu selection**View→Call Stack Window**Toolbar selection**

none

Environments

basic debugger



PDM



profiling

Description

The CALLS command displays the Calls window. The debugger displays this window automatically when you are in auto/C or mixed mode. However, you can close the Calls window; the CALLS command opens the window again.

cd, chdir*Change Directory*

Syntax

cd [*directory name*]
chdir [*directory name*]

Menu selection

none

Toolbar selection

none

Environments

☒ basic debugger ☐ PDM ☒ profiling

Description

The CD or CHDIR command changes the current working directory from within the debugger. You can use relative pathnames as part of the *directory name*. If you do not use a *directory name*, the CD command displays the name of the current directory. You can also use the CD command to change the current drive. For example,

```
cd c:
cd d:\csource
cd c:\asmsrc
```

cls*Clear Screen*

Syntax

cls

Menu selection

none

Toolbar selection

none

Environments

☒ basic debugger ☐ PDM ☒ profiling

Description

The CLS command clears all displayed information from the display area of the Command window.

cnext*Single-Step C, Next Statement*

Syntax**cnext** [*expression*]**Menu selection**

Target→Next C

Toolbar selection**Environments**

basic debugger



PDM



profiling

Description

The CNEXT command is similar to the CSTEP command. It runs a program one C statement at a time, updating the display after executing each statement. If you are using CNEXT to step through assembly language code, the debugger does not update the display until it has executed all assembly language statements associated with a single C statement. Unlike CSTEP, CNEXT steps over function calls rather than stepping into them—you do not see the single-step execution of the function call.

The *expression* parameter specifies the number of statements that you want to single-step. You can use a conditional *expression* for conditional single-step execution. (Section 6.4, *Running Code Conditionally*, page 6-11, discusses this in detail.)

cstep*Single-Step C*

Syntax**cstep** [*expression*]**Menu selection**

Target→Step C

Toolbar selection**Environments**

basic debugger



PDM



profiling

Description

The CSTEP single-steps through a program one C statement at a time, updating the display after executing each statement. If you are using CSTEP to step through assembly language code, the debugger does not update the display until it has executed all assembly language statements associated with a single C statement.

The *expression* parameter specifies the number of statements that you want to single-step. You can use a conditional *expression* for conditional single-step execution. (Section 6.4, *Running Code Conditionally*, page 6-11, discusses this in detail.)

dasm*Display Disassembly at Specific Address***Syntax****dasm** {*address* | *function name*}**Menu selection**

none

Toolbar selection

none

Environments
☒ basic debugger

 ☐ PDM

 ☒ profiling
Description

The DASM command displays code beginning at a specific point within the Disassembly window.

dir*List Directory Contents***Syntax****dir** [*directory name*]**Menu selection**

none

Toolbar selection

none

Description

The DIR command displays a directory listing in the display area of the Command window. If you use the optional *directory name* parameter, the debugger displays a list of the specified directory's contents. If you do not use the parameter, the debugger lists the contents of the current directory.

disp*Add Structure, Array, or Pointer to Watch Window***Syntax****disp** *expression* [, *display format*]**Menu selection**

none

Toolbar selection

none

Environments
☒ basic debugger

 ☐ PDM

 ☐ profiling
Description

The DISP command opens a Watch window to display the contents of one of the following:

- ☐ An array
- ☐ A structure
- ☐ Pointer expressions to a scalar type (of the form **pointer*)

If the *expression* is not one of these types, then DISP acts like a ? command.

When the Watch window is open, you can display the data pointed to by a pointer or display the members of the array or structure by clicking the box icon next to watched item:



When you use the optional *display format* parameter, data is displayed in one of the following formats:

Parameter	Result is displayed in...	Parameter	Result is displayed in...
*	Default for the data type	o	Octal
c	ASCII character (bytes)	p	Valid address
d	Decimal	s	ASCII string
e	Exponential floating point	u	Unsigned decimal
f	Decimal floating point	x	Hexadecimal

You can use the *display format* parameter only when you are displaying a scalar type, an array of scalar type, or an individual member of an aggregate type.

You can also use the DISP command with a typecast expression to display memory contents in any format. Here are some examples:

```
disp *0
disp *(float *)123
disp *(char *)0x111
```

This shows memory in the Watch window as an array of locations; the location that you specify with the *expression* parameter is member [0], and all other locations are offset from that location.

dlog

Record Display Area

Syntax

```
dlog filename [{a | w}]
or
dlog close
```

Menu selection

File→Log File

Toolbar selection

none

Environments

☒ basic debugger ☒ PDM ☒ profiling

Description

The DLOG command allows you to record the information displayed in the Command window or in the PDM display area into a log file and to record all commands that you enter from the command line, from the toolbar, from the menus, or with function keys.

To begin a recording session or in the display area of the PDM, use:

dlog *filename*

To end the recording session, enter:

dlog close 

You can write over existing log files or append additional information to existing files. The optional parameters of the DLOG command control how existing log files are used:

- ☐ **Appending to an existing file.** Use the **a** parameter to open an existing file and append the information in the display area to the information already in the file.
- ☐ **Writing over an existing file.** Use the **w** parameter to open an existing file and write over the current contents of the file. This is the default action if you specify an existing filename without using either the **a** or **w** options; you will lose the contents of an existing file if you do not use the append (a) option.

echo	Echo String to Display Area	Batch File Only
Syntax	echo <i>string</i>	
Menu selection	none	
Toolbar selection	none	
Environments	<input checked="" type="checkbox"/> basic debugger	<input checked="" type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The ECHO command displays <i>string</i> in the display area of the Command window or in the display area of the PDM. You cannot use quote marks around the <i>string</i>, and any leading blanks in your command string are removed when the ECHO command is executed.</p> <ul style="list-style-type: none"><input type="checkbox"/> You can execute the debugger version of the ECHO command only in a batch file.<input type="checkbox"/> You can execute the PDM version of the ECHO command in a batch file or from the command line.	

elif	Test for Alternate Condition	Batch File Only
Description	<p>ELIF provides an alternative test by which you can execute PDM commands in the IF/ELIF/ELSE/ENDIF command sequence. See page 12-26 for more information about these commands.</p>	

else	<i>Execute Alternative Commands</i>	Batch File Only
Description	ELSE provides an alternative list of or PDM commands in the IF/ELSE/ENDIF or IF/ELIF/ELSE/ENDIF command sequences, respectively. See pages 12-26 and 12-27 for more information about these commands.	
endif	<i>Terminate Conditional Sequence</i>	Batch File Only
Description	ENDIF identifies the end of a conditional-execution command sequence begun with an IF command. See pages 12-26 and 12-27 for more information about these commands.	
endloop	<i>Terminate Looping Sequence</i>	Batch File Only
Description	ENDLOOP identifies the end of the LOOP/ENDLOOP command sequence. See pages 12-28 and 12-29 for more information about the LOOP/ENDLOOP commands.	
eval	<i>Evaluate Expression</i>	
Syntax	eval <i>expression</i> e <i>expression</i>	
Menu selection	none	
Toolbar selection	none	
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling	
Description	The EVAL command evaluates an expression like the ? command does <i>but does not show the result</i> in the display area of the Command window. EVAL is useful for assigning values to registers or memory locations in a batch file (where it is not necessary to display the result).	
eval	<i>Evaluate Expression and Set to Variable</i>	PDM Environment
Syntax	eval [-g {group processor name}] <i>variable name=expression[, format]</i>	
Menu selection	none	
Toolbar selection	none	
Environments	<input type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input type="checkbox"/> profiling	
Description	The EVAL command evaluates an expression in a debugger and sets a variable to the result of the expression.	

- ☐ The **-g** option specifies the group or processor that EVAL should be sent to. If you don't use this option, the command is sent to the default group (dgroup).
- ☐ When you send the EVAL command to more than one processor, the PDM takes the *variable name* that you supply and appends a suffix for each processor. The suffix consists of the underscore character (**_**) followed by the name that you assigned the processor.
- ☐ The *expression* can be any expression that uses the symbols described in section 11.7, *Understanding the PDM's Expression Analysis*, on page 11-17.
- ☐ When you use the optional *format* parameter, the value that the variable is set to will be in one of the following formats:

Parameter	Result	Parameter	Result
*	Default for the data type	o	Octal
c	ASCII character (bytes)	p	Valid address
d	Decimal	s	ASCII string
e	Exponential floating point	u	Unsigned decimal
f	Decimal floating point	x	Hexadecimal

file

Display Text File

Syntax

file *filename*

Menu selection

File→Open

Toolbar selection



Environments

☒ basic debugger ☐ PDM ☒ profiling

Description

The FILE command displays the contents of any text file in the File window. This command is intended primarily for displaying C source code. You can view as multiple text files at the same time using multiple File windows.

fill*Fill Memory Word by Word*

Syntax**fill** *address, length, data***Menu selection**Memory→Fill Word**Toolbar selection**

none

Environments☒ basic debugger ☐ PDM ☐ profiling**Environments**☒ basic debugger ☐ PDM ☒ profiling**Description**

The FILL command fills a block of memory word by word with a specified value.

- ☐ The *address* parameter identifies the first address in the block.
- ☐ The *length* parameter defines the number of words to fill.
- ☐ The *data* parameter is the value that is placed in each word in the block.

fillb*Fill Memory Byte by Byte*

Syntax**fillb** *address, length, data***Menu selection**Memory→Fill Byte**Toolbar selection**

none

Environments☒ basic debugger ☐ PDM ☐ profiling**Description**

The FILLB command fills a block of memory byte by byte with a specified value.

- ☐ The *address* parameter identifies the first address in the block.
- ☐ The *length* parameter defines the number of bytes to fill.
- ☐ The *data* parameter is the value that is placed in each byte in the block.

func*Display Function*

Syntax**func** {*function name* | *address*}**Menu selection**

none


Toolbar selection

none

Environments☒ basic debugger ☐ PDM ☒ profiling**Description**

The FUNC command displays a specified C function in the File window. You can identify the function by its name or by an address in the function. FUNC works the same way FILE works, but with FUNC you do not need to identify the name of the file that contains the function.

go	<i>Run to Specified Address</i>
Syntax	go [<i>address</i>]
Menu selection	none
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input type="checkbox"/> profiling
Description	The GO command executes code up to a specific point in your program. If you do not supply an <i>address</i> , then GO acts like a RUN command without an <i>expression</i> parameter.

halt	<i>Halt Target System</i>
Syntax	halt
Menu selection	Target→Halt!
Toolbar selection	
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input type="checkbox"/> profiling
Description	The HALT command halts your program, if you are using a simulator, or halts the target system after you have entered a RUNF command, if you are using an emulator. When you invoke the debugger, it automatically executes a HALT command. If you enter a RUNF, quit the debugger, and later reinvoke the debugger, you will effectively reconnect the emulator to the target system and run the debugger in its normal mode of operation.

help	<i>Display Help Topic for Debugger Command</i>
Syntax	help [<i>debugger command</i>]
Menu selection	none
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	The HELP command opens a help topic that describes the <i>debugger command</i> . If you omit the <i>debugger command</i> , the debugger displays a list of help topics.

help	List PDM Commands	PDM Environment
Syntax	help [<i>command</i>]	
Menu selection	none	
Toolbar selection	none	
Environments	<input type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input type="checkbox"/> profiling	
Description	The HELP command provides a brief description of the requested PDM command. If you omit the <i>command</i> parameter, the PDM lists all of the available PDM commands.	

history	List the Last 20 PDM Commands
Syntax	history
Menu selection	none
Toolbar selection	none
Environments	<input type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input type="checkbox"/> profiling
Description	The HISTORY command displays the last 20 PDM commands that you have entered.

if/elif/else/endif	Conditionally Execute PDM Commands
Syntax	if <i>expression</i> <i>PDM commands</i> [elif <i>expression</i> <i>PDM commands</i> [else <i>PDM commands</i> endif
Menu selection	none
Toolbar selection	none
Environments	<input type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input type="checkbox"/> profiling
Description	These commands allow you to execute PDM commands conditionally in a batch file or from the command line.

- ☐ If the expression for the IF is nonzero, the PDM executes all commands between the IF and ELIF, ELSE, or ENDIF.
- ☐ The ELIF is optional. If the expression for the ELIF is nonzero, the PDM executes all commands between the ELIF and ELSE or ENDIF.
- ☐ The ELSE is optional. If the expressions for the IF and ELIF (if present) are false (zero), the PDM executes the commands between the ELSE and ENDIF.

The IF/ELIF/ELSE/ENDIF can be entered interactively or included in a batch file that is executed by the TAKE command. When you enter IF from the PDM command line, a question mark (?) prompts you for the next entry. The PDM continues to prompt you for input using the ? until you enter ENDIF. After you enter ENDIF, the PDM immediately executes the IF command.

If you are in the middle of interactively entering an IF statement and want to abort it, type **CONTROL C**.

if/else/endif*Conditionally Execute Debugger Commands***Batch File Only****Syntax**

```
if expression
  debugger commands
[else
  debugger commands]
endif
```

Menu selection

none

Toolbar selection

none

Environments

☒ basic debugger ☐ PDM ☒ profiling

Description

These commands allow you to execute debugger commands conditionally in a batch file. If the *expression* is nonzero, the debugger executes the commands between the IF and the ELSE or ENDIF. The ELSE portion of the command sequence is optional.

You can substitute a keyword for the expression. Keywords evaluate to true (1) or false (0). You can use the following keywords with the IF command:

- ☐ **\$\$EMU\$\$** (tests for the emulator version of the debugger)
- ☐ **\$\$SIM\$\$** (tests for the simulator version of the debugger)

The conditional commands work with the following provisions:

- ☐ You can use conditional commands only in a batch file.
- ☐ You must enter each debugger command on a separate line in the file.
- ☐ You cannot nest conditional commands within the same batch file.

line

line	Display the specified line number in the FILE window
Syntax	line line number
Menu selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input type="checkbox"/> profiling
Description	Use the LINE command to view specific lines of code. The LINE command displays the specified <i>line number</i> in the middle of the FILE window. When the <i>line number</i> is already displayed in the FILE window, the LINE command does not affect the display.

load	Load Executable Object File
Syntax	load object filename
Menu selection	File→Load Program
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	The LOAD command loads both an object file and its associated symbol table into memory. In effect, the LOAD command performs both a RELOAD and an SLOAD. If you do not supply an extension, the debugger looks for <i>filename.out</i> . The LOAD command clears the old symbol table and closes any Watch windows.

loop/break/ continue/endloop	Loop Through PDM Commands	Batch File Only
Syntax	loop Boolean expression PDM commands [break] [continue] endloop	
Menu selection	none	
Environments	<input type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input type="checkbox"/> profiling	
Description	The LOOP/BREAK/CONTINUE/ENDLOOP commands allow you to set up a looping situation in a batch file or from the command line. Unlike the debugger	

version of the LOOP/ENDLOOP commands, the PDM version of the LOOP command evaluates only Boolean expressions:

- ☐ If the Boolean expression evaluates to true (1), the PDM executes all commands between the LOOP and BREAK, CONTINUE, or ENDLOOP.
- ☐ If the Boolean expression evaluates to false (0), the loop is not entered.

The optional BREAK command allows you to exit the loop without having to reach the ENDLOOP. This is helpful when you are testing a group of processors and want to exit if an error is detected.

The CONTINUE command, which is also optional, acts as a goto and returns command flow to the enclosing LOOP command. CONTINUE is useful when the part of the loop that follows is complicated; returning to the top of the loop avoids further nesting.

The LOOP/BREAK/CONTINUE/ENDLOOP commands can be entered interactively or included in a batch file that is executed by the TAKE command. When you enter LOOP from the PDM command line, a question mark (?) prompts you for the next entry. The PDM continues to prompt you for input using the ? until you enter ENDLOOP. After you enter ENDLOOP, the PDM immediately executes the LOOP command.

If you are in the middle of interactively entering an LOOP statement and want to abort it, type `CONTROL C`.

loop/endloop	Loop Through Debugger Commands	Batch File Only
Syntax	loop <i>expression</i> <i>debugger commands</i> endloop	
Menu selection	none	
Toolbar selection	none	
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling	
Description	<p>The LOOP/ENDLOOP commands allow you to set up a looping situation in a batch file. These looping commands evaluate in the same method as in the run conditional command expression:</p> <ul style="list-style-type: none"><input type="checkbox"/> If you use an <i>expression</i> that is not Boolean, the debugger evaluates the expression as a loop count.<input type="checkbox"/> If you use a Boolean <i>expression</i>, the debugger executes the command repeatedly as long as the expression is true.	

The LOOP/ENDLOOP commands work under the following conditions:

- ☐ You can use LOOP/ENDLOOP commands only in a batch file.
- ☐ You must enter each debugger command on a separate line in the file.
- ☐ You cannot nest LOOP/ENDLOOP commands within the same file.

ma

Add Block to Memory Map

Syntax

ma *address, length, type*

Menu selection

Memory→Mapping

Toolbar selection

none

Environments

☒ basic debugger ☐ PDM ☒ profiling

Description

The MA command identifies valid ranges of target memory. A new memory range must not overlap an existing entry; if you define a range that overlaps an existing range, the debugger ignores the new range.

- ☐ The *address* parameter defines the starting address of a range in memory. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.
- ☐ The *length* parameter defines the length of the range. This parameter can be any C expression.
- ☐ The *type* parameter identifies the read/write characteristics of the memory range. The *type* must be one of these keywords:

To identify this kind of memory . . .	Use this keyword as the <i>type</i> parameter . . .
Read-only memory	R or ROM
Write-only memory	W or WOM
Read/write memory	R W or RAM
Read-only program memory	PROM
Read/write program memory	PRAM
No-access memory	PROTECT
Input port	INPORT or P R
Output port	OUTPORT or P W
Input/output port	IOPORT or P R W

You can use the INPORT, OUTPORT, and IOPORT type parameters in conjunction with the MC command to simulate I/O ports.

map*Enable/Disable Memory Mapping*

Syntax	map { on off }
Menu selection	<u>M</u> emory→ <u>M</u> apping
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The MAP command enables or disables memory mapping. Disabling memory mapping can cause bus fault problems in the target because the debugger may attempt to access nonexistent memory.</p> <p>When you disable memory mapping with the simulator, you can still access memory locations. However, the debugger does not prevent you from accessing memory locations that you have not defined as valid in the memory map.</p>

mc*Connect Simulated I/O Port to a File***Simulator Only**

Syntax	mc <i>port address, length, filename</i>
Menu selection	none
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The MC command connects to an input or output file. Before you can connect the port, you must add it to the memory map with the MA command.</p> <ul style="list-style-type: none"> <input type="checkbox"/> The <i>port address</i> parameter defines the address. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. The <i>address</i> must be the starting address of a block. <input type="checkbox"/> The <i>length</i> parameter defines the length of the range. This parameter can be any C expression. <input type="checkbox"/> The <i>filename</i> parameter can be any filename. If you connect a port to read from a file, the file must exist or the MC command will fail.

md*Delete Block From Memory Map*

Syntax**md** *address***Menu selection**Memory→Mapping**Toolbar selection**

none

Environments

basic debugger



PDM



profiling

Description

The MD command deletes a range of memory from the debugger's memory map.

The *address* parameter identifies the starting address of the range of memory. If you supply an *address* that is not the starting address of a range, the debugger displays this error message in the display area of the Command window:

```
Specified map not found
```

Note:

If you want to use the MD command to remove a simulated I/O port, you must first disconnect the port with the MI command.

mem*Modify Memory Window Display*

Syntax**mem** *expression* [, [*display format*] [, *window name*]]**Menu selection**

none

Toolbar selection

none

Environments

basic debugger



PDM



profiling

Description

The MEM command identifies a new starting address for the block of memory displayed in the Memory window. The optional *window name* parameter opens an additional Memory window, allowing you to view a separate block of memory. The debugger displays the contents of memory at *expression* in the first data position in the Memory window. The end of the range is defined by the size of the window. The *expression* can be an absolute address, a symbolic address, or any C expression.

When you use the optional *display format* parameter, memory is displayed in one of the following formats:

Parameter	Result is displayed in...	Parameter	Result is displayed in...
*	Default for the data type	o	Octal
c	ASCII character (bytes)	p	Valid address
d	Decimal	u	Unsigned decimal
e	Exponential floating point	x	Hexadecimal
f	Decimal floating point		

mi

Disconnect I/O Port

Simulator Only

Syntax

mi port address

Menu selection

none

Toolbar selection

none

Environments

☒ basic debugger

☐ PDM

☒ profiling

Description

The MI command disconnects a simulated I/O port from its associated system file.

The *port address* parameter identifies the address of the I/O port, which must be defined previously with the MC command.

mix

Enter Mixed Mode

Syntax

mix

Menu selection

Mode→Mixed

Toolbar selection

none

Environments

☒ basic debugger

☐ PDM

☐ profiling

Description


The MIX command changes from the current debugging mode to mixed mode. If you are already in mixed mode, the MIX command has no effect.





ml	List Memory Map
Syntax	ml
Menu selection	Memory→Mapping
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	The ML command lists the memory ranges that are defined for the debugger's memory map. The ML command lists the starting address, ending address, and read/write characteristics of each defined memory range.

move	Move a Window
Syntax	move <i>window name</i> [, [<i>X position</i>] [, [<i>Y position</i>] [, [<i>width</i>] [, <i>length</i>]]]
Menu selection	none
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The MOVE command moves the upper left corner of the window to the specified XY position, repositioning the rest of the window relative to that corner. If you choose, you can resize the window while you move it (see the SIZE command for valid <i>width</i> and <i>length</i> values). Specify the <i>X position</i>, <i>Y position</i>, <i>width</i>, and <i>length</i> parameters in pixels. If you omit these parameters, the MOVE command defaults to the window's current position and size.</p> <p>You can spell out the entire <i>window name</i>, but you need to specify only enough letters to identify the window.</p>

mr	Reset Memory Map
Syntax	mr
Menu selection	none
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	The MR command resets the debugger's memory map by deleting all defined memory ranges from the map.

ms	<i>Save Memory Block to File</i>
Syntax	ms <i>address, length, filename</i>
Menu selection	<u>M</u> emory→ <u>S</u> ave
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The MS command saves the values in a block of memory to a system file; files are saved in COFF format.</p> <ul style="list-style-type: none"> <input type="checkbox"/> The <i>address</i> parameter identifies the first address in the block. <input type="checkbox"/> The <i>length</i> parameter defines the length, in words, of the block. This parameter can be any C expression. <input type="checkbox"/> The <i>filename</i> is a system file. If you do not supply an extension, the debugger adds a .obj extension.

next	<i>Single-Step, Next Statement</i>
Syntax	next [<i>expression</i>]
Menu selection	<u>T</u> arget→ <u>N</u> ext
Toolbar selection	
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input type="checkbox"/> profiling
Description	<p>The NEXT command is similar to the STEP command. If you are in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time. Unlike STEP, NEXT never updates the display when executing called functions; NEXT always steps to the next consecutive statement. Unlike STEP, NEXT steps over function calls rather than stepping into them—you do not see the single-step execution of the function call.</p> <p>The optional <i>expression</i> parameter specifies the number of statements that you want to single-step. You can use a conditional <i>expression</i> for conditional single-step execution. (Section 6.4, <i>Running Code Conditionally</i>, page 6-11, discusses this in detail.)</p>

pause	Pause Execution	Batch File Only
Syntax	pause	
Menu selection	none	
Toolbar selection	none	
Environments	<input checked="" type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input type="checkbox"/> profiling	
Description	<p>The PAUSE command allows you to pause the debugger or PDM while running a batch file or executing a flow control command. Pausing is especially helpful in debugging the commands in a batch file.</p> <p>When the debugger or PDM reads this command in a batch file or during a flow control command segment, the debugger/PDM stops execution and displays a dialog box. To continue processing, click OK or press .</p>	
pesc	Send ESC Key to Debuggers	
Syntax	pesc [-g {group processor name}]	
Menu selection	none	
Toolbar selection	none	
Environments	<input type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input type="checkbox"/> profiling	
Description	<p>The PESC command sends the  key to an individual debugger or to a group of debuggers. PESC halts program execution, but all processors in a group do not halt at the same real time; individual processors halt in the order in which they were added to the group.</p> <p>The -g option identifies the group or processor that the command should be sent to. If you do not use this option, the  key is sent to the default group (dgroup).</p>	
pf	Profile, Full	
Syntax	pf starting point [, update rate]	
Menu selection	Profile→Run	
Toolbar selection		
Environments	<input type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling	
Description	<p>The PF command initiates a RUN and collects a full set of statistics on the defined areas between the <i>starting point</i> and the first stopping point encoun-</p>	

tered. The *starting point* parameter can be a label, a function name, or a memory address.

The optional *update rate* parameter determines how often the Profile window is updated. The *update rate* parameter can have one of these values:

Value	Description
0	This is the default. Statistics are not updated until the session is halted (although you can force an update by clicking the mouse in the window).
≥ 1	Statistics are updated during the session. A value of 1 means that data is updated as often as possible.

phalt

Halt Processors in Parallel

Syntax

phalt [{-g *group* | *processor name*}]

Menu selection

none

Environments

☐ basic debugger ☒ PDM ☐ profiling

Description

The PHALT command halts one or more processors. If you send a PRUN or PRUNF command to a group or to an individual processor, you can use PHALT to halt the group or the individual processor. Each processor in a group is halted at the same real time. If you do not use the **-g** option to specify a group or a processor name, the PHALT command is sent to the default group (dgroup).

pinc

Connect Pin

Syntax

pinc *pinname, filename*

Menu selection

none

Environments

☒ basic debugger ☐ PDM ☐ profiling

Description


The PINC command connects an input file to interrupt pin.

- ☐ The *pinname* parameter identifies the interrupt pin and must be one of the external interrupt pins (pins 4–7).
- ☐ The *filename* parameter is the name of your input file.

pind


pind	Disconnect Pin
Syntax	pind pinname
Menu selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input type="checkbox"/> profiling
Description	The PIND command disconnects an input file from an interrupt pin. The <i>pinname</i> parameter identifies the interrupt pin and must be one of the external interrupt pins (pins 4–7).

pinl	List Pin
Syntax	pinl
Menu selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input type="checkbox"/> profiling
Description	The PINL command displays all of the pins—unconnected pins first, followed by the connected pins. For a connected pin, the simulator displays the name of the pin and the absolute pathname of the file in the Command window.

pq	Profile, Quick
Syntax	pq starting point [, update rate]
Menu selection	Profile→Run
Toolbar selection	
Environments	<input type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The PQ command initiates a RUN command and collects a subset of the available statistics on the defined areas between the <i>starting point</i> and the first stopping point encountered. PQ is similar to PF, except that PQ does not collect exclusive or exclusive max data.</p> <p>The <i>update rate</i> parameter is the same as for the PF command.</p>

pr

Resume Profiling Session

Syntax	pr [clear data [, update rate]]
Menu selection	Profile→Run
Toolbar selection	
Environments	<input type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The PR command resumes the last profiling session (initiated by PF or PQ), starting from the current program counter.</p> <p>The optional <i>clear data</i> parameter tells the debugger whether or not it should clear out the previously collected data. The <i>clear data</i> parameter can have one of these values:</p>

Value	Description
0	This is the default. The profiler continues to collect data (adding the data to the existing data for the profiled areas) and to use the previous internal profile stacks.
nonzero	All previously collected profile data and internal profile stacks are cleared.

The *update rate* parameter is the same as for the PF and PQ commands.

profile

Switch to Profiling Environment

Syntax	profile
Menu selection	Profile→Profile Mode
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The PROFILE command toggles between the basic debugger and profiling environments. If you enter PROFILE from the basic debugger environment, the debugger switches to the profiling environment. If you enter PROFILE from the profiling environment, the debugger switches to the basic debugger environment.</p>

prompt	Change Command-Line Prompt
Syntax	prompt <i>new prompt</i>
Menu selection	none
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	The PROMPT command changes the command-line prompt. The <i>new prompt</i> can be any string of characters (a semicolon or comma ends the string). The <i>new prompt</i> cannot be longer than 132 characters.

prun	Run Code in Parallel
Syntax	prun [-r] [-g {group processor name}]
Menu selection	none
Toolbar selection	none
Environments	<input type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input type="checkbox"/> profiling
Description	<p>The PRUN command is the basic command for running an entire program. You enter the command from the PDM command line to begin execution at the same real time for an individual processor or a group of processors. The -g option identifies the group or processor that the command should be sent to. If you do not use this option, then code runs on the default group (dgroup). You can use the PHALT command to stop a global run.</p> <p>The -r (return) option for the PRUN command determines when control returns to the PDM command line:</p> <ul style="list-style-type: none"><input type="checkbox"/> Without -r, control is not returned to the command line until each debugger in the group finishes running code. If you want to to break out of a synchronous command and regain control of the PDM command line, press (CONTROL) (C) in the PDM window. This returns control to the PDM command line. However, no debugger executing the command is interrupted.<input type="checkbox"/> With -r, control is returned to the command line immediately, even if a debugger is still executing a command. You can type new commands, but the processors cannot execute the commands until they finish with the current command; however, you can perform PHALT, PESC, and STAT commands when the processors are still executing.

prunf*Run Free in Parallel***Syntax****prunf** [-g {group | processor name}]**Menu selection**

none

Toolbar selection

none

Environments
☐ basic debugger

 ☒ PDM

 ☐ profiling
Description

The PRUNF command starts the processors running free, which means they are disconnected from the emulator. RUNF synchronizes the debuggers to cause the processors to begin execution at the same real time. The **-g** option identifies the group or processor that the command should be sent to. If you do not use this option, then code runs on the default group (dgroup).

The PHALT command stops a PRUNF; note that the debugger automatically executes a PHALT when the debugger is invoked.

pstep*Single-Step in Parallel***Syntax****pstep** [-g {group | processor name}] [count]**Menu selection**

none

Toolbar selection

none

Environments
☐ basic debugger

 ☒ PDM

 ☐ profiling
Description

The PSTEP command single-steps synchronously through assembly language code with interrupts disabled. RUNF synchronizes the debuggers to cause the processors to begin execution at the same real time. The **-g** option identifies the group or processor that the command should be sent to. If you do not use this option, then code runs on the default group (dgroup). You can use the PHALT command to stop a global run.

You can use the *count* parameter to specify the number of statements that you want to single-step.



Note:

If the current statement that a processor is pointing to has a breakpoint, that processor will not step synchronously with the other processors when you use the PSTEP command. However, that processor will still single-step.

quit	Exit Debugger
Syntax	quit
Menu selection	File→E <u>x</u> it
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	The QUIT command exits the debugger and returns to the operating system. If you enter this command from the PDM, the PDM and all debuggers running under the PDM are exited.

reconnect	Reset Communication With Emulator	Emulator Only
Syntax	reconnect	
Menu selection	none	
Toolbar selection	none	
Environments	<input checked="" type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling	
Description	<p>The RECONNECT command reinitializes communication between the debugger and the emulator. This command can be used after an unrecoverable fatal error.</p> <p>Any software breakpoints set before a reconnect may still reside in memory after the reconnect. However, the debugger does not recognize that the breakpoints are set. You should reload memory in order to clear out any residual breakpoints.</p>	

reload	Reload Object Code
Syntax	reload [object filename]
Menu selection	File→R <u>e</u> load Program
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	The RELOAD command loads only an object file <i>without</i> loading its associated symbol table. This is useful for reloading a program when target memory has been corrupted. If you enter the RELOAD command without specifying a filename, the debugger reloads the file that you loaded last.

reset	<i>Reset Target System</i>
Syntax	reset
Menu selection	<u>T</u> arget→ <u>R</u> eset Target
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The RESET command resets the target system (emulator only) or simulator. This is a <i>software</i> reset.</p> <p>If you are using the simulator and execute the RESET command, the simulator simulates the processor and peripheral reset operation, putting the processor in a known state.</p>
restart	<i>Reset PC to Program Entry Point</i>
Syntax	restart rest
Menu selection	<u>T</u> arget→Re <u>s</u> tart
Toolbar selection	
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The RESTART or REST command resets the program to its entry point. (This assumes that you have already used one of the load commands to load a program into memory.)</p>
return	<i>Return to Function's Caller</i>
Syntax	return ret
Menu selection	<u>T</u> arget→Re <u>t</u> urn
Toolbar selection	
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input type="checkbox"/> profiling

Description

The RETURN or RET command executes the code in the current C function and halts when execution reaches the caller. Breakpoints do not affect this command, but you can halt execution by doing one of the following actions:

- ☐ Click the Halt icon on the toolbar:



- ☐ From the Target menu, select Halt!.
- ☐ Press `(ESC)`.

run

Run Code

Syntax

run [*expression*]

Menu selection


Target→Run

Toolbar selection**Environments**

☒ basic debugger ☐ PDM ☐ profiling

Description

The RUN command is the basic command for running an entire program. The command's behavior depends on the type of parameter you supply:

- ☐ If you do not supply an *expression*, the program executes until it encounters a breakpoint or until you do one of the following actions:
 - Click the Halt icon on the toolbar:

 - From the Target menu, select Halt!.
 - Press `(ESC)`.
- ☐ If you supply a logical or relational *expression*, the run becomes conditional. (Section 6.4, *Running Code Conditionally*, page 6-11, discusses this in detail.)
- ☐ If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger executes *count* instructions, halts, and updates the display.

runb	<i>Benchmark Code</i>
Syntax	runb
Menu selection	<u>T</u> arget→ <u>R</u> un Benchmark
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input type="checkbox"/> profiling
Description	The RUNB command executes a specific section of code and counts the number of CPU clock cycles consumed by the execution. For RUNB to operate correctly, <i>execution must be halted by a software breakpoint</i> . After RUNB execution halts, the debugger stores the number of cycles into the CLK pseudoregister. For a complete explanation of the RUNB command and the benchmarking process, read section 6.5, <i>Benchmarking</i> , on page 6-12.
runf	<i>Run Free</i> Emulator Only
Syntax	runf
Menu selection	none
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input type="checkbox"/> profiling
Description	<p>The RUNF command disconnects the emulator from the target system while code is executing. When you enter RUNF, the debugger clears all breakpoints, disconnects the emulator from the target system, and causes the processor to begin execution at the current PC. You can quit the debugger, or you can continue to enter commands. However, any command that causes the debugger to access the target at this time produces an error.</p> <p>The HALT command stops a RUNF; the debugger automatically executes a HALT when the debugger is invoked.</p>
sa	<i>Add Stopping Point</i>
Syntax	sa <i>address</i>
Menu selection	none
Toolbar selection	none
Environments	<input type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	The SA command adds a stopping point at <i>address</i> . The <i>address</i> can be a label, a function name, or a memory address.

sconfig*Load Screen Configuration*

Syntax**sconfig** [*filename*]**Menu selection**File→Load Config**Toolbar selection**

none

Environments☒ basic debugger ☐ PDM ☒ profiling**Description**

The SCONFIG command restores the display to a specified configuration. This restores the window locations and sizes that were saved with the SSAVE command into *filename*. The debugger searches for the specified file in the current directory and then in directories named with the D_DIR environment variable. If you do not supply a *filename*, the debugger looks for init.clr.

When you use SCONFIG to restore a configuration that includes multiple File, Watch, or Memory windows, the additional windows are not opened automatically. However, when you open an additional window and use a *window name* that matches a window name that you used before you saved the configuration, the window is placed in the saved location.

sd*Delete Stopping Point*

Syntax**sd** *address***Menu selection**

none

Toolbar selection

none

Environments☐ basic debugger ☐ PDM ☒ profiling**Description**

The SD command deletes the stopping point at *address*.

send*Send Debugger Command to Individual Debuggers*

Syntax**send** [**-r**] [**-g** {*group* | *processor name*}] *debugger command***Menu selection**

none

Toolbar selection

none

Environments☐ basic debugger ☒ PDM ☐ profiling**Description**

The SEND command sends any debugger command to an individual processor or to a group of processors. If the command produces a message, it is displayed in the COMMAND window for the appropriate debugger(s) and also in the PDM window.

- ☐ The **-g** option specifies the group or processor that the debugger command should be sent to. If you do not use this option, the command is sent to the default group (dgroup).
- ☐ The **-r** (return) option determines when control returns to the PDM command line:
 - **Without -r**, control is not returned to the command line until each debugger in the group finishes running code. Any results that are printed in the COMMAND window of the individual debuggers is also echoed in the PDM command window. These results are displayed by processor.

If you want to break out of a synchronous command and regain control of the PDM command line, press **(CONTROL) (C)** in the PDM window. This returns control to the PDM command line. However, no debugger executing the command is interrupted.

 - **With -r**, control is returned to the command line immediately, even if a debugger is still executing a command. When you use **-r**, you *do not* see the results of the commands that the debuggers are executing.

set*Set a Variable to a String***Syntax**

set [*group name* [= *list of processor names*]]
set [*variable* [= *string value*]]

Menu selection

none

Toolbar selection

none

Environments

☐ basic debugger ☒ PDM ☐ profiling

Description

The SET command allows you to create groups of processors to which you can send commands. With the SET command you can:

- ☐ **Define a group of processors.** It is useful to define a group when you plan to send commands to the same set of processors. The commands are sent to the processors in the same order in which you added the processors to the group. To define a group, specify a *group name* and then list the processors you want in the group.
- ☐ **Set the default group.** Defining a default group provides you with a short-hand method of maintaining members in a group or of sending commands to the same group. To set up the default group, use the SET command with a special *group name* called dgroup.

- ❑ **Modify an existing group or creating a group based on another group.** Once you have created a group, you can add processors to it by using the SET command and preceding the existing *group name* with a dollar sign (\$) in the list of processors. You can also use a group as part of another group by preceding the existing group's name with a dollar sign. The dollar sign tells the PDM to use the processors listed previously in the group as part of the new list of processors.
- ❑ **List all groups of processors.** You can use the SET command without any parameters to list all the processors that belong to a group, in the order in which they were added to the group.

You can also use the SET command with system-defined variables to:

- ❑ **Change the prompt for the PDM.** To change the PDM prompt, use the SET command with the system variable called prompt. For example, to change the PDM prompt to 3PROCs, enter:

```
set prompt = 3PROCs
```
- ❑ **Check the execution status of the processors.** In addition to displaying the execution status of a processor or group of processors, the STAT command (described on page 12-53) sets a system variable called status. If *all* of the processors in the specified group are running, the status variable is set to 1. If one or more of the processors in the group is halted, the status variable is set to 0.

You can use this variable when you want an instruction loop to execute until a processor halts (the LOOP/ENDLOOP command is described on page 12-29).

- ❑ **Create your own system variables.** You can use the SET command to create your own system variables that you can use with PDM commands. For more information about creating your own system variables, see page 11-18.

setf*Set Default Data-Display Format***Syntax****setf** [*data type*, *display format*]**Menu selection**

none

Toolbar selection

none

Environments
☒ basic debugger

 ☐ PDM

 ☐ profiling
Description

The SETF command changes the display format for a specific data type. If you enter SETF with no parameters, the debugger lists the current display format for each data type.

☐ The *data type* parameter can be any of the following C data types:

char	short	uint	ulong	double
uchar	int	long	float	ptr


☐ The *display format* parameter can be any of the following characters:

Parameter	Result is displayed in...	Parameter	Result is displayed in...
*	Default for the data type	o	Octal
c	ASCII character (bytes)	p	Valid address
d	Decimal	s	ASCII string
e	Exponential floating point	u	Unsigned decimal
f	Decimal floating point	x	Hexadecimal

Only a subset of the display formats can be used for each data type. Listed below are the valid combinations of data types and display formats.

Valid Display Formats		Valid Display Formats	
Data Type	c d o x e f p s u	Data Type	c d o x e f p s u
char (c)	√ √ √ √	long (d)	√ √ √ √
uchar (d)	√ √ √ √	ulong (d)	√ √ √ √
short (d)	√ √ √ √	float (e)	√ √ √ √
int (d)	√ √ √ √	double (e)	√ √ √ √
uint (d)	√ √ √ √	ptr (p)	√ √ √ √

To return all data types to their default display format, enter:

setf * 

size	Size a Window
Syntax	size <i>window name</i> [, [<i>width</i>] [, <i>length</i>]]
Menu selection	none
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The SIZE command changes the size of the window. Specify the <i>width</i> and <i>length</i> parameters in pixels. If you omit these parameters, the SIZE command defaults to the window's current size.</p> <p>You can spell out the entire <i>window name</i>, but you need to specify only enough letters to identify the window.</p>

sl	List Stopping Point
Syntax	sl
Menu selection	none
Toolbar selection	none
Environments	<input type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	The SL command lists all of the currently set stopping points.

sload	Load Symbol Table
Syntax	sload <i>object filename</i>
Menu selection	<u>F</u> ile→Load <u>S</u> ymbols
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The SLOAD command loads the symbol table of the specified object file. SLOAD is useful in an emulation environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). In such an environment, loading the symbol table allows you to perform symbolic debugging and examine the values of C variables.</p> <p>SLOAD clears the existing symbol table before loading the new one but does not modify memory or set the program entry point. SLOAD closes any Watch windows.</p>

sound*Enable Error Beeping*

Syntax	sound {on off}
Menu selection	none
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	You can cause a beep to sound every time a debugger error message is displayed. This is useful if the Command window is hidden (because you would not see the error message). By default, sound is off.

spawn*Invoke the 'C6x Debugger*

Syntax	spawn emu6x -n processor name [invocation options]
Menu selection	none
Toolbar selection	none
Environments	<input type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input type="checkbox"/> profiling
Description	<p>You must invoke a debugger for each processor that you want the PDM to control. To invoke a debugger, use the SPAWN command.</p> <ul style="list-style-type: none"> <input type="checkbox"/> emu6x is the executable that invokes the debugger. <p>The PDM associates the <i>processor name</i> with the actual processor according to which executable you use. To invoke a debugger, the PDM must be able to find the executable file for that debugger. The PDM first searches the current directory and then searches the directories listed with the PATH statement.</p> <input type="checkbox"/> -n processor name supplies a processor name. You <i>must</i> use the -n option since the PDM uses processor names to identify the various debuggers that are running. The processor name can consist of up to eight alphanumeric or underscore characters and must begin with an alphabetic character. The name is not case sensitive. The processor name must match one of the names defined in your board configuration file (see Appendix B, <i>Describing Your Target System to the Debugger</i>).

sr	Reset Stopping Point
Syntax	sr
Menu selection	none
Toolbar selection	none
Environments	<input type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	The SR command resets (deletes) <i>all</i> currently set stopping points.

ssave	Save Screen Configuration
Syntax	ssave [filename]
Menu selection	File→Save As Config
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The SSAVE command saves the current screen configuration to a file. This saves the window locations and window sizes for all debugging modes, including the size and location for multiple File, Watch, and Memory windows. However, the debugger does not save docking information about docked windows. If you have one or more docked windows and you save and reload the screen configuration, the debugger does not display any windows as docked. If you want the windows docked, you must follow the docking procedure again.</p> <p>The <i>filename</i> parameter names the screen configuration file. You can include path information (including relative pathnames); if you do not supply path information, the debugger places the file in the current directory. If you do not supply a <i>filename</i>, the debugger saves the current configuration into a file named init.clr and places the file in the current directory.</p> <p>If you use a filename that already exists, the debugger overwrites the file with the current configuration.</p>

stat*Find the Execution Status of Processors*

Syntax**stat** [{-g *group* | *processor name*}]**Menu selection**

none

Toolbar selection

none

Environments☐

basic debugger

☒

PDM

☐

profiling

Description

The STAT command tells you whether a processor is running or halted. If a processor is halted when you execute this command, then the PDM also lists the current PC value for that processor. If you do not use the -g option, the PDM displays the status of the processors in the default group (dgroup).

step*Single-Step*

Syntax**step** [*expression*]**Menu selection**Target→Step**Toolbar selection****Environments**☒

basic debugger

☐

PDM

☐



profiling

Description

The STEP command single-steps through assembly language or C code. If you are in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time.

If you are single-stepping through C code and encounter a function call, the STEP command shows you the single-step execution of the called function (assuming that the function was compiled with the compiler's -g option). When function execution is complete, single-step execution returns to the caller. If the function was not compiled with the -g option, the debugger executes the function but does not show single-step execution of the function.

The *expression* parameter specifies the number of statements that you want to single-step. You can use a conditional *expression* for conditional single-step execution. (Section 6.4, *Running Code Conditionally*, page 6-11, discusses this in detail.)

system	Enter Operating-System Command
Syntax	system [<i>operating-system command</i> [, <i>flag</i>]]
Menu selection	none
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The debugger version of the SYSTEM command allows you to enter operating-system commands without explicitly exiting the debugger environment. If you enter SYSTEM with no parameters, the debugger opens a system shell and displays the operating-system prompt. At this point, you can enter any operating-system command. When you finish, enter:</p> <p>exit </p> <p>If you prefer, you can supply the operating-system command as a parameter to the SYSTEM command. If the result of the command is a message or other display, the debugger blanks the top of the debugger display to show the information. In this case, you can use the <i>flag</i> parameter to tell the debugger whether or not it should hesitate after displaying the information. The <i>flag</i> can be 0 or 1.</p> <p>0 If you supply a value of 0 for <i>flag</i>, the debugger immediately returns to the debugger environment after the last item of information is displayed.</p> <p>1 If you supply a value of 1 for <i>flag</i>, the debugger does not return to the debugger environment until you enter:</p> <p>exit  .</p> <p>(This is the default.)</p>

system	Enter Operating-System Command	PDM Environment
Syntax	system <i>operating-system command</i>	
Menu selection	none	
Toolbar selection	none	
Environments	<input type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input type="checkbox"/> profiling	

Description The PDM version of the SYSTEM command allows you to enter a single operating-system command without explicitly exiting the PDM environment. You cannot enter more than one operating-system command with the PDM version of the SYSTEM command.

take*Execute Batch File*

Syntax Basic debugger: **take** *batch filename* [, *suppress echo flag*]
PDM: **take** *batch filename*

Menu selection File→Take

Toolbar selection none

Environments ☒ basic debugger ☒ PDM ☒ profiling

Description The TAKE command tells the debugger or the PDM to read and execute commands from a batch file. The *batch filename* parameter identifies the file that contains commands. If you do not supply a pathname as part of the filename, the debugger/PDM first looks in the current directory and then searches directories named with the D_DIR environment variable.

The *batch filename* for the PDM version of this command must have a .pdm extension, or the PDM will not be able to read the file. In addition, the batch file that the PDM reads can contain only PDM commands.

By default, the debugger echoes the commands to the display area of the Command window and updates the display as it reads the commands from the batch file. To suppress the echoing and updating, enter a 0 as the *suppress echo flag* parameter. If you omit the *suppress echo flag* parameter or enter a nonzero value for that parameter, the debugger behaves in the default manner.

take_abort*Display Abort Prompt*

Syntax **take_abort** {on | off}

Menu selection none

Toolbar selection none

Environments ☒ basic debugger ☐ PDM ☒ profiling

Description The TAKE_ABORT feature prompts you before continuing a take file when a target error is detected. When the take abort feature is OFF, you do not get an abort prompt. The default is OFF.

unalias	Delete Alias Definition
Syntax	unalias { <i>alias name</i> *}
Menu selection	Setup→Alias Commands
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The UNALIAS command deletes defined aliases.</p> <p><input type="checkbox"/> To delete a single alias, enter the UNALIAS command with an alias name. For example, to delete an alias named NEWMAP, enter:</p> <pre>unalias NEWMAP </pre> <p><input type="checkbox"/> To delete all aliases, enter an asterisk instead of an alias name:</p> <pre>unalias * </pre> <p>The * symbol <i>does not</i> work as a wildcard.</p>

unset	Delete Group
Syntax	unset <i>group name</i> unset *
Menu selection	none
Toolbar selection	none
Environments	<input type="checkbox"/> basic debugger <input checked="" type="checkbox"/> PDM <input type="checkbox"/> profiling
Description	<p>The UNSET command deletes a group of processors. You can use this command in conjunction with the SET command to remove a particular processor from a group.</p> <p>To delete all groups, enter an asterisk instead of a group name:</p> <pre>unset * </pre> <p>The * symbol <i>does not</i> work as a wildcard.</p> <div><p>Note:</p><p>When you use UNSET * to delete all of your system variables and processor groups, variables such as prompt, status, and dgroup are also deleted.</p></div>

use	<i>Use Additional Directory</i>
Syntax	use [directory name]
Menu selection	none
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The USE command allows you to name an additional directory that the debugger can search when looking for source files. You can specify only one directory at a time.</p> <p>If you enter the USE command without specifying a directory name, the debugger lists in the display area of the Command window all of the current directories.</p>
vaa	<i>Save All Profile Data to a File</i>
Syntax	vaa filename
Menu selection	<u>P</u> rofile→Save <u>A</u> ll
Toolbar selection	none
Environments	<input type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The VAA command saves all statistics collected during the current profiling session. The data is stored in a system file.</p>
vac	<i>Save Displayed Profile Data to a File</i>
Syntax	vac filename
Menu selection	<u>P</u> rofile→Save <u>V</u> iew
Toolbar selection	none
Environments	<input type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	<p>The VAC command saves all statistics currently displayed in the Profile window. (Statistics that are not displayed are not saved.) The data is stored in a system file.</p>

version	<i>Display the Current Debugger Version</i>
Syntax	version
Menu selection	none
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	The VERSION command displays the debugger's copyright date and version number, as well as the device name.

vr	<i>Reset Profile Window Display</i>
Syntax	vr
Menu selection	none
Toolbar selection	none
Environments	<input type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input checked="" type="checkbox"/> profiling
Description	The VR command resets the display in the Profile window so that all marked areas are listed and statistics are displayed with default labels and in the default sort order.

wa	<i>Add Item to Watch Window</i>
Syntax	wa expression [, [label] [, [display format] [, window name]]]
Menu selection	<u>S</u> etup→ <u>W</u> atch Variable
Toolbar selection	none
Environments	<input checked="" type="checkbox"/> basic debugger <input type="checkbox"/> PDM <input type="checkbox"/> profiling
Description	The WA command displays the value of <i>expression</i> in a Watch window. If a Watch window is not open, executing WA opens a Watch window. The <i>expression</i> parameter can be any C expression, including an expression that has side effects.

WA is most useful for watching an expression whose value changes over time; constant expressions serve no useful function in the watch window. The *label* parameter is optional. When used, it provides a label for the watched entry. If you do not use a *label*, the debugger displays the *expression* in the label field.

When you use the optional *display format* parameter, data is displayed in one of the following formats:

Parameter	Result is displayed in...	Parameter	Result is displayed in...
*	Default for the data type	o	Octal
c	ASCII character (bytes)	p	Valid address
d	Decimal	s	ASCII string
e	Exponential floating point	u	Unsigned decimal
f	Decimal floating point	x	Hexadecimal

If you want to use a *display format* parameter without a *label* parameter, be sure to include an extra comma. For example:

```
wa PC,,o 
```

You can open additional Watch windows by using the *window name* parameter. When you open an additional Watch window, the debugger appends the *window name* to the Watch window label. You can create as many Watch windows as you need.

If you omit the *window name* parameter, the debugger displays the expression in the default Watch window (labeled Watch).

wd

Delete Item From Watch Window

Syntax

```
wd expression [, window name]
```

Menu selection

```
Setup→Watch Variable
```

Toolbar selection

```
none
```

Environments

☒ basic debugger

☐ PDM

☐ profiling

Description

The WD command deletes a specific item from the Watch window. The WD command's *expression* parameter must correspond to one of the variable names listed in the Watch window. The optional *window name* parameter specifies a particular Watch window. If no window names is given, the expression is deleted from the default Watch window.

whatis

whatis

Find Data Type

Syntax

whatis *symbol*

Menu selection

none

Toolbar selection

none

Environments

☒ basic debugger ☐ PDM ☐ profiling

Description

The WHATIS command shows the data type of *symbol* in the display area of the Command window. The *symbol* can be any variable (local, global, or static), a function name, structure tag, typedef name, or enumeration constant.

win

Make a Window Active

Syntax

win *window name*

Menu selection

View menu options

Toolbar selection

none

Environments

☒ basic debugger ☐ PDM ☒ profiling

Description

The WIN command allows you to make a window active by name. You can spell out the entire window name, but you really need to specify only enough letters to identify the window.

If you supply an ambiguous name (such as C, which could stand for CPU or Calls), the debugger selects the first window it finds whose name matches the name you supplied. If the debugger does not find the window you asked for (because you closed the window or misspelled the name), then the WIN command has no effect.

wr*Close Watch Window***Syntax****wr** [{ * | *window name* }]**Menu selection**Setup→Watch Variable**Toolbar selection**

none

Environments
☒ basic debugger

 ☐ PDM

 ☐ profiling
Description

The WR command deletes all items from a Watch window and closes the window.

- ☐ To close the default Watch window, enter:

wr 

- ☐ To close one of the additional Watch windows, use this syntax:

wr *window name*

- ☐ To close all Watch windows, enter:

wr * **zoom***Zoom a Window***Syntax****zoom** [*window name*]**Menu selection**

none

Toolbar selection

none

Environments
☒ basic debugger

 ☐ PDM

 ☒ profiling
Description

The ZOOM command makes the window as large as possible. To unzoom a window, enter the ZOOM command a second time; this returns the window to its prezoom size and position.

You can spell out the entire *window name*, but you really need to specify only enough letters to identify the window.

12.3 Summary of Profiling Commands

The following tables summarize the profiling commands that are used for marking, enabling, disabling, and unmarking areas and for changing the display in the Profile window. These commands are easiest to use from the Profile menu and associated dialog boxes, so they are not included in the alphabetical command summary. The syntaxes for these commands are provided here so that you can include them in batch files.

Table 12–1. Marking areas

To mark this area...	In C only	In disassembly only
Lines		
<input type="checkbox"/> By line number, address	MCLE <i>filename, line number</i>	MALE <i>address</i>
<input type="checkbox"/> All lines in a function	MCLF <i>function</i>	MALF <i>function</i>
Ranges		
<input type="checkbox"/> By line numbers	MCRE <i>filename, line number, line number</i>	MARE <i>address, address</i>
Functions		
<input type="checkbox"/> By function name	MCFE <i>function</i>	not applicable
<input type="checkbox"/> All functions in a module	MCFM <i>filename</i>	
<input type="checkbox"/> All functions everywhere	MCFG	

Table 12–2. Disabling marked areas

To disable this area...	In C only	In disassembly only	In C and disassembly
Lines			
<input type="checkbox"/> By line number, address	DCLE <i>filename, line number</i>	DALE <i>address</i>	not applicable
<input type="checkbox"/> All lines in a function	DCLF <i>function</i>	DALF <i>function</i>	DBLF <i>function</i>
<input type="checkbox"/> All lines in a module	DCLM <i>filename</i>	DALM <i>filename</i>	DBLM <i>filename</i>
<input type="checkbox"/> All lines everywhere	DCLG	DALG	DBLG
Ranges			
<input type="checkbox"/> By line number, address	DCRE <i>filename, line number</i>	DARE <i>address</i>	not applicable
<input type="checkbox"/> All ranges in a function	DCRF <i>function</i>	DARF <i>function</i>	DBRF <i>function</i>
<input type="checkbox"/> All ranges in a module	DCRM <i>filename</i>	DARM <i>filename</i>	DBRM <i>filename</i>
<input type="checkbox"/> All ranges everywhere	DCRG	DARG	DBRG

To disable this area...	In C only	In disassembly only	In C and disassembly
Functions			
<input type="checkbox"/> By function name	DCFE <i>function</i>	not applicable	not applicable
<input type="checkbox"/> All functions in a module	DCFM <i>filename</i>		DBFM <i>filename</i>
<input type="checkbox"/> All functions everywhere	DCFG		DBFG
All areas			
<input type="checkbox"/> All areas in a function	DCAF <i>function</i>	DAAF <i>function</i>	DBAF <i>function</i>
<input type="checkbox"/> All areas in a module	DCAM <i>filename</i>	DAAM <i>filename</i>	DBAM <i>filename</i>
<input type="checkbox"/> All areas everywhere	DCAG	DAAG	DBAG

Table 12–3. Enabling disabled areas

To enable this area...	In C only	In disassembly only	In C and disassembly
Lines			
<input type="checkbox"/> By line number, address	ECLE <i>filename, line number</i>	EALE <i>address</i>	not applicable
<input type="checkbox"/> All lines in a function	ECLF <i>function</i>	EALF <i>function</i>	EBLF <i>function</i>
<input type="checkbox"/> All lines in a module	ECLM <i>filename</i>	EALM <i>filename</i>	EBLM <i>filename</i>
<input type="checkbox"/> All lines everywhere	ECLG	EALG	EBLG
Ranges			
<input type="checkbox"/> By line number, address	ECRE <i>filename, line number</i>	EARE <i>address</i>	not applicable
<input type="checkbox"/> All ranges in a function	ECRF <i>function</i>	EARF <i>function</i>	EBRF <i>function</i>
<input type="checkbox"/> All ranges in a module	ECRM <i>filename</i>	EARM <i>filename</i>	EBRM <i>filename</i>
<input type="checkbox"/> All ranges everywhere	ECRG	EARG	EBRG
Functions			
<input type="checkbox"/> By function name	ECFE <i>function</i>	not applicable	not applicable
<input type="checkbox"/> All functions in a module	ECFM <i>filename</i>		EBFM <i>filename</i>
<input type="checkbox"/> All functions everywhere	ECFG		EBFG
All areas			
<input type="checkbox"/> All areas in a function	ECAF <i>function</i>	EAAF <i>function</i>	EBAF <i>function</i>
<input type="checkbox"/> All areas in a module	ECAM <i>filename</i>	EAAM <i>filename</i>	EBAM <i>filename</i>
<input type="checkbox"/> All areas everywhere	ECAG	EAAG	EBAG

Table 12–4. Unmarking areas

To unmark this area...	In C only	In disassembly only	In C and disassembly
Lines			
<input type="checkbox"/> By line number, address	UCLE <i>filename, line number</i>	UALE <i>address</i>	not applicable
<input type="checkbox"/> All lines in a function	UCLF <i>function</i>	UALF <i>function</i>	UBLF <i>function</i>
<input type="checkbox"/> All lines in a module	UCLM <i>filename</i>	UALM <i>filename</i>	UBLM <i>filename</i>
<input type="checkbox"/> All lines everywhere	UCLG	UALG	UBLG
Ranges			
<input type="checkbox"/> By line number, address	UCRE <i>filename, line number</i>	UARE <i>address</i>	not applicable
<input type="checkbox"/> All ranges in a function	UCRF <i>function</i>	UARF <i>function</i>	UBRF <i>function</i>
<input type="checkbox"/> All ranges in a module	UCRM <i>filename</i>	UARM <i>filename</i>	UBRM <i>filename</i>
<input type="checkbox"/> All ranges everywhere	UCRG	UARG	UBRG
Functions			
<input type="checkbox"/> By function name	UCFE <i>function</i>	not applicable	not applicable
<input type="checkbox"/> All functions in a module	UCFM <i>filename</i>		UBFM <i>filename</i>
<input type="checkbox"/> All functions everywhere	UCFG		UBFG
All areas			
<input type="checkbox"/> All areas in a function	UCAF <i>function</i>	UAAF <i>function</i>	UBAF <i>function</i>
<input type="checkbox"/> All areas in a module	UCAM <i>filename</i>	UAAM <i>filename</i>	UBAM <i>filename</i>
<input type="checkbox"/> All areas everywhere	UCAG	UAAG	UBAG

Table 12–5. Changing the profile window display

(a) Viewing specific areas

To view this area...	In C only	In disassembly only	In C and disassembly
Lines			
<input type="checkbox"/> By line number, address	VFCLE <i>filename, line number</i>	VFALE <i>address</i>	not applicable
<input type="checkbox"/> All lines in a function	VFCLF <i>function</i>	VFALF <i>function</i>	VFBLF <i>function</i>
<input type="checkbox"/> All lines in a module	VFCLM <i>filename</i>	VFALM <i>filename</i>	VFBLM <i>filename</i>
<input type="checkbox"/> All lines everywhere	VFCLG	VFALG	VFBLG
Ranges			
<input type="checkbox"/> By line number, address	VFCRE <i>filename, line number</i>	VFARE <i>address</i>	not applicable
<input type="checkbox"/> All ranges in a function	VFCRF <i>function</i>	VFARF <i>function</i>	VFBRF <i>function</i>
<input type="checkbox"/> All ranges in a module	VFCRM <i>filename</i>	VFARM <i>filename</i>	VFBRM <i>filename</i>
<input type="checkbox"/> All ranges everywhere	VFCRG	VFARG	VFBRG
Functions			
<input type="checkbox"/> By function name	VFCFE <i>function</i>	not applicable	not applicable
<input type="checkbox"/> All functions in a module	VFCFM <i>filename</i>		VFBFM <i>filename</i>
<input type="checkbox"/> All functions everywhere	VFCFG		VFBFG
All areas			
<input type="checkbox"/> All areas in a function	VFCAF <i>function</i>	VFAAF <i>function</i>	VFBAF <i>function</i>
<input type="checkbox"/> All areas in a module	VFCAM <i>filename</i>	VFAAM <i>filename</i>	VFBAAM <i>filename</i>
<input type="checkbox"/> All areas everywhere	VFCAG	VFAAG	VFBAAG

(b) Viewing different data

(c) Sorting the data

To view this information...	Use this command...	To sort on this data...	Use this command...
Count	VDC	Count	VSC
Inclusive	VDI	Inclusive	VSI
Inclusive, maximum	VDN	Inclusive, maximum	VSN
Exclusive	VDE	Exclusive	VSE
Exclusive, maximum	VDX	Exclusive, maximum	VSX
Address	VDA	Address	VSA
All	VDL	Data	VSD

Basic Information About C Expressions

Many of the debugger commands take C expressions as parameters. This allows the debugger to have a relatively small, yet powerful, instruction set. Because C expressions can have side effects—that is, the evaluation of some types of expressions can affect existing values—you can use the same command to display or to change a value. This reduces the number of commands in the command set.

This chapter contains basic information that helps you use C expressions as debugger command parameters.

Topic	Page
13.1 C Expressions for Assembly Language Programmers	13-2
13.2 Using Expression Analysis in the Debugger	13-4

13.1 C Expressions for Assembly Language Programmers

It is not necessary for you to be an experienced C programmer to use the debugger. However, to use the debugger's full capabilities, you should be familiar with the rules governing C expressions. You should obtain a copy of *The C Programming Language* (first or second edition) by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey. This book is referred to in the C community, and in Texas Instruments documentation, as *K&R*.

Note:

A single value or symbol is a legal C expression.

K&R contains a complete description of C expressions; to get you started, here is a summary of the operators that you can use in expression parameters.

☐ Reference operators

<code>-></code>	indirect structure reference	<code>.</code>	direct structure reference
<code>[]</code>	array reference	<code>*</code>	indirection (unary)
<code>&</code>	address (unary)		

☐ Arithmetic operators

<code>+</code>	addition (binary)	<code>-</code>	subtraction (binary)
<code>*</code>	multiplication	<code>/</code>	division
<code>%</code>	modulo	<code>-</code>	negation (unary)
<code>(type)</code>	type cast		

☐ Relational and logical operators

<code>></code>	greater than	<code>>=</code>	greater than or equal to
<code><</code>	less than	<code><=</code>	less than or equal to
<code>==</code>	is equal to	<code>!=</code>	is not equal to
<code>&&</code>	logical AND	<code> </code>	logical OR
<code>!</code>	logical NOT (unary)		

□ Increment and decrement operators

++	increment	--	decrement
----	-----------	----	-----------

These unary operators can precede or follow a symbol. When the operator precedes a symbol, the symbol value is incremented/decremented before it is used in the expression; when the operator follows a symbol, the symbol value is incremented/decremented after it is used in the expression. Because these operators affect the symbol's final value, the parameters they are used with have side effects.

□ Bitwise operators

&	bitwise AND		bitwise OR
^	bitwise exclusive-OR	<<	left shift
>>	right shift	~	1s complement (unary)

□ Assignment operators

=	assignment	+=	assignment with addition
-=	assignment with subtraction	/=	assignment with division
%=	assignment with modulo	&=	assignment with bitwise AND
^=	assignment with bitwise XOR	=	assignment with bitwise OR
<<=	assignment with left shift	>>=	assignment with right shift
*=	assignment with multiplication		

These operators support a shorthand version of the familiar binary expressions; for example, $X = X + Y$ can be written in C as $X += Y$. Because these operators affect a symbol's final value, the parameters they are used with have side effects.

13.2 Using Expression Analysis in the Debugger

The debugger's expression analysis is based on C expression analysis. This includes all mathematical, relational, pointer, and assignment operators. However, a few limitations, as well as a few additional features, are not described in K&R C.

Restrictions

The following restrictions apply to the debugger's expression analysis features.

- ☐ The sizeof operator is not supported.
- ☐ The comma operator (,) is not supported (commas are used to separate parameter values for the debugger commands).
- ☐ Function calls and string constants are currently not supported in expressions.
- ☐ The debugger supports a limited capability of type casts; the following forms are allowed:

(basic type)

*(basic type * ...)*

([structure/union/enum] structure/union/enum tag)

*([structure/union/enum] structure/union/enum tag * ...)*

You can use up to six * characters in a cast.

Additional features

- ☐ All floating-point operations are performed in double precision using standard widening. (This is transparent.) Floats are represented in IEEE floating-point format.
- ☐ All registers can be referenced by name. The 'C6x auxiliary registers are treated as integers and/or pointers.
- ☐ Void expressions are legal (treated like integers).
- ☐ The specification of variables and functions can be qualified with context information. Local variables (including local statics) can be referenced with the expression form:

function name.local name

This expression format is useful for examining the automatic variables of a function that is not currently being executed. Unless the variable is static, however, the function must be somewhere in the current call stack. If you want to see local variables from the currently executing function, you need not use this form; you can simply specify the variable name (just as in your C source).

File-scoped variables (such as statics or functions) can be referenced with the following expression form:

filename.function name
or *filename.variable name*

This expression format is useful for accessing a file-scoped static variable (or function) that may share its name with variables in other files.

In this expression form, *filename* **does not include** the file extension; the debugger searches the object symbol table for any source filename that matches the input name, disregarding any extension. Thus, if the variable *ABC* is in file *source.c*, you can specify it as *source.ABC*.

These expression forms can be combined into an expression of the form:

filename.function name.variable name

- Any integral or void expression can be treated as a pointer and used with the indirection operator (*). Here are several examples of valid use of a pointer in an expression:

*123
*A5
*(A2 + 123)
*(I*J)

By default, the values are treated as integers (that is, these expressions point to integer values).

- Any expression can be type cast to a pointer to a specific type (overriding the default of pointing to an integer, as described above).

Hint: You can use casting with the WA and DISP commands to display data in a desired format.

For example, the expression:

*(float *)10

treats 10 as a pointer to a floating-point value at location 10 in memory. In this case, the debugger fetches the contents of memory location 10 and treats the contents as a floating-point value. If you use this expression as a parameter for the DISP command, the debugger displays memory contents as an array of floating-point values within the DISP window, beginning with memory location 10 as array member [0].

Note how the first expression differs from the expression:

```
(float)*10
```

In this case, the debugger fetches an integer from address 10 and converts the integer to a floating-point value.

You can also type cast to user-defined types such as structures. For example, in the expression:

```
((struct STR *)10)->field
```

the debugger treats memory location 10 as a pointer to a structure of type STR (assuming that a structure is at address 10) and accesses a field from that structure.

What the Debugger Does During Invocation

In some circumstances, you may find it helpful to know the steps that the debugger goes through during the invocation process. These are the steps, in order, that the debugger performs. If you are using PDM to run multiple debuggers, PDM executes the first step. (For more information on the environment variables mentioned below, see Chapter 2, *Getting Started With the Debugger*.)

The debugger:

- 1) Reads options from the operating system's command line.
- 2) Reads any information specified with the `D_OPTIONS` environment variable.
- 3) Reads information from the `D_DIR` and `D_SRC` environment variables.
- 4) Looks for the `init.clr` screen-configuration file.
(The debugger searches for the screen-configuration file in directories named with `D_DIR`.)
- 5) Initializes the debugger screen and windows.
- 6) Finds the batch file that defines your memory map by searching in directories named with `D_DIR`. The debugger expects this file to set up the memory map and follows these steps to look for the batch file:
 - ☐ When you invoke the debugger, it checks to see if you have used the `-t` debugger option. If it finds the `-t` option, the debugger reads and executes the specified file.
 - ☐ If you have not used the `-t` option, the debugger looks for the default initialization batch file. The batch file name differs for each version of the debugger:
 - For the emulator, this file is named *emuinit.cmd*.
 - For the simulator, this file is named *siminit.cmd*.

If the debugger finds the file corresponding to your tool, it reads and executes the file. If the debugger does not find the `-t` option or the initialization batch file, it looks for a file called `init.cmd`. This allows you to have one initialization batch file for more than one debugger tool. To set up this file, you can use the `IF/ELSE/ENDIF` commands (see page 3-8 for more information) to indicate which memory map applies to each tool.

- 7) Loads any object files specified with `D_OPTIONS` or specified on the command line during invocation.
- 8) Determines the initial mode (auto, assembly, or mixed) and displays the appropriate windows on the screen.

At this point, the debugger is ready to process any commands that you enter.

Where the debugger looks for files

You can perform all load-type commands by using menu options. However, if you choose to use the command-line equivalents to these menu options, you need to know where the debugger looks for source files.

The `FILE`, `LOAD`, `RELOAD`, `SLOAD`, `SCONFIG`, and `TAKE` commands expect a filename as a parameter. If the filename includes path information, the debugger uses the file from the specified directory and does not search for the file in any other directory. If you do not supply path information, the debugger must search for the file. The debugger first looks for the file in the current directory. You may, however, have your files in several different directories.

- ☐ If you are using `LOAD`, `RELOAD`, or `SLOAD`, you have only two choices for supplying the path information:

- Specify the path as part of the filename.
- Alternatively, you can use the `CD` command before you enter the `LOAD`, `RELOAD` or `SLOAD` command to change the current directory from within the debugger. The format for this command is:

cd *directory name*

- ☐ If you are using the `FILE` command, you have several options:

- Within the operating-system environment, you can name additional directories with the `D_SRC` environment variable. The format for this environment variable is:

SET D_SRC=pathname;pathname For PCs

setenv D_SRC "pathname;pathname" For SPARCstations

You can name several directories for the debugger to search.

- When you invoke the debugger, you can use the `-i` option to name additional source directories for the debugger to search. The format for this option is `-i pathname`.

You can specify multiple pathnames by using several `-i` options (one pathname per option). The list of source directories that you create with `-i` options is valid until you quit the debugger.

- Within the debugger environment, you can use the `USE` command to name additional source directories. The format for this command is:

use *directory name*

You can specify only one directory at a time.

In all cases, you can use relative pathnames such as `..\csource` or `..\..\code`. The debugger can recognize a cumulative total of 20 paths specified with `D_SRC`, `-i`, and `USE`.

Describing Your Target System to the Debugger

For the debugger to understand how you have configured your target system, you must supply the target configuration information in a file for the debugger to read.

- ❑ If you are using an emulation scan path that contains only one 'C6x and no other devices, you can use the *board.dat* file that comes with the 'C6x emulator kit. This file describes to the debugger the single 'C6x in the scan path and gives the 'C6x the name CPU_A. Because the debugger automatically looks for a file called *board.dat* in the current directory and in the directories specified with the D_DIR environment variable, you can skip this appendix.
- ❑ If you plan to use a target system that has multiple 'C6x devices or that includes devices other than the 'C6x, you must follow these steps:

Step 1: Create the board configuration text file.

Step 2: Translate the board configuration text file to a binary, structured format so that the debugger can read it.

Step 3: Specify the formatted configuration file when invoking the debugger.

These steps are described in this appendix.

Topic	Page
B.1 Step 1: Create the Board Configuration Text File	B-2
B.2 Step 2: Translate the Configuration File to a Debugger-Readable Format	B-5
B.3 Step 3: Specify the Configuration File When Invoking the Debugger	B-6

B.1 Step 1: Create the Board Configuration Text File

To describe the emulation scan path of your target system to the debugger, you must create a board configuration file. Each entry of the file describes one device on your scan path and the entries follow the order of the devices in the scan path. The text version of the configuration file is referred to as *board.cfg* in this book.

Example B–1 shows a board.cfg file that describes a possible 'C6x device chain. It lists six octals named A1–A6, followed by five 'C6x devices named CPU_A, CPU_B, CPU_C, CPU_D, and CPU_E.

Example B–1. A Sample TMS320C6x Device Chain

(a) A sample board.cfg file

Device Name	Device Type	Comments
"A1 "	BYPASS08	;the first device nearest TDO ;(test data out)
"A2 "	BYPASS08	;the next device nearest TDO
"A3 "	BYPASS08	
"A4 "	BYPASS08	
"A5 "	BYPASS08	
"A6 "	BYPASS08	
"CPU_A"	TMS320C6x	;the first 'C6x
"CPU_B"	TMS320C6x	
"CPU_C"	TMS320C6x	
"CPU_D"	TMS320C6x	
"CPU_E"	TMS320C6x	;the last 'C6x nearest TDI ;(test data in)

(b) A sample 'C6x device chain



The order in which you list each device is important. The emulator scans the devices, assuming that the data from one device is followed by the data of the next device on the chain. Data from the device that is closest to the emulation header's TDO (test data out) reaches the emulator first. The device whose data reaches the emulator first is listed first in the board.cfg file; the device whose data reaches the emulator last is listed last in the board.cfg file.

The board.cfg file can have any number of each of these three types of entries:

- ☐ **Debugger devices** such as the 'C6x. These are the only devices that the debugger can recognize.
- ☐ The **TI ACT8997 scan path linker**, or **SPL**. The SPL allows you to have up to four secondary scan paths that can each contain debugger devices ('C6xs) and other devices.
- ☐ **Other devices**. These are any other devices in the scan path. These devices cannot be debugged and must be worked around or bypassed when trying to access the 'C6xs.

Each entry in the board.cfg file consists of at least two pieces of data:

- ☐ **The name of the device.** The device name always appears first and is enclosed in double quotes:

"device name"

This is the same name that you use with the `-n` debugger option, which tells the debugger the name of the 'C6x. The *device name* can consist of up to eight alphanumeric characters or underscore characters and must begin with an alphabetic character.

- ☐ **The type of the device.** The debugger supports the following device types:

- ☐ **TMS320C6x** is an example of a debugger-device type. TMS320C6x describes the 'C6x.

- ☐ **SPL** specifies the scan path linker and must be followed by four subpaths, as in this syntax:

"device name" **SPL** {*subpath0*} {*subpath1*} {*subpath2*} {*subpath3*}

Each *subpath* can contain any number of devices. However, an SPL subpath *cannot* contain another SPL. A subpath that contains no devices must still be listed.

Example B–2 shows a file that contains an SPL.

Example B–2. A board.cfg File Containing an SPL

Device Name	Device Type	Comments
"A1"	BYPASS08	;the first device nearest TDO
"A2"	BYPASS08	
"CPU_A"	TMS320C6x	;the first 'C6x
"HUB"	SPL	;the scan path linker
{		;the first subpath
"B1"	BYPASS08	
"B2"	BYPASS08	
"CPU_B"	TMS320C6x	;the second 'C6x
}		
{		;the second subpath
"C1"	BYPASS08	
"C2"	BYPASS08	
"CPU_C"	TMS320C6x	;the third 'C6x
}		
{		;the third subpath (contains nothing)
}		
{		;the fourth subpath
"D1"	BYPASS08	
"D2"	BYPASS08	
"CPU_D"	TMS320C6x	;the fourth 'C6x
}		
"CPU_E"	TMS320C6x	;the last 'C6x nearest TDI

Note: The indentation in the file is for readability only.

B.2 Step 2: Translate the Configuration File to a Debugger-Readable Format

After you have created the `board.cfg` file, you must translate it from text to a binary, conditioned format so that the debugger can understand it. To translate the file, use the `composer` utility that is included with the emulator kit. At the system prompt, enter the following command:

composer *[input file [output file]]*

- ☐ The *input file* is the name of the `board.cfg` file that you created in step 1; if the file is not in the current directory, you must supply the entire path-name. If you omit the input filename, the `composer` utility looks for a file called `board.cfg` in your current directory.
- ☐ The *output file* is the name that you can specify for the resulting binary file; ideally, use the name `board.dat`. If you want the output file to reside in a directory other than the current directory, you must supply the entire path-name. If you omit an output filename, the `composer` utility creates a file called `board.dat` and places it in the current directory.

To avoid confusion, use a `.cfg` extension for your text filenames and a `.dat` extension for your binary filenames. If you enter only one filename on the command line, the `composer` utility assumes that it is an input filename.

B.3 Step 3: Specify the Configuration File When Invoking the Debugger

When you invoke a debugger (either from the PDM or at the system prompt), the debugger must be able to find the `board.dat` file so that it knows how you have set up your scan path. The debugger looks for the `board.dat` file in the current directory and in the directories named with the `D_DIR` environment variable.

If you used a name other than `board.dat` or if the `board.dat` file is not in the current directory or in a directory named with `D_DIR`, you must use the `-f` option when you invoke the debugger. The `-f` option allows you to specify a board configuration file (and pathname) to be used instead of `board.dat`. The format for this option is:

`-f` *filename*

Debugger Messages

This appendix contains an alphabetical listing of the progress and error messages that the debugger or PDM might display in the display area of the Command window or in the PDM display area. Each listing contains both a description of the situation that causes the message and an action to take if the message indicates a problem or error.

Topic	Page
C.1 Associating Sound With Error Messages	C-2
C.2 Alphabetical Summary of Debugger Messages	C-2
C.3 Alphabetical Summary of PDM Messages	C-22
C.4 Additional Instructions for Expression Errors	C-26
C.5 Additional Instructions for Hardware Errors	C-26

C.1 Associating Sound With Error Messages

You can associate a beeping sound with the display of error messages. To do this, use the SOUND command. The format for this command is:

sound {on | off}

By default, no beep is associated with error messages (SOUND OFF). The beep is helpful if the Command window is hidden behind other windows.

If you are using the debugger with Windows 95 or Windows NT, you must be sure that you have sound enabled in the control panel.

C.2 Alphabetical Summary of Debugger Messages

']' expected

<i>Description</i>	This is an expression error—it means that the parameter contained an opening bracket symbol but did not contain a closing bracket symbol.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

'}' expected

<i>Description</i>	This is an expression error—it means that the parameter contained an opening parenthesis symbol but did not contain a closing parenthesis symbol.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

A

Aborted by user

<i>Description</i>	The debugger halted a long Command display listing because you pressed the ESC key.
<i>Action</i>	None required; this is normal debugger behavior.

B**Breakpoint already exists at *address***

<i>Description</i>	During single-step execution, the debugger attempted to set a breakpoint where one already existed. (This is not necessarily a breakpoint that you set—it may have been an internal breakpoint that the debugger set for single-stepping).
<i>Action</i>	None should be required; you may want to reset the program entry point (Target→Restart) and reenter the single-step command.

Breakpoint table full

<i>Description</i>	200 breakpoints are already set, and there was an attempt to set another. The maximum limit of 200 breakpoints includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.
<i>Action</i>	Open the Breakpoint Control dialog box by selecting Breakpoints from the Setup menu. Delete individual software breakpoints.

C**Cannot allocate host memory**

<i>Description</i>	This is a fatal error—it means that the debugger is running out of memory.
<i>Action</i>	You can invoke the debugger with the <code>-v</code> option so that fewer symbols may be loaded, or you can relink your program and link in fewer modules at a time.

Cannot allocate system memory

<i>Description</i>	This is a fatal error—it means that the debugger is running out of memory.
<i>Action</i>	You can invoke the debugger with the <code>-v</code> option so that fewer symbols may be loaded, or you can relink your program and link in fewer modules at a time.

Cannot connect file to program memory

<i>Description</i>	An attempt has been made to connect a file to program memory using the MC command.
<i>Action</i>	You cannot connect a file to any location in program memory using the MC command.

Cannot detect target power

<i>Description</i>	This hardware error occurs after the emurst command is reset. Follow the steps described below and then restart your emulator.
<i>Action</i>	<ul style="list-style-type: none"><input type="checkbox"/> Check the emulator board to be sure it is installed snugly.<input type="checkbox"/> Check the cable connecting your emulator and target system to be sure it is not loose.<input type="checkbox"/> Check your target board to be sure it is getting the correct voltage.<input type="checkbox"/> Check your emulator scan path to be sure it is uninterrupted.<input type="checkbox"/> Ensure that your port address is set correctly:<ul style="list-style-type: none">■ Check to be sure the <code>-p</code> option used with the <code>D_OPTIONS</code> environment variable matches the I/O address defined by your switch settings. (See page 2-5 for more information on the <code>D_OPTIONS</code> environment variable.)■ Check to see if you have a conflict in address space with another bus setting. If you have a conflict, change the switches on your board to one of the alternate settings listed in the installation guide. Modify the <code>-p</code> option of the <code>D_OPTIONS</code> environment variable to reflect the change in your switch settings.

Cannot edit field

<i>Description</i>	Expressions that are displayed in the Watch window cannot be edited.
<i>Action</i>	If you attempted to edit an expression in the Watch window, you may have actually wanted to change the value of a symbol or register used in the expression. Use the <code>?</code> or <code>EVAL</code> command to edit the actual symbol or register. The expression value is automatically updated.

Cannot find/open initialization file

<i>Description</i>	The debugger cannot find the init.cmd file.
<i>Action</i>	Be sure that init.cmd is in the appropriate directory. If it is not, copy it from the debugger product diskette. If the file is already in the correct directory, verify that the D_DIR environment variable is set up to identify the directory. See the information about setting up the debugger environment information included with your installation instructions.

Cannot halt the processor

<i>Description</i>	This is a fatal error—for some reason, pressing ESC did not halt program execution.
<i>Action</i>	Exit the debugger. Invoke the autoexec.bat file, then invoke the debugger again.

Cannot initialize target system

<i>Description</i>	This error occurs while you are invoking the debugger with the emulator. A variety of events may cause this error to occur.
<i>Action</i>	<ul style="list-style-type: none"> <input type="checkbox"/> Check the cable connecting the emulator to the target system to be sure it is not loose. <input type="checkbox"/> Ensure that your port address is set correctly: <ul style="list-style-type: none"> ■ Check to be sure the <code>-p</code> option used with the D_OPTIONS environment variable matches the I/O address defined by your switch settings. ■ Check to see if you have a conflict in address space with another bus setting. If you have a conflict, change the switches on your board to one of the alternate settings listed in the installation guide. Modify the <code>-p</code> option of the D_OPTIONS environment variable to reflect the change in your switch settings. <input type="checkbox"/> Check the end of your autoexec.bat or initdb.bat file for the emurst.exe command. Execute this command <i>after</i> powering up the target board. See section 2.5 on page 2-6.

For more information on setting up the D_OPTIONS environment variable, see page 2-5 .

Cannot map into reserved memory: ?

<i>Description</i>	The debugger tried to access unconfigured/reserved/nonexistent memory.
<i>Action</i>	Remap the reserved memory accesses.

Cannot map port address

<i>Description</i>	You attempted to do a connect/disconnect on an illegal port address.
<i>Action</i>	Be sure that you are connecting to or disconnecting from an address that is mapped in as an input, output, or I/O port.

Cannot open config file

<i>Description</i>	The SCONFIG command cannot find the screen-customization file that you specified. The debugger also displays this message when you try to load a screen-customization file that was saved by an older version of the debugger.
<i>Action</i>	<ul style="list-style-type: none"><input type="checkbox"/> Be sure that the filename was typed correctly. If it was not, reenter the command with the correct name. If it was, reenter the command and specify full path information with the filename.<input type="checkbox"/> Be sure that the screen-customization file was saved using the current version of the debugger rather than an older version of the debugger.

Cannot open “filename”

<i>Description</i>	The debugger attempted to show <i>filename</i> in the File window but could not find the file.
<i>Action</i>	Be sure that the file exists as named. If it does, enter the USE command to identify the file's directory.

Cannot open new window

<i>Description</i>	A maximum of 127 windows can be open at once. The last request to open a window would have made 128, which is not possible.
<i>Action</i>	Close any unnecessary windows. Windows that can be closed include Watch, File, Calls, and Memory windows. To close any of these windows, make the desired window active and press CONTROL F4 .

Cannot open object file: “filename”

<i>Description</i>	The file specified with the LOAD, SLOAD, or RELOAD command is not an object file that the debugger can load.
<i>Action</i>	Be sure that you are loading an actual object file. Be sure that the file was linked. You may want to run cl6x (with the -z option) or lnk6x again to create an executable object file.

Cannot read processor status

<i>Description</i>	This is a fatal error—for some reason, pressing ESC did not halt program execution.
<i>Action</i>	Exit the debugger. Invoke the autoexec.bat file, then invoke the debugger again. If you are using the emulator, check the cable connections, also.

Cannot reset the processor

<i>Description</i>	This is a fatal error—for some reason, pressing ESC did not halt program execution.
<i>Action</i>	Exit the debugger. Invoke the autoexec.bat file, then invoke the debugger again. If you are using the emulator, check the cable connections, also.

Cannot restart processor

<i>Description</i>	The debugger attempted to reset the PC to the program entry point, but the program does not seem to have an entry point.
<i>Action</i>	Do not use Target→Restart or RESTART if your program does not have an explicit entry point.

Cannot set/verify breakpoint at address

<i>Description</i>	Either you attempted to set a breakpoint in read-only or protected memory, or there are hardware problems with the target system. This may also happen when you enable or disable on-chip memory while using breakpoints.
<i>Action</i>	Check your memory map. If the address that you wanted to breakpoint was not in ROM, see section C.5, <i>Additional Instructions for Hardware Errors</i> , page C-26.

Cannot step

<i>Description</i>	There is a problem with the target system.
<i>Action</i>	See section C.5, <i>Additional Instructions for Hardware Errors</i> , page C-26.

Cannot take address of register

<i>Description</i>	This is an expression error. C does not allow you to take the address of a register.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Command “*command*” not found

<i>Description</i>	The debugger did not recognize the command that you typed.
<i>Action</i>	Reenter the correct command. See Chapter 12, <i>Summary of Commands</i> .

Command timed out, emulator busy

<i>Description</i>	There is a problem with the target system.
<i>Action</i>	See section C.5, <i>Additional Instructions for Hardware Errors</i> , page C-26.

Conflicting map range

<i>Description</i>	A block of memory specified with the Memory→Mapping menu option or the MA command overlaps an existing memory map entry. Blocks cannot overlap.
<i>Action</i>	Use Memory→Mapping or the ML command to list the existing memory map; this helps you find that existing block that the new block would overlap. If the existing block is not necessary, delete it with the Memory Map Control dialog box or with the MD command. Use the Memory Map Control dialog box or the MA command to redefine the block of memory. If the existing block is necessary, use the Memory Map Control dialog box or the MA command to define a range that does not overlap the existing block.

Corrupt call stack

<i>Description</i>	The debugger tried to update the Calls window and could not. This message is displayed in the following situations: <ul style="list-style-type: none"> <input type="checkbox"/> A function was called that did not return. <input type="checkbox"/> The program stack was overwritten in target memory. <input type="checkbox"/> You are debugging code that has optimization enabled (for example, you did not use the <code>-g</code> compile option); if this is the case, ignore this message—code execution is not affected.
<i>Action</i>	If your program called a function that did not return, then this is normal behavior (as long as you intended for the function not to return). Otherwise, you may be overwriting program memory.

E

Emulator I/O address is invalid

<i>Description</i>	The debugger was invoked with the <code>-p</code> option, and an invalid <i>port address</i> was used.
<i>Action</i>	For valid <i>port address</i> values, see page 2-12.

EOF reached —connected at port: <memory add>

<i>Description</i>	The last data of the input file has been read.
<i>Action</i>	You can disconnect the file with the MI command and connect a new file with the MC command. If you do not do anything and resume execution, then the input file automatically rewinds, and input data is read from the beginning of the file.

Error in expression

<i>Description</i>	This is an expression error.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Execution error

<i>Description</i>	There is a problem with the target system.
<i>Action</i>	See section C.5, <i>Additional Instructions for Hardware Errors</i> , page C-26.

F

File already tied to port

Description You attempted to connect to an address that already has a file connected to it.

Action Connect the file to a mapped port that is not connected to a file.

File already tied to this pin

Description You attempted to connect an input file to an interrupt pin that already has a file connected to it.

Action Use the PINC command to connect the file to another interrupt pin that is not connected to a file.

File does not exist

Description The port file could not be opened for reading.

Action Be sure that the file exists as named. If it does, enter the USE command to identify the file's directory.

Files must be disconnected from ports

Description You attempted to delete a memory map that has files connected to it.

Action You must disconnect a port with the MI command before you can delete it from the memory map.

File not found

Description The filename specified for the FILE command was not found in the current directory or any of the directories identified with D_SRC.

Action Be sure that the filename was typed correctly. If it was, reenter the FILE command and specify full path information with the filename.

File not found : “filename”

<i>Description</i>	The filename specified for the LOAD, RELOAD, SLOAD, or TAKE command was not found in the current directory or any of the directories identified with D_SRC.
<i>Action</i>	Be sure that the filename was typed correctly. If it was, reenter the command and specify full path information with the filename.

File too large (filename)

<i>Description</i>	You attempted to load a file that exceeded the maximum loadable COFF file size.
<i>Action</i>	Loading the file without the symbol table (SLOAD), or use cl6x (with the -z option) or lnk6x to relink the program with fewer modules.

Float not allowed

<i>Description</i>	This is an expression error—a floating-point value was used incorrectly.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Function required

<i>Description</i>	The parameter for the FUNC command must be the name of a function in the program that is loaded.
<i>Action</i>	Reenter the FUNC command with a valid function name.

I**Illegal cast**

<i>Description</i>	This is an expression error—the expression parameter uses a cast that does not meet the C language rules for casts.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Illegal left hand side of assignment

<i>Description</i>	This is an expression error—the left-hand side of an assignment expression does not meet C language assignment rules.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Illegal memory access

<i>Description</i>	Your program tried to access unmapped memory.
<i>Action</i>	Modify your source code. Alternatively, you can check and modify your memory map.

Illegal operand of &

<i>Description</i>	This is an expression error—the expression attempts to take the address of an item that does not have an address.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Illegal pointer math

<i>Description</i>	This is an expression error—some types of pointer math are not valid in C expressions.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Illegal pointer subtraction

<i>Description</i>	This is an expression error—the expression attempts to use pointers in a way that is not valid.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Illegal structure reference

<i>Description</i>	This is an expression error—either the item being referenced as a structure is not a structure, or you are attempting to reference a nonexistent portion of a structure.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Illegal use of structures

<i>Description</i>	This is an expression error—the expression parameter is not using structures according to the C language rules.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Illegal use of void expression

<i>Description</i>	This is an expression error—the expression parameter does not meet the C language rules.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Integer not allowed

<i>Description</i>	This is an expression error—the command does not accept an integer as a parameter.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Invalid address**— Memory access outside valid range: *address***

<i>Description</i>	The debugger attempted to access memory at <i>address</i> , which is outside the memory map.
<i>Action</i>	Check your memory map to be sure that you access valid memory.

Invalid argument

<i>Description</i>	One of the command parameters does not meet the requirements for the command.
<i>Action</i>	Reenter the command with valid parameters. Refer to the appropriate command description in Chapter 12, <i>Summary of Commands</i> .

Invalid memory attribute

<i>Description</i>	The third parameter of the MA command specifies the type, or attribute, of the block of memory that is added to the memory map. The parameter entered did not match one of the valid attributes.	
<i>Action</i>	Reenter the MA command. Use one of the following valid parameters to identify the memory type:	
	R, ROM	(read-only memory)
	R W, RAM	(read/write memory)
	W, WOM	(write-only memory)
	PROM	(read-only program memory)
	PRAM	(read/write program memory)
	PROTECT	(no-access memory)
	OUTPORT, P W	(output port)
	INPORT, P R	(input port)
	IOPORT, P R W	(input/output port)

Invalid object file

<i>Description</i>	Either the file specified with File→Load Program, File→Reload Program, File→Load Symbols, the LOAD, the SLOAD, or the RELOAD command is not an object file that the debugger can load, or it has been corrupted.
<i>Action</i>	Be sure that you are loading an actual object file. Be sure that the file was linked. You may want to run cl6x (with the -z option) or lnk6x again to create an executable object file. If the file you attempted to load was a valid executable object file, then it was probably corrupted; recompile, assemble, and link with cl6x.

Invalid watch delete

<i>Description</i>	The debugger cannot delete the parameter supplied with the WD command.
<i>Action</i>	Reenter the WD command. Be sure to specify the symbol name that matches the item you want to delete.

Invalid window position

<i>Description</i>	The debugger cannot move the window to the XY position specified with the MOVE command. Either the XY parameters are not within the screen limits, or the active window may be too large to move to the desired position.
<i>Action</i>	Reenter the MOVE command. Enter the X and Y parameters in pixels.

Invalid window size

<i>Description</i>	The width and length specified with the SIZE or MOVE command may be too large or too small. If valid width and length were specified, then the active window is already at the far right or bottom of the screen and so cannot be made larger.
<i>Action</i>	Reenter the SIZE command. Enter the width and length in pixels.

L

Load aborted

<i>Description</i>	This message always follows another message.
<i>Action</i>	Refer to the message that preceded <i>Load aborted</i> .

Lost power (or cable disconnected)

<i>Description</i>	Either the target cable is disconnected, or the target system is faulty.
<i>Action</i>	Check the target cable connections. If the target seems to be connected correctly, see section C.5, <i>Additional Instructions for Hardware Errors</i> , page C-26.

Lost processor clock

<i>Description</i>	Either the target cable is disconnected, or the target system is faulty.
<i>Action</i>	Check the target cable connections. If the target seems to be connected correctly, see section C.5, <i>Additional Instructions for Hardware Errors</i> , page C-26.

Lval required

<i>Description</i>	This is an expression error—an assignment expression was entered that requires a legal left-hand side.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

M

Memory access error at *address*

<i>Description</i>	Either the processor is receiving a bus fault, or there are problems with target system memory.
<i>Action</i>	See section C.5, <i>Additional Instructions for Hardware Errors</i> , page C-26.

Memory map table full

<i>Description</i>	Too many blocks have been added to the memory map. This rarely happens unless blocks are added word by word (which is inadvisable).
<i>Action</i>	Stop adding blocks to the memory map. Consolidate any adjacent blocks that have the same memory attributes.

More than 4 reads to register *register* at cycle *cnum*

<i>Description</i>	Your program issued more than four reads of the same register in the same cycle, which is illegal. However, conditional registers are not included in this count. For more information about register read constraints, see the <i>TMS320C62x/C67x CPU and Instruction Set Reference Guide</i> .
<i>Action</i>	Modify your source code.

Move path conflicts at cycle *cycle*

<i>Description</i>	Your program issued two instructions that used the same functional unit in the same execute packet. For more information about functional units and resource constraints, see the <i>TMS320C62x/C67x CPU and Instruction Set Reference Guide</i> .
<i>Action</i>	Modify your source code.

Multiple writes to register *register* at cycle *cnum*

Description Your program issued multiple writes to the same register in the same cycle. This problem occurs due to the latency associated with previous instructions in the pipeline. For more information, see the *TMS320C62x/C67x CPU and Instruction Set Reference Guide*.

Action Modify your source code.

N

Name “*name*” not found

Description The command cannot find the object named *name*.

Action If *name* is a symbol, be sure that it was typed correctly. If it was not, reenter the command with the correct name. If it was, then be sure that the associated object file is loaded.

Nesting of repeats cannot exceed 100

Description The debugger cannot simulate more than 100 levels of repeat nesting in an input data file. If this happens, the debugger disconnects the input file from the pin.

Action Correct the input file so that the data does not include nesting repetition exceeding 100. Use the PINC command to reconnect the input file to the desired pin.

No file connected to this pin

Description You tried to disconnect the input file from a pin that was not previously connected to that pin.

Action Use the PINL command to list all of the pins and the files connected to them. Use the PIND command to reenter the correct pinname and filename.

P

Pinname not valid for this chip

Description You attempted to connect or disconnect an input file to an invalid interrupt pin.

Action Reconnect or disconnect the input file to an unused interrupt pin ().

Pointer not allowed

<i>Description</i>	This is an expression error.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Processor is already running

<i>Description</i>	One of the RUN commands was entered while the debugger was running free from the target system.
<i>Action</i>	Enter the HALT command to stop the free run, then reenter the desired RUN command.

R

Read conflicts with long operand at cycle *cnum*

<i>Description</i>	Your program attempted to write more than one long result in a single cycle on each side of the register file. Because the .L and .S units share their long read port with the store port, operations that read a long value cannot be issued on the .L and/or .S units in the same execute packet as a store. For more information about long path conflicts, see the <i>TMS320C62x/C67x CPU and Instruction Set Reference Guide</i> .
<i>Action</i>	Modify your source code.

Read not allowed for port

<i>Description</i>	You attempted to connect a file for input operation to an address that is not configured for read.
<i>Action</i>	Remap the port of correct the access in your source code.

Register access error

<i>Description</i>	Either the processor is receiving a bus fault, or there are problems with target-system memory.
<i>Action</i>	See section C.5, <i>Additional Instructions for Hardware Errors</i> , page C-26.

S

Specified map not found

<i>Description</i>	The MD command was entered with an address or block that is not in the memory map.
<i>Action</i>	Use the ML command to verify the current memory map. When using MD, you can specify only the first address of a defined block.

Structure member name required

<i>Description</i>	This is an expression error—a symbol name is followed by a period but no member name.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Structure member not found

<i>Description</i>	This is an expression error—an expression references a non-existent structure member.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

Structure not allowed

<i>Description</i>	This is an expression error—the expression is attempting an operation that cannot be performed on a structure.
<i>Action</i>	See section C.4, <i>Additional Instructions for Expression Errors</i> , page C-26.

T

Take file stack too deep

<i>Description</i>	Batch files can be nested up to ten levels deep. The batch file that you tried to execute with File→Take or the TAKE command calls batch files that are nested more than ten levels deep.
<i>Action</i>	Edit the batch file that caused the error. Instead of calling another batch file from within the offending file, you can to copy the contents of the second file into the first. This will removes a level of nesting.

Too many breakpoints

<i>Description</i>	200 breakpoints are already set, and there was an attempt to set another. The maximum limit of 200 breakpoints includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.
<i>Action</i>	Open the Breakpoint Control dialog box by selecting Breakpoints from the Setup menu. Delete individual software breakpoints.

Too many paths

<i>Description</i>	More than 20 paths have been specified cumulatively with the USE command, D_SRC environment variable, and -i debugger option.
<i>Action</i>	Do not enter the USE command before entering another command that has a <i>filename</i> parameter. Instead, enter the second command and specify full path information for the <i>filename</i> .

U

Undeclared port address

<i>Description</i>	You attempted to do a connect/disconnect on an address that is not declared as a port.
<i>Action</i>	Verify the address of the port to be connected or disconnected.

User halt

<i>Description</i>	The debugger halted program execution because you clicked the Halt icon on the toolbar, you selected Halt! from the Target menu, or you pressed the ESC key.
<i>Action</i>	None required; this is normal debugger behavior.

W

Window not found

Description The parameter supplied for the WIN command is not a valid window name.

Action Reenter the WIN command. Here are the valid window names; the bold letters show the smallest acceptable abbreviations:

Calls	CPU	Command
Disassembly	Memory	Profile
Watch		

Write conflicts with long writes at cycle *cnum*

Description Your program attempted to write more than one long result in a single cycle on each side of the register file. Because the .S and .L units share a read register port for long source operands and a write register port for long results, only one of these operations can be issued per side in an execute packet. For more information about long path conflicts, see the *TMS320C62x/C67x CPU and Instruction Set Reference Guide*.

Action Modify your source code.

Write not allowed for port

Description You attempted to connect a file for output operation to an address that is not configured for write.

Action Either change the software to write a port that is configured for write, or change the attributes of the port.

C.3 Alphabetical Summary of PDM Messages

This section contains an alphabetical listing of the error messages that the PDM might display. Each message contains both a description of the situation that causes the message and an action to take.

Note:

If errors are detected in a TAKE file, the PDM aborts the batch file execution, and the file line number of the invalid command is displayed along with the error message.

C

Cannot communicate with “name”

Description The PDM cannot communicate with the named debugger, because the debugger either crashed or was exited.

Action Spawn the debugger again.

Cannot communicate with the child debugger

Description This error occurs when you are spawning a debugger. The PDM was able to find the debugger executable file, but the debugger could not be invoked for some reason, and the communication between the debugger and PDM was never established. This usually occurs when you have a problem with your target system.

Action Exit the PDM and go back through the installation instructions in the installation guide. Reinvoke the PDM and try to spawn the debugger again.

Cannot create mailbox

Description The PDM was unable to create a mailbox for the new debugger that you were trying to spawn; the PDM must be able to create a mailbox in order to communicate with each debugger. This message usually indicates a resource limitation (you have more debuggers invoked than your system can handle).

Action If you have numerous debuggers invoked and you’re not using all of them, close some of them. If you are under a UNIX environment, use the ipcs command to check your message queues; use ipcrm to clean up the message queues.

Cannot open log file

- Description* The PDM cannot find the filename that you supplied when you entered the DLOG command.
- Action* Be sure that the file resides in the current directory or in one of the directories specified by the D_DIR environment variable.
- ☐ Check to see if you mistyped the filename.

Cannot open take file

- Description* The PDM cannot find the batch filename supplied for the TAKE command. You will also see this message if you try to execute a batch file that does not have a .pdm extension.
- Action* Be sure that the file resides in the current directory or in one of the directories specified by the D_DIR environment variable.
- ☐ Check to see whether you mistyped the filename.
 - ☐ Be sure that the batch filename has a .pdm extension.
 - ☐ Be sure that the file has executable rights.

Cannot open temporary file

- Description* The PDM is unable create a temporary file in the current directory.
- Action* Change the permissions of the current directory.

Cannot seek in file

- Description* While the PDM was reading a file, the file was deleted or modified.
- Action* Be sure that the files the PDM reads are not deleted or modified during the read.

Cannot spawn child debugger

- Description* The PDM couldn't spawn the debugger that you specified, because the PDM couldn't find the debugger executable file (emu6x). The PDM will first search for the file in the current directory and then search the directories listed with the PATH statement.
- Action* Check to see if the executable file is in the current directory or in a directory that is specified by the PATH statement. Modify the PATH statement if necessary, or change the current directory.

Command error

<i>Description</i>	The syntax for the command that you entered was invalid (for example, you used the wrong options or arguments).
<i>Action</i>	Reenter the command with valid parameters.

D

Debugger spawn limit reached

<i>Description</i>	The PDM spawned the maximum number of debuggers that it can keep track of in its internal tables. The maximum number of debuggers that the PDM can track is 2048. However, your system may not have enough resources to support that many debuggers.
<i>Action</i>	Before trying to spawn an additional debugger, close any debuggers that you don't need to run.

I

Illegal flow control

<i>Description</i>	One of the flow control commands (IF/ELIF/ELSE/ENDIF or LOOP/BREAK/CONTINUE/ENDLOOP) has an error. This error usually occurs when there is some type of imbalance in one of these commands.
<i>Action</i>	Check the flow command construct for such problems as an IF without an ENDIF, a LOOP without an ENDLOOP, or a BREAK that does not appear between a LOOP and an ENDLOOP. Edit the batch file that contains the problem flow command, or interactively reenter the correct command.

Input buffer overflow

<i>Description</i>	The PDM is trying to execute or manipulate an alias or shell variable that has been recursively defined.
<i>Action</i>	Use the SET and/or ALIAS commands to check the definitions of your aliases and system variables. Modify them as necessary.

Invalid command

<i>Description</i>	The command that you entered was not valid.
<i>Action</i>	Refer to the command summary in Chapter 12, <i>Summary of Commands and Special Keys</i> , for a complete list of commands and their syntax.

Invalid expression

<i>Description</i>	The expression that you used with a flow control command or the @ command is invalid. You may see specific messages before this one that provide more information about the problem with the expression. The most common problem is the failure to use the \$ character when evaluating the contents of a system variable.
<i>Action</i>	Check the expression that you used. Refer to section 11.7, <i>Understanding the PDM's Expression Analysis</i> , page 11-17, for more information about expression analysis.

Invalid shell variable name

<i>Description</i>	The system variable name that you used the SET command to assign is invalid. Variable names can contain any alphanumeric characters or underscore characters.
<i>Action</i>	Use a different name.

M**Maximum loop depth exceeded**

<i>Description</i>	The LOOP/ENDLOOP command that you tried to execute had more than 10 nested LOOP/ENDLOOP constructs. LOOP/ENDLOOP constructs can be nested up to 10 deep.
<i>Action</i>	Edit the batch file that contains the LOOP/ENDLOOP construct, or reenter the LOOP/ENDLOOP command interactively.

Maximum take file depth exceeded

<i>Description</i>	The batch file that you tried to execute with the TAKE command called or nested more than 10 other batch files. The TAKE command can handle batch files that are nested up to 10 deep.
<i>Action</i>	Edit the batch file.

U**Unknown processor name “name”**

Description The processor name that you specified with the `—g` option or a processor name within a group that you specified with the `—g` option does not match any of the names of the debuggers that were spawned under the PDM.

Action Be sure that you’ve correctly entered the processor name.

C.4 Additional Instructions for Expression Errors

Whenever you receive an expression error, you should reenter the command and edit the expression so that it follows the C language expression rules. If necessary, refer to a C language manual such as *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie.

C.5 Additional Instructions for Hardware Errors

If you continue to receive the messages that send you to this section, this indicates persistent hardware problems.

- ☐ If a bus fault occurs, the emulator may not be able to access memory.
- ☐ The 'C6x must be reset before you can use the emulator. Most target systems reset the 'C6x at power-up; your target system may not be doing this.

Glossary

A

active window: The window that is currently selected for moving, sizing, editing, closing, or some other function.

aggregate type: A C data type, such as a structure or array, in which a variable is composed of multiple variables, called members.

aliasing: A method of customizing debugger commands; aliasing provides a shorthand method for entering often-used command strings.

ANSI C: A version of the C programming language that conforms to the C standards defined by the *American National Standards Institute*.

assembly mode: A debugging mode that shows assembly language code in the Disassembly window and does not show the File window, no matter what type of code is currently running.

autoexec.bat: A batch file that contains DOS commands for initializing your PC.

auto mode: A context-sensitive debugging mode that automatically switches between showing assembly language code in the Disassembly window and C code in the File window, depending on what type of code is currently running.

B

batch file: One of two different types of files. One type contains DOS commands for the PC to execute. A second type of batch file contains debugger commands for the debugger to execute. The PC does not execute debugger batch files, and the debugger does not execute PC batch files.

benchmarking: A type of program execution that allows you to track the number of CPU cycles consumed by a specific section of code.

big endian: An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *little endian*

breakpoint: A point within your program where execution will halt because of a previous request from you.

C

Calls window: A window that lists the functions called by your program.

casting: A feature of C expressions that allows you to use one type of data as if it were a different type of data.

cl6x: A shell utility that invokes the 'C6x compiler, assembler, and linker to create an executable object file version of your program.

click: To press and release a mouse button without moving the mouse.

code-display windows: Windows that show code, text files, or code-specific information. This category includes the Disassembly, File, and Calls windows.

command line: The portion of the Command window where you can enter commands.

Command window: A window that provides an area for you to enter commands and for the debugger to echo command entry, show command output, and list progress or error messages.

common object file format (COFF): A binary object file format that promotes modular programming by supporting the concept of *sections*. All COFF sections are independently relocatable in memory space; you can place any section into any allocated block of target memory.

CPU window: A window that displays the contents of 'C6x on-chip registers, including the program counter, status register, A-file registers, and B-file registers.

cursor: An icon on the screen (such as an arrow or a horizontal line) that is used as a pointing device. The cursor is usually under mouse or keyboard control.

D

D_DIR: An environment variable that identifies the directory containing the commands and files necessary for running the debugger.

D_OPTIONS: An environment variable that you can use for identifying often-used debugger options.

D_SRC: An environment variable that identifies directories containing program source files.

data-display windows: Windows for observing and modifying various types of data. This category includes the Memory, CPU, and Watch windows.

debugger: A window-oriented software interface that helps you to debug 'C6x programs running on a 'C6x emulator or simulator.

disassembly: Assembly language code formed from the reverse-assembly of the contents of memory.

Disassembly window: A window that displays the disassembly (reverse assembly) of memory contents.

display area: The portion of the Command window or PDM window where the debugger/PDM echoes command entry, shows command output, and lists progress or error messages.

dock (a window): To anchor a floating window to an outer edge of the debugger application window. A docked window has no title bar and cannot be moved. However, a docked window can be resized.

drag: To move an object on the debugger display by pressing one of the mouse buttons and moving the mouse.

E

EISA: *Extended Industry Standard Architecture.* A standard for PC buses.

emulator: A debugging tool that is external to the target system and provides direct control over the 'C6x processor that is on the target system.

emurst: A utility that resets the emulator.

environment variable: A special system symbol that the debugger uses for finding directories or obtaining debugger options.

F

File window: A window that displays the contents of the current C code. The File window is intended primarily for displaying C code but can be used to display any text file.

float (a window): To cause a debugger window to sit on top of the debugger application window outside the edges of the debugger application window. A floating window always appears active.

I

init.cmd: A batch file that contains debugger-initialization commands. If this file is not present when you first invoke the debugger, then all memory is invalid.

I/O switches: Hardware switches on the emulator that identify the PC I/O memory space used for emulator-debugger or EVM-debugger communications.

ISA: *Industry Standard Architecture*. A subset of the EISA standard.

L

little endian: An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*

M

memory map: A map of memory space that tells the debugger which areas of memory can and cannot be accessed.

Memory window: A window that displays the contents of memory.

menu bar: A row of pulldown menu selections found at the top of the debugger display.

mixed mode: A debugging mode that simultaneously shows both assembly language code in the Disassembly window and C code in the File window.

O

open-collector output: An output circuit that actively drives both high and low logic levels.

P

PC: Personal computer or program counter, depending on the context and where it is used in this book: 1) In installation instructions or information relating to hardware and boards, *PC* means *personal computer*. 2) In general debugger and program-related information, *PC* means *program counter*, which is the register that identifies the current statement in your program.

PDM: *Parallel Debug Manager.* A program used for creating and controlling multiple debuggers for the purpose of debugging code in a parallel-processing environment.

point: To move the mouse cursor until it overlays the desired object on the screen.

port address: The PC I/O memory space that the debugger uses for communicating with the emulator. The port address is selected via switches on the emulator board and communicated to the debugger with the `-p` debugger option.

pulldown menu: A command menu that is accessed by name or with the mouse from the menu bar at the top of the debugger display.

R

ripple-carry output signal: An output signal from a counter indicating that the counter has reached its maximum value.

S

scalar type: A C type in which the variable is a single variable, not composed of other variables.

scroll bar: A bar on the right side or bottom of a window that allows you to adjust the contents of the window to display hidden information.

scroll bar handle: The rectangular box in the center of the right scroll bar in the Disassembly or Memory window that marks the center of disassembled code or memory contents.

scrolling: A method of moving the contents of a window up, down, left, or right to view contents that were not originally shown.

section: A relocatable block of code or data that will ultimately occupy contiguous space in the memory map.

side effects: A feature of C expressions in which using an assignment operator in an expression affects the value of one of the components used in the expression.

simulator: A development tool that simulates the operation of the 'C6x and lets you execute and debug applications programs by using the C source debugger.

single-step: A form of program execution that allows you to see the effects of each statement. The program is executed statement by statement; the debugger pauses after each statement to update the data-display windows.

status bar: An area at the bottom of the debugger application window that displays context-sensitive help and the status of the processor.

symbol table: A file that contains the names of all variables and functions in your program.

T

target system: A 'C6x board that works with the emulator; the emulator doesn't contain a 'C6x device, so it must use a 'C6x target board. Usually, the target system is a board that you have designed; you use the emulator and debugger to help you debug your design.

totem-pole output: An output circuit that actively drives both high and low logic levels.

W

Watch window: A window that displays the values of selected expressions, symbols, addresses, and registers.

window: A defined rectangular area of space on the display.

software breakpoints
defining command strings
memory contents
halting, temporarily

Index

! command 11-13 to 11-14, 12-12
? command
 description 7-3, 12-11
 display formats 7-24, 12-12
 examining register contents 7-14
 modifying PC 6-3
 side effects 7-5 to 7-6
& operator 7-8
\$EMU\$, 3-8
\$SIM\$, 3-8
% in alias parameter 3-3
@ command 11-19, 12-13
* (default) display format 7-22
* operator (indirection) 7-9, 7-19

A

ABE pseudoregister 10-13
absolute addresses 6-16, 7-8
ACE pseudoregister 10-14
active hardware events 10-6
active window
 definition D-1
 making a window active 12-60
ADDR command
 description 5-8, 12-13
 finding current PC 6-2
address data, profile window 8-20
addresses
 absolute addresses 6-16, 7-8
 accessible locations 4-2
 contents of (indirection) 7-9, 7-19
 hexadecimal notation 7-8
 I/O address space 4-14 to 4-15
 in Memory window 7-8
 invalid memory 4-2
 nonexistent memory locations 4-2
 protected areas 4-2, 4-8
 symbolic addresses 7-8
 undefined areas 4-2, 4-8
ADR pseudoregister 10-13
AEN pseudoregister 10-13
aggregate types
 definition D-1
 displaying 7-2, 7-18 to 7-21
ALIAS command 12-14
 PDM version 11-15 to 11-16
Alias Control dialog box 3-2
aliasing 11-15 to 11-16
 ALIAS command 12-14
 PDM version 11-15 to 11-16
 defining an alias 3-3
 definition D-1
 deleting an alias 3-4
 description 3-2 to 3-4
 editing an alias 3-4
 limitations 3-4
 redefining an alias 3-4
 supplying parameters 3-3
alternative data formats for display 7-22
analysis, memory system
 Analysis Statistics window 9-9
 commands 9-10
 counting events 9-6
 defining conditions 9-5
 event breakpoints 9-7
 list of events 9-2
 process 9-3
 removing event 9-7
 resetting event counters 9-9
 running programs 9-8
 viewing analysis data 9-9
Analysis Events dialog box
 emulator 10-5
 simulator 9-5

- Analysis menu
 - emulator 10-4
 - simulator 9-4
 - analysis module 10-1 to 10-14
 - Analysis Statistics window 10-11
 - counting events 10-6
 - dialog box* 10-6
 - EMU pins* 10-7 to 10-9
 - list of* 10-2
 - customized analysis commands 10-12
 - defining conditions 10-5 to 10-9
 - disabling 10-4
 - EMU pins 10-7 to 10-9
 - description* 10-2
 - restrictions* 10-7
 - enabling 10-4
 - functions 10-2
 - global breakpoints 10-9
 - hardware breakpoints 10-2
 - EMU pins* 10-2, 10-7 to 10-9
 - list of* 10-2, 10-8
 - internal counter 10-6
 - major functions 10-2
 - process 10-3
 - running programs 10-10
 - viewing analysis data 10-11
 - analysis pseudoregisters, summary 10-13 to 10-14
 - Analysis Statistics window
 - emulator 10-11
 - simulator 9-9
 - analysis status pseudoregister (AST) 10-14
 - ANSI C, definition D-1
 - arithmetic operators 13-2
 - arrays
 - displaying 7-18 to 7-21
 - member operators 13-2
 - as shell option 2-2, 8-3
 - ASM command 12-14
 - assembler 1-8
 - assembly language code
 - displaying in Disassembly window 2-15
 - displaying object code 5-2 to 5-5
 - displaying source code 5-5
 - assembly mode
 - ASM command 12-14
 - definition D-1
 - description 2-15
 - assembly mode (continued)
 - restrictions 2-15
 - typical display 2-15
 - assembly optimizer 1-8
 - assignment operators 7-5 to 7-6, 13-3
 - assistance from TI vii
 - AST pseudoregister 10-14
 - auto mode
 - C command 12-16
 - definition D-1
 - description 2-14
 - restrictions 2-14
 - typical assembly display 2-15
 - typical C display 2-16
 - autoexec.bat file, definition D-1
- ## B
- BA command 12-15
 - basic data-management commands 7-3
 - basic run commands 6-4 to 6-8
 - batch files
 - board.cfg B-1 to B-6
 - sample* B-2, B-4
 - board.dat 2-11, B-1 to B-6
 - controlling command execution 11-10 to 11-12
 - conditional commands* 3-8, 11-10 to 11-12, 12-26 to 12-61
 - looping commands* 3-9 to 3-10, 11-11 to 11-22, 12-28 to 12-29
 - definition D-1
 - displaying 5-6
 - displaying text when executing 3-7, 11-12, 12-21
 - echoing messages 3-7, 11-12, 12-21
 - emuinit.cmd 4-11, A-1
 - entering memory analysis commands 9-13
 - executing 3-11, 12-55
 - halting execution 3-11
 - init.clr 12-46, A-1
 - init.cmd
 - definition* D-4
 - during invocation* 4-11, A-2
 - init.pdm 2-8
 - initialization
 - emuinit.cmd* 4-11, A-1
 - init.cmd* 4-11, A-2
 - init.pdm* 2-8
 - sample memory map* 4-9
 - siminit.cmd* 4-11, A-1

- batch files (continued)
 - mem.map 4-12
 - memory maps 4-10, 4-13
 - pausing 3-11, 12-36
 - sample file 3-7
 - sim6x.cfg 2-11
 - siminit.cmd A-1
 - TAKE command 4-13, 12-55
 - PDM version* 11-9
 - BD command 12-15
 - benchmarking
 - CLK pseudoregister 6-12
 - constraints 6-12
 - definition D-1
 - description 6-12
 - RUNB command 12-45
 - big endian, defined D-1
 - big-endian format, selecting (*--me* option) 2-11
 - bitwise operators 13-3
 - BL command 12-15
 - board configuration
 - creating the file B-2 to B-6
 - naming an alternate file 2-11, B-6
 - specifying the file B-6
 - translating the file B-5
 - board.cfg file B-1 to B-6
 - device names B-3
 - device types
 - SPL* B-3
 - TMS320C6x* B-3
 - sample B-2, B-4
 - translating B-5
 - types of entries B-3 to B-5
 - board.dat, changing from the default file 2-11
 - board.dat file B-1 to B-6
 - default B-1
 - .bpt extension 6-18
 - BR command 12-16
 - BREAK command 11-11 to 11-22, 12-28 to 12-29
 - break events 10-2
 - Breakpoint Control dialog box 6-16, 8-15
 - breakpoint symbol 6-17, 8-16
 - breakpoints (event) 9-7
 - breakpoints (hardware) 10-8
 - definition D-2
 - global 10-7 to 10-9
 - types of events 10-2
 - breakpoints (software)
 - adding 6-15 to 6-16, 12-15
 - benchmarking with RUNB 6-12
 - clearing 6-17, 12-15, 12-16
 - command summary 12-7
 - definition D-2
 - description 6-14 to 6-19
 - listing set breakpoints 6-15, 12-15
 - loading breakpoint settings 6-19
 - maximum number 6-15
 - multiple or single statement 6-15
 - restrictions 6-15
 - saving breakpoint settings 6-18
 - setting 6-15 to 6-16
 - setting profile stopping points 8-15 to 8-16
 - with conditional run 6-11
 - Breakpoints toolbar icon
 - clearing a breakpoint 6-17
 - clearing all software breakpoints 6-17
 - loading breakpoint settings 6-19
 - saving breakpoint settings 6-18
 - setting a breakpoint 6-15
 - .bss section, clearing 2-10
- ## C
- c (ASCII character) display format 7-22
 - C (ASCII) display format 7-22
 - C command 12-16
 - C compiler 1-8
 - c* debugger option 2-10
 - C expressions 7-5 to 7-6, 13-1 to 13-6
 - C optimizer 1-8
 - C source
 - displaying 2-14, 2-16, 12-23
 - managing memory data 7-9
 - c6xtools directory
 - for HP-UX systems 2-3
 - for SPARC systems 2-3
 - for Windows 95 systems 2-3
 - CALLS command 12-16
 - effect on debugging modes 2-17
 - Calls window
 - definition D-2
 - description 1-4
 - displaying code for a function 5-7
 - casting
 - definition D-2
 - description 7-9, 13-4

- Change View context menu option 8-23
- char data type 7-22, 7-23
- CHDIR (CD) command 12-17, A-2
- cl6x shell, definition D-2
- clearing software breakpoints
 - debugger only 6-17
 - profiler only 8-16
- clearing the .bss section (-c) 2-10
- clearing the display area 12-17
- clicking, definition D-2
- CLK pseudoregister
 - description 6-12
 - restrictions in C code 6-12
 - validity of value in 6-12
- closing
 - debugger 2-18, 12-42
 - log files 3-13, 11-10, 12-21
 - PDM 12-42
 - Watch window 7-20, 12-61
- CLS command 12-17
- CNEXT command 6-10, 12-18
- code
 - debugging 1-13
 - debugging optimized code 2-2
 - preparing for debugging 2-2
 - profiling 8-1 to 8-28
 - profiling optimized code 2-2
- code development 1-7 to 1-9
- code-display windows
 - Calls window 1-4, 5-7
 - definition D-2
 - description 1-4
 - Disassembly window 1-4 to 1-14, 5-4 to 5-5
 - File window 1-4, 5-6 to 5-8
- COFF
 - in code development 1-8
 - loading 4-2
- comma operator 13-4
- command history, PDM version 11-13 to 11-14, 12-12
- command line
 - changing the prompt 12-40
 - definition D-2
- Command window
 - definition D-2
 - description 1-4
 - display area, clearing 12-17
- Command window (continued)
 - display during profiling 8-27
 - recording information from the display area 3-12 to 3-13, 12-20
- commands
 - alphabetical summary 12-11 to 12-61
 - available during profiling 8-4
 - available with analysis module 10-12
 - available with memory system analysis 9-10
 - breakpoint commands summary 12-7
 - code-execution (run) commands summary 12-8
 - command strings 3-2 to 3-4, 11-15 to 11-16
 - conditional commands 3-8, 11-10 to 11-12, 12-26 to 12-61
 - controlling command execution
 - conditional commands* 11-10 to 11-12, 12-26 to 12-61
 - looping commands* 11-11 to 11-22, 12-28 to 12-29
 - customizing 3-2 to 3-4, 10-12, 11-15 to 11-16
 - data-management commands summary 12-5
 - entering and using 3-1 to 3-13
 - entering from a batch file 3-11
 - entering operating system commands 3-5
 - file-display commands 12-4
 - functional summary (debugger) 12-2 to 12-10
 - help (online) access 1-14
 - load commands summary 12-4
 - looping commands 3-9 to 3-10, 11-11 to 11-22, 12-28 to 12-29
 - memory commands summary 12-7
 - mode commands 12-4
 - PDM commands 12-3
 - profiling commands 12-62 to 12-65
 - profiling commands summary 12-9
 - restrictions on validity 2-17
 - screen-customization commands summary 12-4
 - system commands 11-16
 - system commands summary 12-6
 - using a system shell 3-6
 - window commands 12-4
- common object file format, definition D-2
- compiler 1-8
- composer utility B-5
- condition for analysis 10-5, 10-9
- conditional commands 3-8, 11-10 to 11-12, 12-26 to 12-61
- conditional execution 6-11

- conditional run 6-11
- conditional single-stepping 6-8, 6-11
- configure analysis counter events pseudoregister (ACE) 10-14
- configure hardware breakpoints pseudoregister (ABE) 10-13
- constraints
 - benchmarking 6-12
 - CLK 6-12
- context-sensitive help, accessing 1-14
- CONTINUE command 11-11 to 11-22, 12-28 to 12-29
- continuous run
 - halting 6-13
 - starting 6-6
- continuous step
 - halting 6-13
 - starting 6-10
- continuous step execution, stepping until a break-point 6-10
- Count CPU Events dialog box 10-6
- count data 8-21
- counter, internal, disabling 10-6
- counters, resetting internal event (simulator) 9-9
- CPU window
 - definition D-2
 - description 1-4, 7-13 to 7-17
 - editing registers 7-5
 - reordering registers 7-14
- cr linker option 2-10
- CSTEP command 6-9, 12-18
- current directory, changing 12-17, A-2
- current field, editing 7-5
- current PC
 - finding 6-2
 - selecting 6-2
- cursors, definition D-2
- customizing, memory types 4-4
- customizing the display
 - changing the prompt 12-40
 - loading a custom display 12-46
 - saving a custom display 12-52

D

- d (decimal) display format 7-22
- d debugger option 2-10
- D_DIR environment variable, effects on debugger invocation A-1
- D_OPTIONS environment variable 2-10 to 2-13
 - definition D-2
 - effects on debugger invocation A-1, A-2
 - for HPUNIX systems 2-5 to 2-7
 - for SPARC systems 2-5 to 2-7
 - for Windows 95 systems 2-5
 - ignoring (-x option) 2-13
 - setting up 2-5
- D_SRC environment variable 2-9
 - definition D-2
 - effects on debugger invocation A-1
 - for HPUNIX systems 2-4 to 2-5
 - for SPARC systems 2-4 to 2-5
 - for Windows 95 systems 2-4
 - naming additional directories A-2
 - setting up 2-4
- D_DIR environment variable 2-8
 - definition D-2
 - for HPUNIX systems 2-3 to 2-4
 - for SPARC systems 2-3
 - for Windows 95 systems 2-3
 - setting up 2-3
- DASM command
 - description 12-19
 - effect on debugging modes 2-17
 - finding current PC 6-2
- data-display windows, definition D-3
- data formats 7-22
- data management, determining variable types 7-3
- data-management commands
 - controlling data format 7-9
 - EVAL command, PDM version 12-22
 - side effects 7-5 to 7-6
 - summary 12-5
- data memory, adding to memory map 4-10 to 4-18
- data type
 - changing the default 7-22
 - for displaying debugger data 7-23
 - parameter for SETF command 7-23
- data-management commands, EVAL command, PDM version 11-21

- data-display windows
 - CPU window 1-4, 7-2, 7-13 to 7-17
 - description 1-4
 - Memory window 1-4, 7-2, 7-7 to 7-12
 - overview 7-2
 - Watch window 1-4, 7-2, 7-18 to 7-21
- data-management
 - changing data value 7-5 to 7-8
 - changing memory range displayed in Memory window 7-7 to 7-25
 - changing the default display format 7-24
 - commands 7-22 to 7-25
 - editing data in a window 7-5
 - editing data with expressions that have side effects 7-5 to 7-25
 - evaluating an expression 7-3
 - in a Watch window 7-18 to 7-20
 - in memory 7-7 to 7-12
 - in registers 7-5, 7-13
- debugger
 - commands
 - alphabetical summary* 12-11 to 12-61
 - functional summary* 12-2 to 12-10
 - profiling* 12-62 to 12-65
 - definition D-3
 - description 1-3 to 1-6
 - display, illustration 1-3
 - exiting 2-18, 12-42
 - installation, describing the target system B-1 to B-6
 - invocation 12-51
 - description* 2-7 to 2-9
 - emu?? command* 2-7
 - emu6x command* 2-7
 - options* 2-10 to 2-13
 - sim6x command* 2-7
 - standalone* 2-7
 - task ordering* A-1 to A-3
 - under PDM control* 2-8 to 2-9
 - key features 1-2
 - messages C-1 to C-26
 - setting up default options (D_OPTIONS) 2-5
- debugging modes
 - assembly mode 2-15
 - auto mode 2-14
 - command summary 12-4
 - default mode 2-14
 - description 2-14 to 2-17
 - mixed mode 2-16
- debugging modes (continued)
 - restrictions 2-17
 - restrictions on validity 2-17
- decrement operator 13-3
- default
 - data formats 7-22 to 7-25
 - debugging mode 2-14
 - display 2-14
 - group 11-4, 12-47
 - memory map 4-9
 - Memory window 7-7
 - stopping point for profiling 8-15
- defining an alias 3-3
- defining areas for profiling
 - description 8-5 to 8-12
 - disabling areas 8-10 to 8-12, 12-62 to 12-63
 - enabling areas 8-11 to 8-12, 12-63
 - marking areas 8-5 to 8-9, 12-62
 - restrictions 8-12
 - unmarking areas 8-12, 12-64
- defining command strings. *See* aliasing; commands, command strings
- deleting watched values 7-20
- determining type of a variable 7-3
- developing code 1-7 to 1-9
- device name B-3
- device types
 - debugger devices B-3
 - SPL B-3
 - TMS320C6x B-3
- dgroup 11-4
- dialog boxes
 - accessing online help 1-14
 - enabling parameters 10-8
- DIR command 12-19
- directories
 - i debugger option 2-11
 - auxiliary files
 - for HP/UX systems* 2-3
 - for SPARC systems* 2-3
 - for Windows 95 systems* 2-3
 - c6xtools
 - for HP/UX systems* 2-3
 - for SPARC systems* 2-3
 - for Windows 95 systems* 2-3
 - changing current directory 12-17

- directories (continued)
 - identifying additional source directories 12-57
 - for HPUX systems* 2-4
 - for SPARC systems* 2-4
 - for Windows 95 systems* 2-4
 - identifying alternate directories (D_DIR) 2-3
 - identifying current directory A-2
 - identifying directories with program source files (D_SRC) 2-4
 - listing contents of current directory 12-19
 - relative pathnames 12-17
 - search algorithm 3-11, A-1 to A-3
 - USE command 12-57
 - disabling analysis 9-4, 10-4
 - disabling areas for profiling 8-10 to 8-12
 - disabling memory mapping 4-7 to 4-8
 - disassembly
 - definition D-3
 - description 5-4
 - displaying 5-4 to 5-5
 - Disassembly window
 - Address field 5-4
 - definition D-3
 - description 1-4
 - modifying the display 12-19
 - running code to a specific point 6-5
 - scrolling through the contents 5-5
 - setting a breakpoint 6-15
 - setting current PC 6-2
 - viewing disassembly 5-4 to 5-5
 - DISP command
 - description 12-19
 - display formats 7-24, 12-20
 - effect on debugging modes 2-17
 - display, basic debugger 1-3
 - display area
 - clearing 12-17
 - definition D-3
 - recording information from 3-12 to 3-13, 11-10, 12-20
 - display-customization commands 12-4
 - display formats
 - ? command 7-24, 12-12
 - data types 7-23
 - description 7-22 to 7-25
 - DISP command 7-24, 12-20
 - EVAL command 11-21, 12-23
 - MEM command 7-24, 12-33
 - resetting types 7-24
 - display formats (continued)
 - SETF command 7-22 to 7-26, 12-49
 - table 7-22
 - WA command 7-24, 12-59
 - Display Rate frequency bar, Profile window 8-17
 - displaying
 - assembly language code 5-2
 - disassembly* 5-4
 - source* 5-5
 - batch files 5-6
 - C code 5-6 to 5-8
 - C function 5-7
 - code at a specific point 5-8
 - data for load and store instructions (emulator) 7-2
 - data in nondefault formats 7-22 to 7-25
 - debugger on a different machine (-d option) 2-10
 - pointer data 7-20
 - register contents 7-13
 - structure data 7-20
 - text files 5-6
 - text when executing a batch file 3-7, 12-21
 - watched values 7-19 to 7-20
 - DLOG command 12-20
 - ending recording session 11-10
 - PDM version 11-10
 - starting recording session 11-10
 - docking a window, definition D-3
 - double data type 7-23
 - double-precision floating-point registers 7-17
 - dragging, definition D-3
- E**
- e (exponential floating-point) display format 7-22
 - E command 12-22
 - ECHO command 3-7, 12-21
 - PDM version 11-12
 - editing
 - data values 7-5
 - expression side effects 7-5
 - overwrite method 7-5
 - EISA, definition D-3
 - ELIF command 11-10 to 11-12, 12-21, 12-26 to 12-61
 - ELSE command 3-8, 12-27
 - PDM version 11-10 to 11-12, 12-26 to 12-61
 - \$\$EMU\$\$ constant 3-8

- EMU pins 10-9
 - description 10-2
 - external counter 10-9
 - restrictions 10-7
 - setup for external counter 10-7 to 10-9
 - setup for global breakpoints 10-9
- emu6x command 2-8, 12-51
 - options 2-7, 2-10 to 2-13
 - n* 2-11
- emuinit.cmd file 2-3, A-1
- emulator
 - definition D-3
 - describing the target system to the debugger B-1 to B-6
 - creating the board configuration file B-2 to B-6*
 - specifying the file B-6*
 - translating the file B-5*
 - displaying data for load and store instructions 7-2
 - \$\$EMU\$\$ constant 3-8
 - external counter 10-7
 - halting on the first instruction of an interrupt service routine 6-4
 - invoking the debugger 2-7, 12-51
 - standalone 2-7*
 - under PDM control 2-8 to 2-9*
 - reconnecting to debugger 12-42
 - resetting 2-6, 6-7
 - running code while disconnected from target system 6-6
 - specifics in halting 6-13
 - using conditional RUN command 6-4
- emurst command 2-6
- emurst file, definition D-3
- enable analysis pseudoregister (AEN) 10-13
- enabling analysis 9-4, 10-4
- enabling areas for profiling 8-11 to 8-12
- enabling memory mapping 4-7 to 4-8
- ENDIF command 3-8, 12-27
 - PDM version 11-10 to 11-12, 12-26 to 12-61
- ENDLOOP command 3-9 to 3-10, 12-29
 - PDM version 11-11 to 11-22, 12-28 to 12-29
- entering commands, from the PDM 2-8, 11-2
- entering operating system commands 3-5
- entering profiling environment
 - menu option 8-4
 - profile* option 2-12
- entry point (of program) 6-2
- environment variables
 - D_OPTIONS 2-5, 2-10 to 2-13
 - D_DIR 2-3, 2-8
 - D_SRC 2-4, 2-9
 - definition D-3
 - effects on debugger invocation A-1
 - for debugger options 2-10 to 2-13
- error messages C-22 to C-26
 - beeping 12-51, C-2
 - description C-1 to C-26
- escape key, halting execution 6-13
- EVAL command
 - description 7-4, 12-22
 - display formats 11-21, 12-23
 - modifying PC 6-3
 - PDM version 11-21, 12-22
 - side effects 7-5 to 7-6
- evaluating an expression 7-3
- event
 - counting (emulator) 10-6
 - counting (simulator) 9-6
 - definition (emulator) 10-5
 - definition (simulator) 9-5
 - event breakpoint (simulator) 9-7
 - hardware breakpoint 10-8
 - removing (simulator) 9-7
- event_break command 9-11
- event_counter_reset command 9-12
- event_counter_start command 9-11
- event_disable command 9-11
- event_enable command 9-10
- event_list command 9-12
- event_number parameter, determining 9-10
- event_reset command 9-12
- exclusive data 8-17, 8-20, 8-21
- exclusive maximum data 8-17, 8-20, 8-21
- executing code
 - checking execution status 11-20, 12-48
 - finding execution status 11-8, 12-53
 - while disconnected from the target system 12-41
- executing code while disconnected from the target system 6-6, 12-45
- executing commands 3-1 to 3-13
- execution, pausing 11-13
- exiting the debugger 2-18, 12-42

expressions
 addresses 7-8
 analysis 13-4 to 13-6
 description 13-1 to 13-6
 evaluating 11-21
 by the PDM 11-17
 evaluation 12-22
 with ? command 7-3, 12-11
 with DISP command 12-19
 with EVAL command 7-4, 12-22
 with LOOP command 3-9, 12-29
 operators 11-17, 13-2 to 13-3
 restrictions 13-4
 void expressions 13-4
 with side effects 7-5 to 7-6

external counter value pseudoregister (XCNT) 10-14

external event counter 10-7 to 10-9

external interrupts 4-16
 connect input file 4-17, 12-37
 disconnect pins 4-18, 12-38
 list pins 4-18, 12-38
 PINC command 4-17, 12-37
 PIND command 4-18, 12-38
 PINL command 4-18, 12-38
 programming simulator 4-17, 4-18
 setting up input file
 relative clock cycle 4-16
 repetition 4-17
 setting up input files 4-16
 absolute clock cycle 4-16

F

f (decimal floating-point) display format 7-22
 -f debugger option 2-11, B-6

F1 key, accessing online help 1-14

F5 key
 running a profiling session 8-17
 running code 6-4, 9-8, 10-10

F8 key, single-stepping 6-9

F9 key, changing the File window display 5-7

F10 key, single-stepping over function calls 6-10

faster simulator
 debugger features not supported 1-11
 description 1-10

features, not supported by limited simulators 1-11

FILE command
 description 12-23
 effect on debugging modes 2-17

File menu
 Load Program option 5-2, 7-11
 Load Symbols option 5-3
 Log File option 3-12
 Open option 5-6
 Reload Program option 5-3
 Take option 3-11

File window
 definition D-3
 description 1-4, 5-6 to 5-8
 displaying any text file 5-6
 displaying assembly language source 5-5
 running code to a specific point 6-5
 setting a breakpoint 6-15
 setting current PC 6-2

file/load commands 12-4

files
 batch files 3-7
 connecting to I/O ports 4-14 to 4-15, 12-31
 creating executable object files 2-2
 debugger executable 2-7
 disconnecting from I/O ports 4-15, 12-33
 executable (emulator) 2-6
 loading object files 5-2
 log files 3-12 to 3-13, 11-10
 saving memory to a file 7-10 to 7-11, 12-35

FILL command 12-24

Fill Memory dialog box 7-11, 7-12

FILLB command 12-24

float data type 7-23

floating a window, definition D-3

floating-point double-precision registers 7-17

floating-point operations 13-4

floating-point single-precision registers 7-16

floating-point simulator
 debugger features not supported 1-11
 description 1-10

flow diagram
 analysis process (emulator) 10-3
 analysis process (simulator) 9-3
 code development 1-7
 debugging process 1-13
 profiling process 8-3

full profile 8-17, 12-36

FUNC command
 description 5-7, 12-24
 effect on debugging modes 2-17
function calls
 displaying functions 12-24
 executing function only 12-43
 in expressions 7-5, 13-4
 stepping over 12-18, 12-35
 tracking in Calls window 5-7

G

–g assembler option, displaying assembly language source 5-5
–g shell option 2-2, 6-9, 8-3
global breakpoints 10-9
GO command 6-5, 12-25
green arrow 8-6, 8-10, 8-11
grouping/reference operators 13-2
groups
 adding a processor 11-4, 12-48
 commands
 SET command 11-4 to 11-5, 12-47 to 12-48
 UNSET command 11-5, 12-56
 defining 11-4 to 11-5, 12-47
 deleting 11-5, 12-56
 examples 11-3
 identifying 11-2 to 11-5
 listing all groups 11-5, 12-48
 setting default 11-4, 12-47

H

HALT command 6-13, 12-25
Halt toolbar icon 6-13
halting
 batch file execution 3-11
 debugger 2-18, 12-42
 emulator-specific information 6-13
 multiple processors 10-9
 PDM 12-42
 processors in parallel 11-8, 12-37
 program execution 2-18, 6-13, 12-42
 target system 12-25
 temporarily. *See* breakpoints (software)
hardware breakpoints, dialog box 10-8
help, accessing 1-14 to 1-15

HELP command 1-14 to 1-15, 12-25, 12-26
Help menu, Help Topics option 1-14
Help toolbar icon 1-14
Help Topics toolbar icon 1-14
hexadecimal notation 6-16
 addresses 7-8
 data formats 7-22
history, of commands 12-12
HISTORY command 11-14, 12-26
history of commands 11-13 to 11-14

I

–i debugger option 2-11, A-3
I/O memory
 adding to memory map 4-10 to 4-18
 connecting I/O port 4-14 to 4-15
 deleting from memory map 12-32
 disconnecting I/O port 4-15
 simulating 4-14 to 4-15, 12-31, 12-33
memory
 batch file search order, memory initialization 4-11
 invalid addresses 4-2
 invalid locations 4-8
 map, adding ranges 4-10 to 4-18
 mapping, MA command 4-10 to 4-18
 nonexistent locations 4-2
 protected areas 4-2, 4-8
 simulating I/O memory 4-14 to 4-15
 simulating ports
 MC command 4-14 to 4-15
 MI command 4-15
 undefined areas 4-2, 4-8
 valid types 4-4, 4-5, 4-10
I/O switch settings, definition D-4
ICNT pseudoregister 10-14
icons, toolbar (basic display) 1-3
identifying a new board configuration file (–f) 2-11
identifying a new simulator configuration file 2-11
identifying additional source directories
 –i option 2-11
 D_DIR environment variable 2-3
identifying directories containing program source files (D_SRC) 2-4
identifying new initialization file (–t option) 2-13
IF/ELIF/ELSE/ENDIF commands 11-10 to 11-12, 12-26 to 12-61

IF/ELSE/ENDIF commands
 conditions 3-9, 3-10, 12-27
 creating initialization batch file 3-8
 description 3-8, 12-27
 predefined constants 3-8
 ignoring D_OPTIONS (-x option) 2-13
 inclusive data 8-20, 8-21
 inclusive maximum data 8-20, 8-21
 increment operator 13-3
 indirection operator (*) 7-9, 7-19
 init.clr file 12-46, A-1
 init.cmd file
 definition D-4
 during invocation 2-13, 4-11, A-2
 init.pdm file 2-8
 initialization batch files
 creating using IF/ELSE/ENDIF 3-8
 emuinit.cmd A-1
 example 4-9
 init.cmd 4-11, A-2
 init.pdm 2-8
 naming an alternate file (-t option) 2-13
 siminit.cmd A-1
 INPORT keyword 4-10
 int data type 7-23
 internal counter, disabling 10-6
 internal counter value pseudoregister (ICNT) 10-14
 interpreting profile data 8-25
 interrupt pins 4-16
 interrupt simulation
 ending 4-18
 initiating 4-17
 interrupts ignored while single-stepping 6-8
 invalid memory addresses 4-2, 4-8
 invoking
 debugger 2-7 to 2-9, 12-51
 standalone 2-7
 under PDM control 2-8 to 2-9
 parallel debug manager 2-8
 IOPORT keyword 4-10
 ISA, definition D-4

K

key sequences, halting actions 11-6, 11-7, 12-40, 12-47

L

limits
 breakpoints 6-15
 customized prompt length 12-40
 paths A-3
 LINE command 12-28
 linker 1-8
 little endian, defined D-4
 little-endian format 2-11
 Load Breakpoint File dialog box 6-19
 LOAD command 7-21, 12-28
 load instructions, displayed by the emulator 7-2
 Load List menu option (breakpoints) 6-19
 Load Program dialog box 5-2
 load/file commands 12-4
 loading
 assembly language code 5-2 to 5-5
 batch files 3-11
 COFF files, restrictions 4-2
 object code
 after invoking the debugger 5-2
 description 5-2 to 5-5
 symbol table only 2-13, 5-3, 12-50
 while invoking the debugger 2-7, 2-9, 5-3
 with global symbols only 2-13
 with symbol table 5-2
 without symbol table (RELOAD) 5-3, 12-42
 saved breakpoint settings 6-19
 Log File dialog box, opening a file 3-12
 log files 3-12 to 3-13, 11-10
 logical operators
 conditional execution 6-11
 description 13-2
 long data type 7-23
 LOOP/BREAK/CONTINUE/ENDLOOP commands 11-11 to 11-22, 12-28 to 12-29
 LOOP/ENDLOOP commands
 conditions 3-9, 3-10, 12-30
 description 3-9 to 3-10, 12-29
 looping commands 3-9 to 3-10, 11-11 to 11-22, 12-28 to 12-29

M

MA command 4-9, 4-10 to 4-18, 12-30
emulator syntax 4-10

managing data

- basic commands 7-3 to 7-4
- changing data values 7-5 to 7-6
- in memory 7-7 to 7-12
- in registers 7-13 to 7-17
- in Watch windows 7-18 to 7-21

MAP command 12-31

marking areas for profiling 8-5 to 8-9

MC command 4-14 to 4-15, 12-31

MD command 12-32

—me debugger option 2-11

MEM command

- description 7-8, 12-32
- display formats 12-33
- effect on debugging modes 2-17
- using to change display format of data 7-24

memory

- batch file search order A-1
- command summary 12-7
- data formats 7-22 to 7-25
- displaying in different numeric format 7-9
- filling
 - byte by byte* 7-12, 12-24
 - word by word* 7-11 to 7-12, 12-24
- saving 12-35
- saving values to a file 7-10 to 7-11
- simulating I/O memory 12-31, 12-33
- simulating ports
 - MC command* 12-31
 - MI command* 12-33

memory contents. *See* data management; memory window

Memory Map Control dialog box 4-3

memory mapping

- adding ranges 4-3 to 4-5, 4-10, 12-30
- checking memory accesses against 4-2
- command summary 12-7
- creating a map 4-3 to 4-6
- default map 4-9
- defining a map 4-2
- defining and executing a map in a batch file 4-10
- definition (memory map) D-4
- deleting ranges 4-6, 12-32

memory mapping (continued)

- description 4-1 to 4-18
- disabling 4-7 to 4-8
- enabling 4-7 to 4-8
- listing current map 4-3
- modifying a map 4-2, 4-3, 4-6
- multiple maps 4-13
- potential problems 4-2
- resetting 12-34
- restrictions 4-4
- returning to default 4-12
- sample map 4-9

Memory menu

- Fill Byte option 7-12
- Fill Word option 7-11
- Mapping option 4-3, 4-4
- Save option 7-10

memory system, Analysis Statistics window 9-9

memory system analysis

- counting events 9-6
- defining conditions 9-5
- event breakpoints 9-7
- not supported by fast simulator 9-1
- process 9-3
- removing event 9-7
- resetting event counters 9-9
- running programs 9-8
- viewing analysis data 9-9

memory system analysis commands

- entering through a batch file 9-13
- event_break 9-11
- event_counter_reset 9-12
- event_counter_start 9-11
- event_disable 9-11
- event_enable 9-10
- event_list 9-12
- event_reset 9-12

memory types

- customizing 4-4
- list of basic types 4-5

Memory window

- Address field 7-7
- changing range of memory displayed 7-7
- definition D-4
- description 1-4, 7-7 to 7-12
- displaying memory contents 7-7 to 7-26
- editing memory contents 7-5
- modifying display 12-32
- naming 7-8

Memory window (continued)
 opening additional windows 7-8
 scrolling through the contents 7-7

menu
 context menus 1-6
 definition (pulldown menu) D-5

menu bar
 basic display 1-3
 definition D-4

messages C-1 to C-26

–mg shell option 2-2

MI command 4-15, 12-33

MIX command 12-33

mixed mode
 definition D-4
 description 2-16
 MIX command 12-33
 restrictions 2-17
 typical display 2-17

ML command 12-34

mode commands 12-4

modifying
 current directory 12-17
 data values 7-5
 memory map 4-2, 4-3

mouse icon 8-6

MOVE command 12-34

moving a window 12-34

MR command 12-34

MS command 12-35

multiple debuggers, invoking 2-8

N

–n debugger option 2-9, 2-11, 11-2, 12-51

natural format 13-5

Next C Statement toolbar icon 6-10

NEXT command 6-10, 12-35

Next toolbar icon 6-10

nonexistent memory locations 4-2

notational conventions iv to v

O

o (octal) display format 7-22

–o shell option 2-2

.obj extension 7-10

object code

–v option 2-13
 loading global symbols only (–v option) 2-13
 loading symbol table only (–s option) 2-13
 –s option 2-13, 5-3

object files

creating 5-2
 loading 2-9
after invoking the debugger 5-2
LOAD command 12-28
symbol table only 2-13, 12-50
while invoking the debugger 2-7, 2-9, 5-3
with global symbols only 2-13
with symbol table 5-2
without symbol table (RELOAD) 5-3, 12-42

online help, accessing 1-14 to 1-15

Open File dialog box 5-6

Open Take File dialog box 3-11

Open toolbar icon 5-6

open-collector output, definition D-4

operating system

entering commands from the debug-
 ger 3-5 to 3-13, 12-54
 entering commands from the PDM 12-54
 exiting from system shell 12-54

operators 11-17

& operator 7-8
 * operator (indirection) 7-9, 7-19
 Boolean precedence 3-10
 causing side effects 7-6
 comma operator 13-4
 description 13-2 to 13-3
 in expressions 3-10, 6-11

optimized code

debugging 2-2
 profiling 2-2

optimizer

assembly 1-7
 C 1-7

options

debugger 2-10 to 2-13
 emurst 2-6

OUTPORT keyword 4-10

overwrite editing 7-5

P

p (valid address) display format 7-22

-p debugger option 2-12

P|R keyword 4-10

P|R|W keyword 4-10

P|W keyword 4-10

parallel debug manager

adding a processor to a group 12-48

assigning processor names, -n option 2-9, 12-51

changing the PDM prompt 12-48

checking the execution status 12-48

closing 12-42

command history 12-12

commands 12-3

! command 12-12

@ command 12-13

ALIAS command 12-14

DLOG command 12-20

ECHO command 12-21

EVAL command 12-22

HELP command 12-26

HISTORY command 12-26

IF/ELIF/ELSE/ENDIF commands 12-26 to 12-61

LOOP/BREAK/CONTINUE/ENDLOOP commands 12-28 to 12-29

PAUSE command 3-11, 12-36

PDM command 2-8

PESC command 12-36

PHALT command 12-37

PRUN command 12-40

PSTEP command 12-41

QUIT command 12-42

RUNF command 12-41

SEND command 12-46 to 12-47

SET command 12-47 to 12-48

SPAWN command 2-8 to 2-9, 12-51

STAT command 12-48, 12-53

TAKE command 12-55

UNSET command 12-56

viewing descriptions 12-26

creating system variables 12-48

defining a group 12-47

definition D-5

parallel debug manager (continued)

deleting a group 12-56

UNSET command 12-56

description 1-12

displaying text strings 12-21

finding the execution status 12-53

global halt 12-37

grouping processors, SET command 12-47 to 12-48

halting code execution 12-36

invoking 2-8

listing all groups of processors 12-48

messages C-22 to C-26

overview 2-8

pausing 3-11, 12-36

recording information from the display area 12-20

running code 12-40

sending commands to debuggers 12-46 to 12-47

setting the default group 12-47

single-stepping through code 12-41

supported operating systems 2-8

using with UNIX 2-8

parallel debug manager (PDM) 11-1 to 11-21

adding a processor to a group 11-4

assigning processor names 11-2
-n option 11-2

changing the PDM prompt 11-19

checking the execution status 11-20

command history 11-13 to 11-14

commands

! command 11-13 to 11-14

@ command 11-19

ALIAS command 11-15 to 11-16

creating system variables 11-18 to 11-19

deleting system variables 11-20

DLOG command 11-10

ECHO command 11-12

EVAL command 11-21

HISTORY command 11-14

IF/ELIF/ELSE/ENDIF commands 11-10 to 11-12

LOOP/BREAK/CONTINUE/ENDLOOP commands 11-11 to 11-22

PAUSE command 11-13

PESC command 11-8

PHALT command 11-8

PRUN command 11-7

PRUNF command 11-7

parallel debug manager (PDM) (continued)

commands

PSTEP command 11-7*SEND* command 11-6*SET* command 11-4 to 11-5*STAT* command 11-8, 11-20*SYSTEM* command 11-16*TAKE* command 11-9*UNALIAS* command 11-15 to 11-16*UNSET* command 11-5

controlling command execution 11-10 to 11-12

creating system variables 11-18 to 11-19

concatenating strings 11-18*substituting strings* 11-19

defining a group 11-4

deleting a group 11-5

UNSET command 11-5

deleting system variables 11-20

displaying text strings 11-12

expression analysis 11-17

finding the execution status 11-8

global halt 11-8

grouping processors 11-2 to 11-5

example 11-3*SET* command 11-4 to 11-5

halting code execution 11-8

listing all groups of processors 11-5

listing system variables 11-20

pausing 11-13

recording information from the display

area 11-10

running code 11-7

running free 11-7

sending commands to debuggers 11-6

setting the default group 11-4

single-stepping through code 11-7

system variables 11-18 to 11-20

parameters

emu?? command 2-7*emu6x* command 2-7

in alias definition (%) 3-3

notation *iv**sim6x* command 2-7*SPAWN* command 2-8 to 2-9, 12-51

path environment variable 2-8

PATH statement 2-8, 12-51

PAUSE command 3-11, 11-13, 12-36

PC (program counter)

definition D-4

finding the current PC 6-2

modifying 6-2

PDM, invocation 2-7

PDM command 2-8

PESC command 11-8, 12-36

PF command 12-36

PHALT command 11-8, 12-37

PINC command 4-17, 12-37

PIND command 4-18, 12-38

PINL command 4-18, 12-38

pointers

natural format 13-5

typecasting 13-5

pointing, definition D-5

port address 2-12, 4-14 to 4-15

definition D-5

ports, simulating 4-14 to 4-15, 12-31, 12-33

PQ command 12-38

PR command 12-39

PRAM keyword 4-10

predefined constants for conditional commands 3-8

.prf extension 8-27

processor name 2-11

processors

assigning names 11-2

organizing into groups 11-3 to 11-5

PROFILE command 12-39

profile cycles data 8-25

–profile debugger option 2-12

Profile Marking dialog box

disabling areas

description 8-10 to 8-11*valid areas* 8-13 to 8-14

enabling areas

description 8-11*valid areas* 8-13 to 8-14

marking areas

description 8-8 to 8-9*valid areas* 8-9

unmarking areas

description 8-12*valid areas* 8-13 to 8-14

Profile menu

- Change View option 8-24
- Profile Mode option 8-4
- Run option 8-17
- Save All option 8-28
- Save View option 8-27
- Select Areas option 8-8

Profile Run dialog box

- resuming a session 8-19
- running a session 8-17

Profile View dialog box, areas for viewing 8-13

Profile View dialog box

- changing profile display 8-22, 8-24
- sorting profile data 8-23

Profile window

- changing profile display 8-22, 8-24
- description 1-4, 8-20 to 8-26
- displaying areas 8-24 to 8-25
- displaying different data 8-21 to 8-22
- marking areas 8-8
- resetting 8-25, 12-58
- sorting data 8-23
- viewing associated code 8-25 to 8-26

profiling

areas

- description* 8-5 to 8-12
- disabling* 8-10 to 8-12, 12-62 to 12-63
- enabling* 8-11 to 8-12, 12-63
- marking* 8-5 to 8-9, 12-62
- restrictions* 8-12
- unmarking* 8-12, 12-64
- valid* 8-9

breakpoints (software)

- clearing* 8-16
- resetting* 8-16
- setting* 8-15

changing display 8-24 to 8-25, 12-65

collecting statistics

- full statistics* 8-17 to 8-18, 12-36
- subset of statistics* 8-17 to 8-18, 12-38

commands

- debugger commands available during profiling* 8-4
- MA command* 4-10
- summary for batch files* 12-62 to 12-65
- summary for debugger command line* 12-9

compiling a program for profiling 8-3

description 8-1 to 8-28

entering environment 2-12, 8-4

highlighting marked areas 8-6 to 8-7

profiling (continued)

- key features 8-2
- overview 8-3
- resetting Profile window 8-25, 12-58
- restrictions 8-4
- resuming a session 8-19, 12-39
- running a session
 - description* 8-17 to 8-19
 - full* 8-17 to 8-18, 12-36
 - quick* 8-17 to 8-18, 12-38
- saving statistics
 - all views* 8-27, 12-57
 - current view* 8-28, 12-57
- stopping point
 - adding* 8-15, 12-45
 - deleting* 8-16, 12-46, 12-52
 - description* 8-15 to 8-16
 - listing* 12-50
 - resetting* 8-16, 12-52
- strategy 8-3
- switching to profile mode 12-39
- viewing data
 - associated code* 8-25 to 8-26
 - description* 8-20 to 8-26
 - displaying areas* 8-24 to 8-25, 12-65
 - displaying different data* 8-21 to 8-22, 12-65
 - sorting data* 8-23, 12-65

program

- debugging 1-13
- entry point
 - finding* 6-2
 - resetting* 12-43
- halting execution 2-18, 6-13, 12-42
- preparation for debugging 2-2
- running 6-4 to 6-5

program address breakpoint value pseudoregister (ADR) 10-13

program memory, adding to memory

- map 4-10 to 4-18

PROM keyword 4-10

PROMPT command 12-40

PROTECT keyword 4-10

protected area of memory 4-2

PRUN command 11-7, 12-40

PRUNF command 11-7

pseudoregisters 7-16 to 7-17, 10-13 to 10-15

PSTEP command 11-7, 12-41
 with breakpoints 11-7
 ptr data type 7-23
 pulldown menus, definition D-5

Q

quick profile 8-17, 12-38
 QUIT command 2-18, 12-42

R

R keyword 4-10
 R|W keyword 4-10
 RAM initialization model 2-10
 RAM keyword 4-10
 RECONNECT command 12-42
 reconnecting to emulator 6-13, 12-42
 recording Command window displays 3-12 to 3-13, 12-20
 reference/grouping operators 13-2
 registers
 CLK pseudoregister 6-12
 displaying/modifying 7-13 to 7-17
 pseudoregisters 7-16 to 7-17
 referencing by name 13-4
 reordering in the CPU window 7-14
 single-precision floating-point 7-16
 related documentation v to vi
 relational operators
 conditional execution 6-11
 description 13-2
 relative pathnames 12-17, A-3
 RELOAD command 5-3, 12-42
 repeating commands 11-13 to 11-14, 12-12
 RESET command 12-43
 resetting
 emulator 2-6, 6-7
 memory map 12-34
 program entry point 12-43
 simulator 6-7
 target system 6-7, 12-43
 RESTART (REST) command 12-43
 Restart toolbar icon 6-2

restrictions
 breakpoints 6-15
 C expressions 13-4
 debugging modes 2-17
 memory mapping 4-4
 profiling environment 8-4
 RETURN (RET) command 12-43
 Return toolbar icon 6-6
 ripple-carry output signal, definition D-5
 ROM keyword 4-10
 run commands
 HALT command 6-13, 12-25
 PESC command 11-8, 12-36
 PHALT command 11-8, 12-37
 PRUN command 11-7, 12-40
 PRUNF command 11-7
 PSTEP command 11-7, 12-41
 RUN command 6-4, 9-8, 10-10, 12-44
 RUNF command 6-6, 10-10, 12-41, 12-45
 summary 12-8
 run cycles data 8-25
 Run to Cursor context menu option 6-5
 Run toolbar icon 6-4, 8-17, 9-8, 10-10
 RUNB command
 affecting analysis 10-10
 description 12-45
 using to count clock cycles 6-12
 RUNF command 6-6, 12-41, 12-45
 running programs
 conditionally 6-11
 continuous run 6-6
 defining a starting point 6-2
 halting execution 6-13
 program entry point 6-2 to 6-3
 running code in current C function 6-6
 running entire program 6-4 to 6-5
 single-stepping 6-8 to 6-10
 through breakpoints 6-6
 up to a specific point 6-5
 while disconnected from the target system 6-6
 with analysis enabled 9-8, 10-10

S

s (ASCII string) display format 7-22
 -s debugger option 2-13, 5-3
 SA command 12-45
 Save Breakpoint File dialog box 6-18

- Save List menu option (breakpoints) 6-18
- Save Memory to COFF File dialog box 7-10
- Save Profile File dialog box 8-28
- Save Profile View File dialog box 8-27
- saving breakpoint settings 6-18
- saving memory contents to a COFF file 7-10
- saving profile data 8-27 to 8-28
- scalar type, definition D-5
- scan path linker B-3
 - device type B-3
 - example B-4
- SCONFIG command 12-46
- screen-customization commands 12-4
- scroll bar, definition D-5
- scroll bar handle
 - definition D-5
 - description 5-5, 7-7
- scrolling, definition D-5
- SD command 12-46
- section, definition D-5
- Select Areas context menu option 8-8, 8-10, 8-11
- selecting big-endian format (*-me* option) 2-11
- SEND command 11-6, 12-46 to 12-47
- serial ports, connecting an I/O port 4-14 to 4-15
- SET command 11-4 to 11-5, 12-47 to 12-48
 - adding processors to a group 11-4, 12-48
 - changing the PDM prompt 11-19, 12-48
 - creating system variables 11-18 to 11-19, 12-48
 - concatenating strings* 11-18
 - substituting strings* 11-19
 - defining a group 11-4, 12-47
 - defining the default group 11-4, 12-47
 - listing all groups 11-5, 12-48
 - listing system variables 11-20
- Set Up Hardware Breakpoints dialog box 10-8
- SETF command 7-22 to 7-26, 12-49
- setting a hardware breakpoint 10-8
- setting a software breakpoint 6-15 to 6-16, 8-15
- Setup menu
 - Alias Commands option 3-2
 - Breakpoints option
 - clearing a breakpoint* 6-17
 - loading breakpoint settings* 6-19
 - saving breakpoint settings* 6-18
 - setting a breakpoint* 6-16
 - Watch Variable option 6-12, 7-15, 7-19
- shell options, debugger 2-2
- shell program (cl6x) 2-2
- short data type 7-23
- side effects
 - definition D-5
 - description 7-5 to 7-6, 13-3
 - valid operators 7-6
- \$\$SIM\$\$ constant 3-8
- sim6x command, options 2-7, 2-10 to 2-13
- sim6x.cfg, changing from the default file 2-11
- siminit.cmd file 2-3, 2-13, 4-11, A-1
- simulating interrupts 4-16
- simulator
 - definition D-5
 - external interrupts 4-16 to 4-18
 - fast version
 - about* 1-10
 - debugger features not supported* 1-11
 - floating-point version
 - about* 1-10
 - debugger features not supported* 1-11
 - I/O memory 4-14 to 4-15, 12-31, 12-33
 - invoking the debugger 2-7 to 2-9
 - standalone* 2-7
 - limited versions 1-10
 - resetting 6-7
 - \$\$SIM\$\$ constant 3-8
- simulator configuration, naming an alternate file 2-11
- single-precision floating-point registers 7-16
- Single Step C toolbar icon 6-9
- single-step commands
 - CNEXT command 6-10, 12-18
 - CSTEP command 6-9, 12-18
 - NEXT command 6-10, 12-35
 - PSTEP command 12-41
 - STEP command 6-9, 12-53
- single-step execution
 - and function calls 6-10, 12-18, 12-35, 12-53
 - assembly language code 6-8 to 6-9, 12-53
 - C code 6-8 to 6-20, 12-18
 - definition D-6
 - description 6-8 to 6-10
 - in parallel 12-41
- single-step
 - commands, PSTEP command 11-7
 - execution, in parallel 11-7
- single-stepping, interrupts ignored 6-8
- SIZE command 12-50

- sizeof operator 13-4
- sizing a window
 - description 12-50
 - while moving it 12-34
- SL command 12-50
- SLOAD command
 - description 12-50
 - effect on Watch window 7-21
 - s debugger option 2-13
- software breakpoints. *See* breakpoints (software)
- software reset 6-7
- sorting profile data 8-23
- SOUND command 12-51, C-2
- space key, displaying data in structures or arrays 7-20
- SPAWN command 2-8 to 2-9, 12-51
 - options 2-9, 2-10 to 2-13
 - n 2-9, 2-11, 12-51
 - p 2-12
- SPL device type B-3
- SR command 12-52
- SSAVE command 12-52
- starting point for program execution 6-2 to 6-3
- STAT command 11-8, 11-20, 12-48, 12-53
- status bar, definition D-6
- STEP command 6-9, 12-53
- Step toolbar icon 6-9
- stopping point for profiling
 - adding 8-15, 12-45
 - deleting 8-16, 12-46, 12-52
 - description 8-15 to 8-16
 - listing 12-50
 - resetting 8-16, 12-52
- store instructions, displayed by the emulator 7-2
- strategy for profiling 8-3
- structures
 - direct reference operator 13-2
 - indirect reference operator 13-2
- switch settings, I/O address space 2-12
- symbol table
 - definition D-6
 - loading object code with global symbols only (–v) 2-13
 - loading object code without (–v) 5-3
 - loading object code without (RELOAD) 12-42
 - loading without object code 2-13, 5-3, 12-50

- symbolic addresses 7-8
- SYSTEM command 3-5 to 3-13, 12-54
 - PDM version 11-16
- system commands
 - ALIAS command, PDM version 11-15 to 11-16
 - DLOG command, PDM version 11-10
 - ECHO command, PDM version 11-12
 - entering from command line 3-5
 - entering several from system shell 3-6
 - IF/ELIF/ELSE/ENDIF commands 11-10 to 11-12, 12-26 to 12-61
 - LOOP/BREAK/CONTINUE/ENDLOOP commands 11-11 to 11-22, 12-28 to 12-29
 - PAUSE command 11-13
 - RECONNECT command 12-42
 - summary 12-6
 - SYSTEM command, PDM version 11-16
 - TAKE command, PDM version 11-9
 - UNALIAS command, PDM version 11-15 to 11-16
- system reset 6-7
- system shell 3-5 to 3-13

T

- t debugger option
 - description 2-13
 - during debugger invocation 4-11, A-1
 - in defining a memory map 4-11
- TAKE command
 - defining a memory map 4-11
 - description 12-55
 - executing log file 11-10
 - identify new initialization file (–t option) 2-13
 - PDM version 11-9
 - reading new memory map 4-13
 - returning to the original memory map 4-12
- Target menu
 - Continuous Run option 6-6
 - Continuous Step option 6-10
 - Halt! option 6-5, 6-13
 - Next C option 6-10
 - Next option 6-10
 - Reset Target option 6-7
 - Restart option 6-2
 - Return option 6-6
 - Run Free option 6-6
 - Run option 6-4, 9-8, 10-10
 - Step C option 6-9
 - Step option 6-9

- target system
 - 'C6x 1-8
 - definition D-6
 - describing to the debugger B-1 to B-6
 - creating the board configuration file B-2 to B-6*
 - specifying the file B-6*
 - translating the file B-5*
 - disconnected from emulator 6-6
 - memory definition for debugger 4-1 to 4-18
 - resetting 6-7, 12-43
- terminating the debugger 2-18, 12-42
- text files, displaying 5-6
- TMS320C6x device type B-3
- Toggle Breakpoint context menu option 6-15, 6-17
- toolbar, in basic display 1-3
- totem-pole output, definition D-6
- type casting 13-4
- type checking 7-3

U

- u (unsigned decimal) display format 7-22
- uchar data type 7-23
- uint data type 7-23
- ulong data type 7-23
- UNALIAS command 12-56
 - PDM version 11-15 to 11-16
- UNIX, using with the PDM 2-8
- unmarking areas 8-12
- UNSET command 11-5, 12-56
 - deleting system variables 11-20
- USE command 2-11, 12-57, A-3

V

- v debugger option 2-13
- VAA command 12-57
- VAC command 12-57
- variables
 - aggregate values in Watch window 7-18 to 7-21, 12-19
 - assigning to the result of an expression 11-19, 12-13
 - determining type 7-3
 - displaying in different numeric format 13-5

- variables (continued)
 - displaying/modifying 7-18 to 7-21
 - PDM 11-18 to 11-20
 - scalar values in Watch window 7-18 to 7-21
- VERSION command 12-58
- viewing profile data
 - description 8-20 to 8-26
 - displaying areas 8-24 to 8-25, 12-65 to 12-66
 - displaying different data 8-21 to 8-22, 12-65 to 12-66
 - sorting data 8-23, 12-65 to 12-66
 - viewing associated code 8-25 to 8-26
- void expressions 13-4
- VR command 12-58

W

- W keyword 4-10
- WA command
 - description 12-58
 - display formats 7-24, 12-59
- Watch add dialog box 7-15, 7-19
- watch commands
 - WA command 12-58
 - WD command 12-59
 - WR command 7-20, 12-61
- Watch window
 - adding items 7-19 to 7-20, 12-58
 - closing 7-20, 12-61
 - definition D-6
 - deleting items 7-20, 12-59
 - description 1-4, 7-18 to 7-21
 - displaying additional data 7-20
 - editing values 7-5
 - effect of load commands 7-21
 - labeling watched data 12-58
 - naming 7-20
 - opening 7-19 to 7-20, 12-58
- WD command 12-59
- WHATIS command 7-3, 12-60
- WIN command 12-60
- windows
 - Analysis Statistics window, analysis interface (emulator) 10-11
 - Analysis Statistics window (simulator) 9-9
 - Calls window 5-7
 - commands summary 12-4
 - CPU window 7-13 to 7-17
 - definition D-6

windows (continued)

- description 1-4 to 1-6
 - Disassembly window 5-4 to 5-5
 - File window 5-6 to 5-8
 - Memory window 7-7 to 7-12
 - moving 12-34
 - Profile window 8-20 to 8-26
 - sizing 12-50
 - summary table, debugger 1-5
 - Watch window 7-18 to 7-21
- WOM keyword 4-10
- WR command 7-20, 12-61

X

- x (hexadecimal) display format 7-22
- x debugger option 2-13
- X Window System, displaying debugger on a different machine 2-10
- XCNT pseudoregister 10-14

Z

- ZOOM command 12-61
- zooming a window 12-61