

TMS320C2x
C Source Debugger
User's Guide

Literature Number: SPRU070
March 1991



IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its products 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.

Customers are responsible for their applications using TI components.

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

TI assumes no liability for applications assistance or customer product design. TI does not 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 products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: [Standard Terms and Conditions of Sale for Semiconductor Products](http://www.ti.com/sc/docs/stdterms.htm). www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Read This First

What Is This Book About?

This book tells you how to use the TMS320C2x C source debugger with these debugging tools:

- 'C2x software development system (SWDS)
- 'C2x 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. The SWDS version won't work with the simulator, and vice versa. Separate installation chapters are provided for each tool. Be sure to install the correct version of the debugger for your environment.

How to Use This Manual

The goal of this book is to help you install the C source debugger and learn how to use it. This book is divided into three distinct parts:

- Part I: Hands-On Information** is presented first so that you can start using your debugger the same day you receive it.
 - There are two versions of the debugger—one for the SWDS and one for the simulator—and two sets of installation instructions (Chapters 1 and 2, respectively).
 - Chapter 3 is a tutorial that introduces you to many of the debugger features.
- Part II: Debugger Description** contains detailed information about using the debugger.
 - Chapter 4 is analogous to a traditional manual introduction. It lists the key features of the debugger, describes additional 'C2x software tools, and tells you how to prepare a 'C2x program for debugging.

- The remaining chapters in Part II detail the individual topics that are introduced in the tutorial. For example, Chapter 5 describes all of the debugger's windows and tells you how to move them and size them; Chapter 6 describes everything you need to know about entering commands.
- **Part III: Reference Material** provides supplementary information.
 - Chapter 12 provides a complete reference to all the tasks introduced in Parts I and II. This includes a functional and an alphabetical reference of the debugger commands and a topical reference of function key actions.
 - Chapter 13 provides information about C expressions. The debugger commands are powerful because they accept C expressions as parameters; however, the debugger can also be used to debug assembly language programs. The information about C expressions will aid assembly language programmers who are unfamiliar with C.
 - Part III also includes a glossary and an index.

The way you use this book should depend on your experience with similar products. As with any book, it would be best for you to begin on page 1 and read to the end. Because most people don't read technical manuals from cover to cover, here are some suggestions about what you should read.

- If you have used TI development tools or other debuggers before, then you may want to:
 - Use the appropriate installation chapter (Chapter 1 if you're using the SWDS; Chapter 2 if you're using the simulator).
 - Complete the tutorial in Chapter 3.
 - Read the alphabetical command reference in Chapter 12.
- If this is the first time that you have used a debugger or similar tool, then you may want to:
 - Use the appropriate installation chapter (Chapter 1 if you're using the SWDS; Chapter 2 if you're using the simulator).
 - Complete the tutorial in Chapter 3.
 - Read all of the chapters in Part II.

Notational Conventions

This document uses the following conventions.

- The TMS320C25 and TMS320C26 processors are referred to collectively as the '**C2x generation**'.
- The C source debugger has a very flexible command-entry system; there are usually a variety of ways to perform any specific action. For example, you may be able to perform the same action by typing in a command, using the mouse, or using function keys. This document uses three symbols to identify the methods that you can use to perform an action:

Symbol

Description



Identifies an action that you perform by using the mouse.



Identifies an action that you perform by using function keys.



Identifies an action that you perform by typing in a command.

- The following symbols identify mouse actions. For simplicity, these symbols represent a mouse with two buttons. However, you can use a mouse with only one button or a mouse with more than two buttons.

Symbol Action



Point. Without pressing a mouse button, move the mouse to point the cursor at a window or field on the display. (Note that the mouse cursor displayed on the screen is not shaped like an arrow; it's shaped like a block.)



Press and hold. Press a mouse button. If your mouse has only one button, press it. If your mouse has more than one button, press the left button.



Release. Release the mouse button that you pressed.



Click. Press a mouse button and, without moving the mouse, release the button.



Drag. While pressing the left mouse button, move the mouse.

- Debugger commands are not case sensitive; you can enter them in lowercase, 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, interactive displays, and window contents 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 giant	struct zzz giant[100];
whatis xxx	<pre> struct xxx { int a; int b; int c; int f1 : 2; int f2 : 4; struct xxx *f3; int f4[10]; } </pre>

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 COMMAND window display area.

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

wa *expression* [, *label*]

wa is the command. This command has two parameters, indicated by *expression* and *label*. The first parameter must be an actual C expression; the second parameter, which can be any string of characters, is optional.

- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you don’t enter the brackets themselves. Here’s an example of a command that has an optional parameter:

run [*expression*]

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

Information About Cautions

This is an example of a caution statement.
A caution statement describes a situation that could potentially damage your software or equipment.

Please read each caution statement carefully.

Related Documentation From Texas Instruments

The following books describe the TMS320C2x devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Customer Support Center (CRC) at (800) 232-3200. When ordering, please identify the book by its title and literature number.

The ***TMS320C2x User's Guide*** (literature number SPRU014B) discusses hardware aspects of the TMS320C2x generation devices, including the TMS32020, TMS320C25, and TMS320C26. Topics in this user's guide include pin functions, architecture, stack operation, and interfaces; the manual also includes a full discussion and summary of the TMS320C2x assembly language instruction set.

The ***TMS320C2x Software Development System Technical Reference*** (literature number SPRU019A) contains an overview of the key features, step-by-step installation procedures, hardware and software system requirements, equations that were developed for programming the programmable logic arrays that are used on the TMS320C2x SWDS, troubleshooting procedures, error code listings, and schematics for the SWDS board.

The ***TMS320 Fixed-Point DSP Assembly Language Tools User's Guide*** (literature number SPRU018) describes the assembly language tools (assembler, linker, archiver, and object code converter utility), assembler directives, macros, common object file format (COFF), and symbolic debugging directives.

The ***TMS320C2x C Compiler Reference Guide*** (literature number SPRU024B) tells you how to use the TMS320C2x C Compiler. This C compiler accepts standard Kernighan and Ritchie C source code and produces TMS320C2x assembly language source code.

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

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

If You Need Assistance. . .

<i>If you want to. . .</i>	<i>Do this. . .</i>
Request more information about Texas Instruments digital signal processing (DSP) products	Call the CRC† hotline: (800) 232–3200 Or write to: Market Communications Manager Texas Instruments Incorporated P.O. Box 1443, MS 736 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call the CRC† hotline: (800) 232–3200
Ask questions about product operation or report suspected problems	Call the DSP hotline: (713) 274–2320
Report mistakes in this document or any other TI documentation	Send your comments to Technical Publications Manager Texas Instruments Incorporated P.O. Box 1443, MS 702 Houston, Texas 77251–1443

† Texas Instruments Customer Response Center

Trademarks

PC-DOS is a trademark of International Business Machines.

MS-DOS is a trademark of Microsoft Corporation.

VEGA Deluxe is a trademark of Video Seven Incorporated.

VAX and VMS are trademarks of Digital Equipment Corporation.

Sun-3, Sun-4, and SunView are trademarks of Sun Microsystems, Incorporated.

Contents

Part I: Hands-On Information

1 Installing the SWDS and the C Source Debugger 1-1

Lists the hardware and software you'll need to install and run the SWDS with the debugger, guides you through a 3-step installation process, and tells you how to invoke the SWDS version of the debugger.

1.1	What You'll Need	1-2
	Hardware checklist	1-2
	Software checklist	1-3
1.2	Step 1: Installing the SWDS Board in Your PC	1-4
	Preparing the SWDS board for installation	1-4
	SWDS memory map	1-5
	Selecting wait state (jumper P5)	1-7
	SWDS board installation	1-7
1.3	Step 2: Installing the Debugger Software	1-9
1.4	Step 3: Setting Up the Debugger Environment	1-9
	Invoking the new or modified batch file	1-10
	Modifying the PATH statement	1-11
	Setting up the environment variables	1-11
	Identifying the correct jumper settings	1-12
	Resetting the SWDS	1-12
1.5	Invoking the Debugger	1-13
1.6	Troubleshooting	1-15
1.7	Exiting the Debugger	1-15

2 Installing the Simulator and the C Source Debugger 2-1

Lists the hardware and software you'll need to install and run the simulator with the debugger, guides you through installation on one of three possible systems, lists debugger restrictions associated with the three systems, and tells you how to invoke the simulator version of the debugger.

- 2.1 Installing the Simulator on PC Systems 2-2
 - 2.1.1 What You'll Need 2-2
 - Hardware checklist 2-2
 - Software checklist 2-2
 - 2.1.2 Step 1: Installing the Simulator and Debugger Software 2-3
 - 2.1.3 Step 2: Setting Up the Debugger Environment 2-4
 - Invoking the new or modified batch file 2-5
 - Modifying the PATH statement 2-5
 - Setting up the environment variables 2-5
 - 2.1.4 Restrictions Associated With the PC Version of the Simulator 2-6
 - 2.1.5 Using the Simulator With Microsoft Windows 2-7
- 2.2 Installing the Simulator on VAX/VMS Systems 2-8
 - 2.2.1 What You'll Need 2-8
 - Hardware checklist 2-8
 - Software checklist 2-8
 - 2.2.2 Installing the Simulator and Debugger Software 2-9
 - 2.2.3 Restrictions Associated With the VMS Version of the Simulator 2-10
- 2.3 Installing the Simulator on Sun Systems 2-11
 - 2.3.1 What You'll Need 2-11
 - Hardware checklist 2-11
 - Software checklist 2-11
 - 2.3.2 Installing the Simulator and Debugger Software 2-12
 - 2.3.3 Restrictions Associated With the Sun Version of the Simulator 2-12
- 2.4 Invoking the Debugger 2-13
- 2.5 Exiting the Debugger 2-14

3 An Introductory Tutorial to the C Source Debugger 3-1

This chapter provides a step-by-step introduction to the debugger and its features.

- How to use this tutorial 3-2
- A note about entering commands 3-3
- An escape route (just in case) 3-3
- Invoke the debugger and load the sample program's object code 3-4
- Take a look at the display. 3-5
- What's in the DISASSEMBLY window? 3-6
- Select the active window 3-6
- Resize the active window 3-8
- Move the active window 3-9
- Scroll through a window's contents 3-10
- Display the C source version of the sample file 3-11

Execute some code	3-11
Become familiar with the three debugging modes	3-12
Open another text file, then redisplay a C source file	3-14
Use the basic run command	3-15
Set some breakpoints	3-15
Benchmark a section of code (simulator)	3-17
Watch some values and single-step through code	3-18
Run code conditionally	3-20
WHATIS that?	3-21
Clear the COMMAND window display area	3-22
Display the contents of an aggregate data type	3-22
Display data in another format	3-25
Change some values	3-26
Define a memory map	3-27
Close the debugger	3-28

Part II: Debugger Description

4 Overview of a Code Development and Debugging System 4-1

Discusses features of the debugger and additional tools.

4.1 Description of the 'C2x C Source Debugger	4-2
Key features of the debugger	4-3
4.2 Developing Code for the 'C2x	4-5
4.3 Preparing Your Program for Debugging	4-8
4.4 Debugging 'C2x Programs	4-10

5 The Debugger Display 5-1

Describes the default displays, tells you how to switch between assembly language and C debugging, describes the various types of windows on the display, and tells you how to move and size the windows.

5.1 Debugging Modes and Default Displays	5-2
Auto mode	5-2
Assembly mode	5-3
Mixed mode	5-4
Restrictions associated with debugging modes	5-4
5.2 Descriptions of the Different Kinds of Windows and Their Contents	5-5
COMMAND window	5-6
DISASSEMBLY window	5-7
FILE window	5-8
CALLS window	5-9
MEMORY window	5-11
CPU window	5-13
DISP windows	5-14
WATCH window	5-15

- 5.3 Cursors 5-16
- 5.4 The Active Window 5-17
 - Identifying the active window 5-17
 - Selecting the active window 5-18
- 5.5 Manipulating Windows 5-20
 - Resizing a window 5-20
 - Moving a window 5-22
- 5.6 Manipulating a Window's Contents 5-25
 - Scrolling through a window's contents 5-25
 - Editing the data displayed in windows 5-27
- 5.7 Closing a Window 5-28

6 Entering and Using Commands 6-1

Describes the rules for entering commands from the command line, tells you how to use the pulldown menus and dialog boxes (for entering parameter values), describes general information about entering commands from batch files, and describes the use of DOS-like system commands.

- 6.1 Entering Commands From the Command Line 6-2
 - How to type in and enter commands 6-3
 - Sometimes, you can't type a command 6-4
 - Using the command history 6-4
 - Clearing the display area 6-5
- 6.2 Using the Menu Bar and the Pulldown Menus 6-6
 - Using the pulldown menus 6-7
 - Escaping from the pulldown menus 6-8
 - Entering parameters in a dialog box 6-9
 - Using menu bar selections that don't have pulldown menus 6-10
 - How the menu selections correspond to commands 6-11
- 6.3 Entering Commands From a Batch File 6-13
- 6.4 Additional System Commands 6-14

7 Defining a Memory Map 7-1

Contains instructions for setting up a memory map that will enable the debugger to correctly access target memory, includes hints about using batch files, and tells you how to simulate I/O ports for use with the simulator or SWDS version of the debugger.

- 7.1 The Memory Map: What It Is and Why You Must Define It 7-2
- 7.2 A Sample Memory Map 7-3
 - Defining a memory map for the simulator 7-3
 - Defining a memory map for the SWDS 7-4
- 7.3 Identifying Usable Memory Ranges 7-6
- 7.4 Enabling Memory Mapping 7-7
- 7.5 Checking the Memory Map 7-7
- 7.6 Modifying the Memory Map During a Debugging Session 7-8
 - Returning to the original memory map 7-9
- 7.7 Simulating I/O Space 7-10

Connecting an I/O port	7-10
Observing serial port data	7-11
Configuring memory to use serial port simulation	7-12
Disconnecting an I/O port	7-13
8 Loading, Displaying, and Running Code	8-1
<i>Tells you how to use the three debugger modes to view the type of source files that you'd like to see, how to load source files and object files, how to run your programs, and how to halt program execution.</i>	
8.1 Code-Display Windows:	
Viewing Assembly Language Code, C Code, or Both	8-2
Selecting a debugging mode	8-3
8.2 Displaying Your Source Programs (or Other Text Files)	8-4
Displaying assembly language code	8-4
Displaying C code	8-6
Displaying other text files	8-7
8.3 Loading Object Code	8-8
Loading code while invoking the debugger	8-8
Loading code after invoking the debugger	8-8
8.4 Where the Debugger Looks for Source Files	8-9
8.5 Running Your Programs	8-10
Defining the starting point for program execution	8-10
Running code	8-11
Single-stepping through code	8-12
Running code while disconnected from the target system	8-14
Running code conditionally	8-15
8.6 Halting Program Execution	8-16
8.7 Benchmarking	8-17
Benchmarking with the simulator	8-17
Benchmarking with the SWDS	8-18
9 Managing Data	9-1
<i>Describes the data-display windows and tells you how to edit data (memory contents, register contents, and individual variables).</i>	
9.1 Where Data Is Displayed	9-2
9.2 Basic Commands for Managing Data	9-2
9.3 Basic Methods for Changing Data Values	9-4
Editing data displayed in a window	9-4
Advanced “editing”—using expressions with side effects	9-5
9.4 Managing Data in Memory	9-6
Displaying memory contents	9-6
Displaying program memory	9-7
Displaying memory contents while you're debugging C	9-8
Saving memory values in a file	9-9
Filling a block of memory	9-10

9.5	Managing Register Data	9-11
	Displaying register contents	9-11
9.6	Managing Data in a DISP (Display) Window	9-12
	Displaying data in a DISP window	9-12
	Closing a DISP window	9-14
9.7	Managing Data in a WATCH Window	9-15
	Displaying data in the WATCH window	9-15
	Deleting watched values and closing the WATCH window	9-16
9.8	Managing Signal Information (Simulator Only)	9-17
	Monitoring the $\overline{\text{BIO}}$ pin	9-17
10	Using Breakpoints	10-1
	<i>Describes the use of software breakpoints to halt code execution.</i>	
10.1	Setting a Breakpoint	10-2
10.2	Clearing a Breakpoint	10-4
10.3	Finding the Breakpoints That Are Set	10-5
11	Customizing the Debugger Display	11-1
	<i>Contains information about the commands that you can use for customizing the display and identifies the display areas that you can modify.</i>	
11.1	Changing the Colors of the Debugger Display	11-2
	area names: common display areas	11-3
	area names: window borders	11-4
	area names: COMMAND window	11-4
	area names: DISASSEMBLY and FILE windows	11-5
	area names: data-display windows	11-6
	area names: menu bar and pulldown menus	11-7
11.2	Changing the Border Styles of the Windows	11-8
11.3	Saving and Using Custom Displays	11-9
	Changing the default display for monochrome monitors	11-9
	Saving a custom display	11-10
	Loading a custom display	11-10
	Invoking the debugger with a custom display	11-11
	Returning to the default display	11-11
11.4	Changing the Prompt	11-12

Part III: Reference Material

12 Summary of Commands and Special Keys 12-1

Provides a functional summary of the debugger commands and function keys; also provides a complete alphabetical summary of all debugger commands.

12.1	Functional Summary of Debugger Commands	12-2
	Changing modes	12-3
	Managing windows	12-3
	Performing DOS-like tasks	12-3
	Managing and displaying data	12-4
	Displaying files and loading programs	12-4
	Managing breakpoints	12-5
	Customizing the screen	12-5
	Memory mapping	12-5
	Running programs	12-6
12.2	Alphabetical Summary of Debugger Commands	12-7
12.3	Summary of Special Keys	12-36
	Editing text on the command line	12-36
	Using the command history	12-36
	Switching modes	12-37
	Halting or escaping from an action	12-37
	Displaying pulldown menus	12-37
	Running code	12-38
	Selecting or closing a window	12-38
	Moving or sizing a window	12-38
	Scrolling through a window's contents	12-39
	Editing data or selecting the active field	12-39

13 Basic Information About C Expressions 13-1

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.

13.1	C Expressions for Assembly Language Programmers	13-2
13.2	Restrictions and Features Associated With Expression Analysis in the Debugger ..	13-4
	Restrictions	13-4
	Additional features	13-4

A What the Debugger Does During Invocation A-1

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.

B Debugger Messages B-1

Describes progress and error messages that the debugger may display.

- B.1 Alphabetical Summary of Debugger Messages B-2
- B.2 Additional Instructions for Expression Errors B-18
- B.3 Additional Instructions for Hardware Errors B-18

C Registers and Pseudoregisters C-1

D Glossary D-1

Defines acronyms and key terms used in this book.

Figures

1-1.	Location of the Components on the SWDS Board	1-4
1-2.	SWDS Board Jumpers P1 through P4	1-5
1-3.	DOS-Command Setup for the Debugger	1-10
2-1.	DOS-Command Setup for the Debugger	2-4
2-2.	Keyboard Mapping for VAX/VMS Systems	2-10
4-1.	The Debugger Display	4-2
4-2.	'C2x Software Development Flow	4-5
4-3.	Steps You Go Through to Prepare a Program	4-8
5-1.	Typical Assembly Display (for Auto Mode and Assembly Mode)	5-2
5-2.	Typical C Display (for Auto Mode Only)	5-3
5-3.	Typical Mixed Display (for Mixed Mode Only)	5-4
5-4.	Default Appearance of an Active and an Inactive Window	5-17
6-1.	The COMMAND Window	6-2
6-2.	The Menu Bar in the Debugger Display	6-6
6-3.	All of the Pulldown Menus	6-6
7-1.	Definition of On-Chip Memory Maps	7-3
7-2.	Initial Memory Map Defined by the siminit.cmd File	7-3
7-3.	Example of a Memory Map That You Could Define for the Simulator	7-4
7-4.	Definition of On-Chip Memory Maps	7-4
7-5.	Initial Memory Map Defined by the dbinit.cmd File	7-5

Tables

1-1.	PC Memory Segment Selections/Jumpers Settings (P1 through P4)	1-6
1-2.	Debugger Options	1-13
2-1.	Debugger Options	2-13
5-1.	Width and Length Limits for Window Sizes	5-21
5-2.	Minimum and Maximum Limits for Window Positions	5-23
7-1.	Acceptable Memory Map Configurations for the SWDS	7-5
7-2.	Serial Port Pseudoregisters	7-12
11-1.	Colors and Other Attributes for the COLOR and SCOLOR Commands	11-2
11-2.	Summary of Area Names for the COLOR and SCOLOR Commands	11-3

Installing the SWDS and the C Source Debugger



If you are using the debugger with the 'C2x simulator, do not follow the installation instructions in this chapter—turn to Chapter 2.

This chapter helps you install the 'C2x SWDS board with the appropriate version of the C source debugger. When you complete the installation, turn to Chapter 3, *An Introductory Tutorial to the C Source Debugger*.

Topic	Page
<i>The chapter begins with checklists of the hardware and software you'll need for installing the SWDS and the debugger.</i>	1.1 What You'll Need 1-2
	Hardware checklist 1-2
	Software checklist 1-3
<i>Installing the SWDS and debugger is a 3-step process. Before you can install the SWDS board, you must select the appropriate switch settings. After installing the debugger, you must modify the DOS environment, enabling the debugger to operate properly.</i>	1.2 Step 1: Installing the SWDS Board in Your PC 1-4
	Preparing the SWDS board for installation 1-4
	SWDS memory map 1-5
	Selecting wait state (jumper P5) 1-7
	SWDS board installation 1-7
	1.3 Step 2: Installing the Debugger Software 1-9
	1.4 Step 3: Setting Up the Debugger Environment 1-9
	Invoking the new or modified batch file 1-10
	Modifying the PATH statement 1-11
	Setting up the environment variables 1-11
Identifying the correct jumper settings 1-12	
Resetting the SWDS 1-12	
<i>When you finish the installation steps, you'll need to know how to invoke and exit the debugger.</i>	1.5 Invoking the Debugger 1-13
	1.6 Troubleshooting 1-15
	1.7 Exiting the Debugger 1-15

1.1 What You'll Need

In addition to the items shipped with the 'C2x C source debugger and SWDS, you'll need the following items.

Hardware checklist

- | | | |
|--------------------------|--------------------------------------|--|
| <input type="checkbox"/> | host | An IBM PC/AT or 100% compatible ISA/EISA-bus PC with a hard-disk system and a floppy-disk drive |
| <input type="checkbox"/> | memory | Minimum of 640K (debugger occupies approximately 400K) |
| <input type="checkbox"/> | display | Monochrome or color (color recommended) |
| <input type="checkbox"/> | slot | One 16-bit slot |
| <input type="checkbox"/> | SWDS board power requirements | Approximately 3 amps @ 5 volts (15 watts) |
| <input type="checkbox"/> | optional hardware | Mouse (must be compatible with a Microsoft mouse) |
| <input type="checkbox"/> | | An EGA- or VGA-compatible graphics display card |
| <input type="checkbox"/> | | A 17" or 19" monitor. The C source debugger has several modes that allow you to display varying amounts of information on your PC monitor. If you have an EGA- or VGA-compatible graphics card and a large monitor (17" or 19"), you can take advantage of some of the debugger's larger screen modes. (To use larger screen sizes, you must invoke the debugger with the appropriate options; Table 1-2, page 1-13, explains this in detail.) |
| <input type="checkbox"/> | miscellaneous materials | A blank, formatted disk |

Software checklist

- | | | |
|--------------------------|-------------------------|--|
| <input type="checkbox"/> | operating system | MS-DOS or PC-DOS (version 3.0 or later) |
| <input type="checkbox"/> | software tools | 'C2x C compiler, assembler, and linker |
| <input type="checkbox"/> | required files † | <i>c2xreset</i> resets the 'C2x SWDS |
| <input type="checkbox"/> | | † <i>c2xmon.out</i> is the monitor program for the 'C2x SWDS |
| <input type="checkbox"/> | optional files † | <i>dbinit.cmd</i> is a general-purpose batch file that contains debugger commands. The version of <i>dbinit.cmd</i> that's shipped with the debugger defines a 'C2x memory map. If this file isn't present when you invoke the debugger, then all memory is invalid at first. When you first start using the debugger, this memory map should be sufficient for your needs. Later, you may want to define your own memory map. For information about setting up your own memory map, refer to Chapter 7, <i>Defining a Memory Map</i> . |
| <input type="checkbox"/> | | † <i>init.clr</i> is a general-purpose screen configuration file. If <i>init.clr</i> isn't present when you invoke the debugger, the debugger uses the default screen configuration.

The default configuration is for color monitors; an additional file, <i>mono.clr</i> , can be used for monochrome monitors. When you first start to use the debugger, the default screen configuration should be sufficient for your needs. Later, you may want to define your own custom configuration. For information about these files and about setting up your own screen configuration, refer to Chapter 11, <i>Customizing the Display</i> . |
| <input type="checkbox"/> | | † <i>swdsdiag.exe</i> uses <i>c2xreset</i> while it checks the hardware of the SWDS. This file is included for hardware diagnostics and is discussed in the <i>TMS320C2x Software Development System Technical Reference</i> (literature number SPRU019A). |

† Included as part of the debugger package

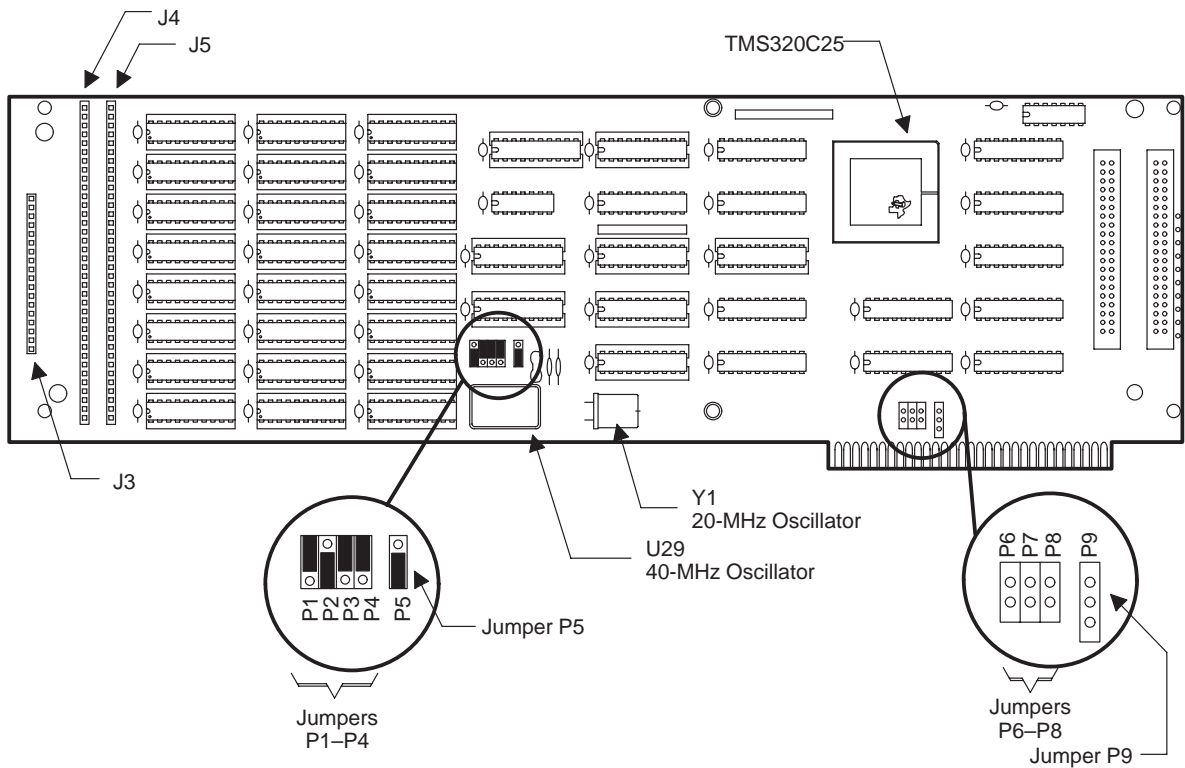
1.2 Step 1: Installing the SWDS Board in Your PC

This section contains the hardware installation information for the SWDS.

Preparing the SWDS board for installation

The SWDS board is shipped as shown in Figure 1–1. Jumpers P1–P4 are set for segment D. Jumper P5 is set for no wait state. A TMS320C25 is installed in socket U40, and a canned 40-MHz oscillator is installed in socket U29 (for operation with the onboard 'C25). Figure 1–1 shows the location of the jumpers, connectors, 'C25, and crystal oscillator on the SWDS board.

Figure 1–1. Location of the Components on the SWDS Board



Jumpers P1–P4 allow you to select the appropriate decoded memory segment address for your PC. The proper settings for these jumpers depend on which PC memory segment address (1–9, A–F) you select.

Jumper P5 allows you to select either one wait state or no wait state.

Jumpers P6–P9 are reserved.

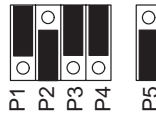
Connectors J3–J5 are reserved.

SWDS memory map

The SWDS occupies one segment (64K bytes) of PC memory. The segment address is decoded by jumpers P1 through P4. The memory segment is the decoded value of the four most significant PC address lines, A16–A19. The decoded memory segment is changed by setting SWDS board jumpers P1–P4 to decode the desired segment from 0–F. Table 1–1 lists all possible PC memory segment selections and their corresponding jumper settings.

















Many segments are already used by the PC for memory and screen functions. Consult your PC technical reference manual to determine segment usage. The SWDS is shipped with the jumpers set for the D segment as shown in Figure 1–2. The D segment is recommended for the IBM PC/AT.

Figure 1–2. SWDS Board Jumpers P1 through P4



Jumpers P1–P4 Set for D Segment (As Shipped)

Table 1-1. PC Memory Segment Selections/Jumpers Settings (P1 through P4)

PC Memory Segment Address	Jumper Setting P1-P4	PC Address Lines (A16-A19) Binary Values			
		A16	A17	A18	A19
0000		0	0	0	0
1000		1	0	0	0
2000		0	1	0	0
3000		1	1	0	0
4000		0	0	1	0
5000		1	0	1	0
6000		0	1	1	0
7000		1	1	1	0
8000		0	0	0	1
9000		1	0	0	1
A000		0	1	0	1
B000		1	1	0	1
C000		0	0	1	1
D000		1	0	1	1
E000		0	1	1	1
F000		1	1	1	1

Selecting wait state (jumper P5)

The 24K words of 25-ns static RAM on the SWDS allows the 'C25 to execute at a 40-MHz crystal frequency with no wait states. Jumper P5 selects between one wait and no wait states. Figure 1–1 on page 1-4 shows jumper P5 set in the no-wait-state position (as shipped). Moving jumper P5 to the opposite position extends all program and data memory accesses by one wait state; this does not extend the I/O instructions. The extension is done optionally by the target system control of the 'C25 RDY line, via the in-circuit emulation adapter.

SWDS board installation

If you plan to connect the SWDS board to a realtime target system, select two vacant PC slots side-by-side. Use the vacant slot to your left (facing the rear of the PC) for the SWDS board, and use the vacant slot to your right for the two ribbon cables that connect the SWDS board to an external target system. Feed the two ribbon cables through the open slot in the back of the PC and plug them into SWDS board connectors J1 and J2.

To install the SWDS board, perform the following steps:

- 1) Ensure that you've made any necessary changes to the memory jumpers P1 through P4.
- 2) Remove the cover of the PC. Refer to the PC user's guide instructions for removing the cover to access the I/O expansion channel card cage.
- 3) Locate the PC slot planned for the SWDS board (facing the rear of the PC). Remove the metal plate covering the chassis opening. Save the screw to secure the SWDS board to the chassis later. If you plan to connect the SWDS board to a realtime target system, remove the metal plate covering the chassis opening adjacent to the SWDS board PC slot.
- 4) If the board was shipped with the cables connected, remove the two ribbon cables connected to the SWDS board at J1 and J2. If no data logging/in-circuit emulation will be done, skip the rest of this step. If the SWDS will be connected to a realtime external target system, feed the ends of the two ribbon cables through the chassis opening adjacent to the SWDS board PC slot from outside the PC.

- 5) Align the SWDS board with the PC slot and press into place, ensuring that the board and the metal support bracket are firmly seated. If the two target-system ribbon connectors were fed through the adjacent chassis opening in the previous step, connect them to connectors J1 and J2. Plug the longer cable into J2, observing proper placement of the keyed pin. Plug the shorter cable into J1, observing proper placement of the keyed pin.
- 6) Tighten the screw in the mounting bracket hole to keep the SWDS board from working loose.
- 7) Replace the PC cover.
- 8) Turn the PC on.

1.3 Step 2: Installing the Debugger Software

This section explains the simple process of installing the debugger software on a hard-disk system. The debugger package includes a single disk that may contain multiple directories. To install the debugger, you must copy the contents directory from the product disk.

Step 1: Make a backup copy of the product disk. (If necessary, refer to the DOS manual that came with your computer.)

Step 2: On your hard disk or system disk, create a directory named *c2xhll*. This directory will contain the 'C2x C source debugger software.

```
MD C:\c2xhll
```

Step 3: Insert the product disk into drive A. Copy the contents of the disk:

```
COPY A:\*.* C:\C2xHLL\*.* /V
```

1.4 Step 3: Setting Up the Debugger Environment

To ensure that your debugger works correctly, you must:

- 1) Modify the PATH statement to identify the *c2xhll* directory.
- 2) Define environment variables so that the debugger can find the files it needs.
- 3) Identify the SWDS memory segment address.
- 4) Reset the SWDS board.



Not only must you do these things before you invoke the debugger for the first time, *you must do them any time you power up or reboot your PC.*

You can accomplish these tasks by entering individual DOS commands, but it's simpler to put the commands in a batch file. You can edit your system's *autoexec. bat* file; in some cases, modifying the *autoexec* may interfere with other applications running on your PC. So, if you prefer, you can create a separate batch file that performs these tasks.

Figure 1–3 (a) shows an example of an autoexec.bat file that contains the suggested modifications (highlighted in bold type). Figure 1–3 (b) shows a sample batch file that you could create instead of editing the autoexec.bat file. (For the purpose of discussion, assume that this sample file is named *initdb.bat*.)

Figure 1–3. DOS-Command Setup for the Debugger

(a) Sample autoexec.bat file to use with the debugger

Modifications:

```

DATE
TIME
ECHO OFF
PATH=C:\DOS;C:\dsptools;C:\c2xh11
SET D_DIR=C:\c2xh11
SET D_SRC=C:\c25code
SET D_OPTIONS=-p d000
SET C_DIR=C:\dsptools
CLS
c2xreset
    
```

PATH statement →
Environment variables and memory segment →
Reset the SWDS →

(b) Sample initdb.bat file to use with the debugger

```

PATH=C:\c2xh11;%path%
SET D_DIR=C:\c2xh11
SET D_SRC=C:\c25code
SET D_OPTIONS=-p d000
c2xreset
    
```

PATH statement →
Environment variables and memory segment →
Reset the SWDS →

Invoking the new or modified batch file

- If you modify the autoexec.bat file, be sure to invoke it before invoking the debugger for the first time. To invoke this file, enter:

`autoexec` 

- If you create an initdb.bat file, you must invoke it before invoking the debugger for the first time. After that, you'll need to invoke initdb.bat any time that you power up or reboot your PC. To invoke this file, enter:

`initdb` 

Modifying the PATH statement

Step 1: Define a path to the debugger directory. The general format for doing this is:

```
PATH=C:\c2xh11
```

This allows you to invoke the debugger without specifying the name of the directory that contains the debugger executable file.

- If you are modifying an autoexec that already contains a PATH statement, simply include `;C:\c2xh11` at the end of the statement as shown in Figure 1–3 (a).
- If you are creating an `initdb.bat` file, use a different format for the PATH statement:

```
PATH=C:\c2xh11;%path%
```

The addition of `;%path%` ensures that this PATH statement won't undo PATH statements in any other batch files (including the `autoexec.bat` file).

Setting up the environment variables

An environment variable is a special system symbol that the debugger uses for finding or obtaining certain types of information. The debugger uses three environment variables, named `D_DIR`, `D_SRC`, and `D_OPTIONS`. The next three steps tell you how to set up these environment variables. The format for doing this is the same for both the `autoexec.bat` and `initdb.bat` files.

Step 2: Set up the `D_DIR` environment variable to identify the `c2xh11` directory:

```
SET D_DIR=C:\c2xh11
```

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

This directory contains auxiliary files (`c2xreset`, `dbinit.cmd`, etc.) that the debugger needs.

Step 3: Set up the `D_SRC` environment variable to identify any directories that contain program source files that you'll want to look at while you're debugging code. The general format for doing this is:

```
SET D_SRC=C:\pathname1;\pathname2...
```

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

For example, if your 'C2x programs were in a directory named `csource`, the `D_SRC` setup would be:

```
SET D_SRC=C:\csource
```

Step 4: You can use several options when you invoke the debugger. If you use the same options over and over, it's convenient to specify them with D_OPTIONS. The general format for doing this is:

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

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

This tells the debugger to load the specified object file and use the specified options each time you invoke the debugger. These are the options that you can identify with D_OPTIONS:

```
-b[bbbb]      -p memory segment address  -i pathname  
  
-s            -t filename                -v  
-c            -c
```

For more information about options, see Section 1.5 (page 1-13). Note that you can override D_OPTIONS by invoking the debugger with the -x option.

Identifying the correct jumper settings

Step 5: You must use the debugger's -p option to identify the memory segment address that the SWDS is using, even if you are using the default memory segment address. You can use -p each time you invoke the debugger, or you can specify this information by using the D_OPTIONS environment variable. Table 1-1 on page 1-6 lists the memory segment addresses and corresponding jumper settings.

Resetting the SWDS

Step 6: To reset the SWDS, add this line to the autoexec.bat or initdb.bat file:

```
c2xreset
```

1.5 Invoking the Debugger

Here's the basic format for the command that invokes the debugger:



```
db2x [filename] [-options]
```

- db2x** is the command that invokes the debugger.
- filename* is an optional parameter that names an object file that the debugger will load into memory during invocation. The debugger looks for the file in the current directory; if the file isn't in the current directory, you must supply the entire path-name. If you don't supply an extension for the filename, the debugger assumes that the extension is .out.
- options* supply the debugger with additional information (see Table 1-2).

You can also specify filename and option information with the D_OPTIONS environment variable (see *Setting up the environment variables*, page 1-11).

Table 1-2. Debugger Options

Option	Description																		
-b[bbbb]	By default, the debugger uses an 80 character by 25 line screen. If you're using the PC version, you must also have a special graphics card installed.																		
	<table border="1"> <thead> <tr> <th>Option</th> <th>Characters/Lines</th> <th>Notes</th> </tr> </thead> <tbody> <tr> <td><i>none</i></td> <td>80 by 25</td> <td><i>This is the default display</i></td> </tr> <tr> <td>-b</td> <td>80 by 43 (PC with EGA) 80 by 50 (PC with VGA)</td> <td><i>Use any EGA or VGA card</i></td> </tr> <tr> <td>-bb</td> <td>120 by 43</td> <td rowspan="4">} <i>Currently, the debugger supports these modes on a Video Seven VEGA Deluxe card.</i></td> </tr> <tr> <td>-bbb</td> <td>132 by 43</td> </tr> <tr> <td>-bbbb</td> <td>80 by 60</td> </tr> <tr> <td>-bbbbb</td> <td>100 by 60</td> </tr> </tbody> </table>	Option	Characters/Lines	Notes	<i>none</i>	80 by 25	<i>This is the default display</i>	-b	80 by 43 (PC with EGA) 80 by 50 (PC with VGA)	<i>Use any EGA or VGA card</i>	-bb	120 by 43	} <i>Currently, the debugger supports these modes on a Video Seven VEGA Deluxe card.</i>	-bbb	132 by 43	-bbbb	80 by 60	-bbbbb	100 by 60
Option	Characters/Lines	Notes																	
<i>none</i>	80 by 25	<i>This is the default display</i>																	
-b	80 by 43 (PC with EGA) 80 by 50 (PC with VGA)	<i>Use any EGA or VGA card</i>																	
-bb	120 by 43	} <i>Currently, the debugger supports these modes on a Video Seven VEGA Deluxe card.</i>																	
-bbb	132 by 43																		
-bbbb	80 by 60																		
-bbbbb	100 by 60																		
-c	Sets memory reserved for uninitialized data to all zeros.																		
-i <i>pathname</i>	<p>-i identifies additional directories that contain your source files. Replace <i>pathname</i> with an appropriate directory name. You can specify several pathnames; use the -i option as many times as necessary:</p> <pre>db2x -i <i>path</i>₁ -i <i>path</i>₂ -i <i>path</i>₃ . . .</pre> <p>Using -i is similar to using the D_SRC environment variable (described on page 1-11). 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).</p>																		

Table 1–2. Debugger Options (Continued)

Option	Description
–p <i>memory segment address</i>	–p identifies the memory segment address that the debugger uses for communicating with the SWDS. Depending on your jumper settings, replace <i>memory segment address</i> with the correct memory segment address (found in Table 1–1 on page 1-6.)
–s	If you supply a <i>filename</i> when you invoke the debugger, you can use the –s option to tell the debugger to load only the file’s symbol table (without the file’s object code). This is similar to the debugger’s SLOAD command.
–t <i>filename</i>	–t allows you to specify an initialization command file other than dbinit.cmd. If –t is present on the command line, the file specified by <i>filename</i> will be invoked as the command file instead of dbinit.cmd.
–v	<p>This command 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 space.</p> <p>The –v option affects all loads, including loading when you invoke the debugger and loading with the LOAD command within the debugger environment.</p>
–x	–x tells the debugger to ignore any information supplied with D_OPTIONS.

1.6 Troubleshooting

If the SWDS is behaving unpredictably, or if you're receiving unexpected results, check the specific cases below. You might need to remap your SWDS memory segment or adjust your system:

PC with an expanded memory system (EMS)

The EMS standard defines a memory map swap area. If that swap area is the area where the SWDS is mapped, you'll need to remap the SWDS. For memory segment addresses and corresponding jumper settings, refer to Table 1-1 on page 1-6.

PC with extended/expanded memory managers


A memory manager could indicate that the SWDS memory segment is available, and then try to load something else into that segment. To avoid this, you should prevent the memory manager from accessing the SWDS memory segment.

PC with shadow RAM or RAM BIOS

Some systems automatically move the BIOS of the operating system or graphics card to fast RAM, and the BIOS could be moved to the SWDS memory segment. You should turn that feature off, if possible. If you can't turn that feature off, you'll need to remap the SWDS memory segment or relocate the BIOS. Table 1-1 on page 1-6 contains SWDS memory segment addresses and corresponding jumper settings.

1.7 Exiting the Debugger

To exit the debugger and return to the operating system, enter this command:

```
quit 
```

You don't need to worry about where the cursor is or which window is active—just type. If a program is running, press **(ESC)** to halt program execution before you quit the debugger.

Installing the Simulator and the C Source Debugger


STOP

If you are using the debugger with the 'C2x SWDS, do not follow the installation instructions in this chapter—turn to Chapter 1.

This chapter contains information that will help you prepare to use the simulator with the C source debugger. The simulator runs on three types of host systems:

- IBM PCs and compatibles
 - running MS-DOS or PC-DOS
 - running Microsoft Windows 3.0 (or later) on top of MS-DOS or PC-DOS
- VAX/VMS systems
- Sun-3 and Sun-4 systems

Topic	Page
<i>This section tells you how to install the PC version of the simulator and debugger.</i>	2.1 Installing the Simulator on PC Systems 2-2
	What You'll Need 2-2
	Step 1: Installing the Simulator and Debugger Software 2-3
	Step 2: Setting Up the Debugger Environment 2-4
	Restrictions Associated With the PC Version of the Simulator 2-6
	Using the Simulator With Microsoft Windows 2-7
	2.2 Installing the Simulator on VAX/VMS Systems 2-8
What You'll Need 2-8	
Installing the Simulator and Debugger Software 2-9	
Restrictions Associated With the VMS Version of the Simulator 2-10	
<i>This section tells you how to install the Sun version of the simulator and debugger. Pay special attention to the restrictions.</i>	2.3 Installing the Simulator on Sun Systems 2-12
	What You'll Need 2-12
	Installing the Simulator and Debugger Software 2-13
	Restrictions Associated With the SUN Version of the Simulator 2-13
<i>When you finish installing the simulator, you'll need to know how to invoke and exit the debugger.</i>	2.4 Invoking the Debugger 2-14
	2.5 Exiting the Debugger 2-15

2.1 Installing the Simulator on PC Systems

This section tells you how to install and set up the simulator on PC systems.

2.1.1 What You'll Need

In addition to the items shipped with the 'C2x C source debugger and simulator, you'll need the following items.

Hardware checklist

- | | | |
|--------------------------|--------------------------------|--|
| <input type="checkbox"/> | host | An IBM PC/AT or 100% compatible ISA/EISA-based PC with a hard-disk system and a floppy-disk drive |
| <input type="checkbox"/> | memory | Minimum of 640K; in addition, if you are running under Microsoft Windows, you'll need at least 256K of extended memory |
| <input type="checkbox"/> | display | Monochrome or color (color recommended) |
| <input type="checkbox"/> | optional hardware | Mouse (must be compatible with a Microsoft mouse) |
| <input type="checkbox"/> | | An EGA- or VGA-compatible graphics display card |
| <input type="checkbox"/> | | A 17" or 19" monitor. The C source debugger has several modes that allow you to display varying amounts of information on your PC monitor. If you have an EGA- or VGA-compatible graphics card and a large monitor (17" or 19"), you can take advantage of some of the debugger's larger screen modes. (To use larger screen sizes, you must invoke the debugger with the appropriate options; Table 2-1, page 2-14, explains this in detail.) |
| <input type="checkbox"/> | miscellaneous materials | A blank, formatted disk |

Software checklist

- | | | |
|--------------------------|-------------------------|--|
| <input type="checkbox"/> | operating system | MS-DOS or PC-DOS (version 3.0 or later)
Optional: Microsoft Windows 3.0 (or later) |
| <input type="checkbox"/> | software tools | 'C2x C compiler, assembler, and linker |
| <input type="checkbox"/> | optional file † | <i>siminit.cmd</i> is a general-purpose batch file that contains debugger commands. This batch file, shipped with the debugger, defines a 'C2x memory map. If this file isn't present when you invoke the debugger, then all memory is invalid at first. When you first start using the debugger, this memory map should be sufficient for your needs. Later, you may want to define your own memory map. For information about setting up your own memory map, refer to Chapter 7, <i>Defining a Memory Map</i> . |

† Included as part of the debugger package

2.1.2 Step 1: Installing the Simulator and Debugger Software

This section explains the simple process of installing the simulator and debugger on a hard-disk system. The software package includes a single disk with multiple files. To install the simulator and debugger, you must copy these files from the product disk.

Step 1: Make a backup copy of the product disk. (If necessary, refer to the DOS manual that came with your computer.)

Step 2: On your hard disk or system disk, create a directory named *sim2x*. This directory will contain the 'C2x software.

```
MD C:\sim2x
```

Step 3: Insert the product disk into drive A. Copy the contents of the product disk into the *sim2x* directory:

```
COPY A:\*.* C:\sim2x\*.* /V
```

Step 4: You must set up to use the correct executable according to whether or not you plan to use Microsoft Windows. If you plan to use Microsoft Windows, delete *sim2x.exe* from *your hard disk* (not from the *product disk*) and rename *sim2xw.exe* to *sim2x.exe*:

```
del sim2x.exe  
ren sim2xw.exe sim2x.exe
```

If you do not plan to use Microsoft Windows, delete *sim2xw.exe* from *your hard disk* (not from the *product disk*):

```
del sim2xw.exe
```

Note that if you are also using the SWDS, you may want to install the simulator package in a different directory.

2.1.3 Step 2: Setting Up the Debugger Environment

To ensure that your debugger works correctly, you must:

- Modify the PATH statement to identify the sim2x directory.
- Define environment variables so that the debugger can find the files it needs.



Not only must you do these things before you invoke the debugger for the first time, *you must do them any time you power up or reboot your PC.*

You can accomplish these tasks by entering individual DOS commands, but it's simpler to put the commands in a batch file. You can edit your system's autoexec. bat file; in some cases, modifying the autoexec may interfere with other applications running on your PC. So, if you prefer, you can create a separate batch file that performs these tasks.

Figure 2–1 (a) shows an example of an autoexec.bat file that contains the suggested modifications (highlighted in bold type). Figure 2–1 (b) shows a sample batch file that you could create instead of editing the autoexec.bat file. (For the purpose of discussion, assume that this sample file is named *initdb.bat*.)

Figure 2–1. DOS-Command Setup for the Debugger

(a) Sample autoexec.bat file to use with the debugger

Modifications:

PATH statement →

Environment variables →

```

DATE
TIME
ECHO OFF
PATH=C:\DOS;C:\dsptools;C:\sim2x
SET D_DIR=C:\sim2x
SET D_SRC=C:\c25code
SET D_OPTIONS=-b
SET C_DIR=C:\dsptools
CLS
    
```

(b) Sample initdb.bat file to use with the debugger

PATH statement →

Environment variables →

```

PATH=C:\sim2x;%path%
SET D_DIR=C:\sim2x
SET D_SRC=C:\c25code
SET D_OPTIONS=-b
    
```

Invoking the new or modified batch file

- If you modify the `autoexec.bat` file, be sure to invoke it before invoking the debugger for the first time. To invoke this file, enter:

```
autoexec 
```

- If you create an `initdb.bat` file, you must invoke it before invoking the debugger for the first time. (If you are using Microsoft Windows, invoke `initdb.bat` *before* entering Microsoft Windows.) After that, you'll need to invoke `initdb.bat` any time that you power up or reboot your PC. To invoke this file, enter:

```
initdb 
```

Modifying the PATH statement

- Step 1:** Define a path to the debugger directory. The general format for doing this is:

```
PATH=C:\sim2x
```

This allows you to invoke the debugger without specifying the name of the directory that contains the debugger executable file.

- If you are modifying an `autoexec` that already contains a `PATH` statement, simply include `;C:\sim2x` at the end of the statement as shown in Figure 2–1 (a).
- If you are creating an `initdb.bat` file, use a different format for the `PATH` statement:

```
PATH=C:\sim2x;%path%
```

The addition of `;%path%` ensures that this `PATH` statement won't undo `PATH` statements in any other batch files (including the `autoexec.bat` file).

Setting up the environment variables

An environment variable is a special system symbol that the debugger uses for finding or obtaining certain types of information. The debugger uses three environment variables, named `D_DIR`, `D_SRC`, and `D_OPTIONS`. The next three steps tell you how to set up these environment variables. The format for doing this is the same for both the `autoexec.bat` and `initdb.bat` files.

Step 2: Set up the D_DIR environment variable to identify the sim2x directory:

```
SET D_DIR=C:\sim2x
```

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

This directory contains auxiliary files (such as siminit.cmd) that the debugger needs.

Step 3: Set up the D_SRC environment variable to identify any directories that contain program source files that you'll want to look at while you're debugging code. The general format for doing this is:

```
SET D_SRC=C:\pathname1;\pathname2...
```

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

For example, if your 'C2x programs were in a directory named *csource*, the D_SRC setup would be:

```
SET D_SRC=C:\csource
```

Step 4: You can use several options when you invoke the debugger. If you use the same options over and over, it's convenient to specify them with D_OPTIONS. The general format for doing this is:

```
SET D_OPTIONS= [object filename] [debugger options]
```

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

This tells the debugger to load the specified object file and use the specified options each time you invoke the debugger. These are the options that you can identify with D_OPTIONS:

-b[bbbb]	-i <i>pathname</i>	-mv <i>version</i>
-s	-t <i>filename</i>	-v
		-c

For more information about options, see Section 2.4 (page 2-14). Note that you can override D_OPTIONS by invoking the debugger with the -x option.

2.1.4 Restrictions Associated With the PC Version of the Simulator

Note that these restrictions do not apply if you are using Microsoft Windows.

The size of the PC version of the simulator limits the size of memory that can be configured in the memory map. (Memory mapping is described in detail in Chapter 7, *Defining a Memory Map*.) You can configure a maximum of 56K words in any combination of program and data memory.

For example, you could configure 24K words of program memory and 32K words of data memory with these memory-mapping commands:

```
ma 0,0,0x6000, RAM           Add 24K of program memory
ma 0x200,1,0x6000, RAM      Add 24K of data memory
ma 0x8000, 1, 0x2000, RAM  Add 8K of data memory
```

2.1.5 Using the Simulator With Microsoft Windows

If you're using Microsoft Windows, you can freely move or resize the debugger display on the screen. If the resized display is bigger than the debugger requires, the extra space is not used. If the resized display is smaller than required, the display is clipped. Note that when the display is clipped, it can't be scrolled.

You may want to create an icon to make it easier to invoke the debugger from within the Microsoft Windows environment. Refer to your Microsoft Windows manual for details.

You should run Microsoft Windows in either the *standard mode* or the *386 enhanced mode* to get the best results and to avoid the restrictions described in subsection 2.1.4.

2.2 Installing the Simulator on VAX/VMS Systems

This section tells you how to install and set up the simulator on VMS systems.

2.2.1 What You'll Need

In addition to the items shipped with the 'C2x C source debugger and simulator, you'll need the following items.

Hardware checklist

- | | | |
|--------------------------|----------------|--|
| <input type="checkbox"/> | host | A DEC VAX system with a 9-track tape drive |
| <input type="checkbox"/> | display | VT100 or equivalent |

Software checklist

- | | | |
|--------------------------|-------------------------|--|
| <input type="checkbox"/> | operating system | VMS (version 4.5 or later) |
| <input type="checkbox"/> | software tools | 'C2x C compiler, assembler, and linker |
| <input type="checkbox"/> | optional files † | <i>siminit.cmd</i> is a general-purpose batch file that contains debugger commands. This batch file, shipped with the debugger, defines a 'C2x memory map. If this file isn't present when you invoke the debugger, then all memory is invalid at first. When you first start using the debugger, this memory map should be sufficient for your needs. Later, you may want to define your own memory map. For information about setting up your own memory map, refer to Chapter 7, <i>Defining a Memory Map</i> . |
| <input type="checkbox"/> | † | <i>clrs.dat</i> is a general-purpose screen configuration file. If <i>clrs.dat</i> isn't present when you invoke the debugger, the debugger uses the default screen configuration. For information about these files and about setting up your own screen configuration, refer to Chapter 11, <i>Customizing the Display</i> . |

† Included as part of the debugger package

2.2.2 Installing the Simulator and Debugger Software

This section explains the simple process of installing the simulator and debugger on a hard disk system. The software package is shipped on a 1600-bpi magnetic tape. To install the simulator and debugger, you must restore the directory from the tape.

Step 1: Select the VMS device name for the tape drive. These installation instructions use the name *MFA0* (which is a typical name for a tape drive), but you should use the name that you have selected.

Step 2: Mount the tape on an appropriate VAX tape drive. Be sure that the tape is ONLINE and at LOAD POINT before proceeding.

Step 3: Mount the tape drive:

```
ALLOC MFA0: 
MOUNT/FOR/DEN=1600 MFA0: 
```

If the mounting is successful, you will see this message:

```
SIM2x MOUNTED ON MFA0
```



Step 4: Create a directory named *sim2x* to contain the 'C2x simulator and debugger software:

```
CREATE/DIR DUA0:[SIM2X] 
```

Step 5: Copy the files from tape to disk:

```
BACKUP/LOG/VERIFY MFA0:SIM2X.BCK DUA0:[SIM2X...]*.* 
```

Step 6: After the files are successfully copied, rewind the tape and free the tape drive:

```
DISMOUNT MFA0: 
DEALLOCATE MFA0: 
```

Step 7: Place the following commands in your LOGIN.COM file:

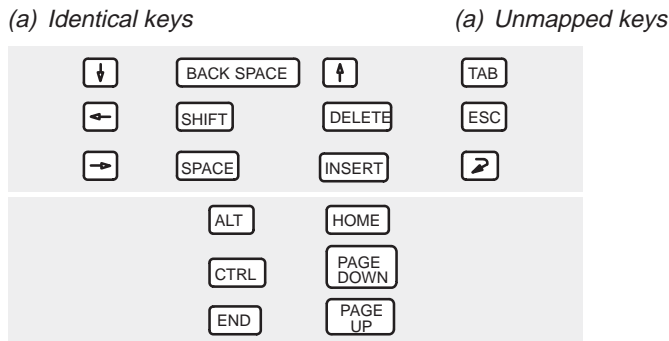
```
DEFINE IPCDIR DUA0:[SIM2X]
SIM2x ::= $IPCDIR:SIM2X
```

2.2.3 Restrictions Associated With the VMS Version of the Simulator

Several restrictions are associated with the VAX/VMS version of the debugger interface. These restrictions, listed below, override the information presented in Parts II and III of this manual.

- This version of the debugger works with VT100-compatible terminals only. Many descriptions of color monitors and color features do not apply.
- All unconfigured memory regions are displayed in reverse video.
- The C source debugger has a very flexible command-entry system; there are usually a variety of ways to perform any specific action. For example, you may be able to perform the same action by typing in a command, using the mouse, or using a function key. However:
 - The VAX/VMS system does not use a mouse; therefore, you cannot use the mouse methods described in this book.
 - As described throughout this manual, keys used for special purposes refer to a standard PC keyboard, which differs from VAX keyboards. Some of the keys are identical and have the same functions on both keyboards (Figure 2-2 a); some have no equivalent (Figure 2-2 b), and others are mapped (Figure 2-2 c).
- The simulator is an instruction-level simulator that does not simulate the pipeline.

Figure 2-2. Keyboard Mapping for VAX/VMS Systems



(b) Mapped keys

PC Key Sequence	Mapped to This key on the Numeric Pad of the VAX Keyboard	PC Key	Mapped to This Key on the Numeric Pad of the VAX Keyboard	PC Key	Mapped to This Key on the Numeric Pad of the VAX Keyboard
ALT L	PF1	F1	1	F6	6
ALT B	PF2	F2	2	F7	7
ALT W	PF3	F3	3	F8	8
ALT M	PF4	F4	4	F9	9
ALT C	_	F5	5	F10	10
ALT D	'				

2.3 Installing the Simulator on Sun Systems

This section tells you how to install and set up the simulator on Sun-3 and Sun-4 systems.

2.3.1 What You'll Need

In addition to the items shipped with the 'C2x C source debugger and simulator, you'll need the following items.

Hardware checklist

- | | | |
|--------------------------|--------------------------|---|
| <input type="checkbox"/> | host | A Sun-3 or Sun-4 system (running SunView) with a cartridge tape drive |
| <input type="checkbox"/> | display | Monitor running SunView |
| <input type="checkbox"/> | optional hardware | Mouse |

Software checklist

- | | | |
|--------------------------|-------------------------|--|
| <input type="checkbox"/> | operating system | Sun OS (running SunView) |
| <input type="checkbox"/> | software tools | 'C2x C compiler, assembler, and linker |
| <input type="checkbox"/> | optional file † | <i>siminit.cmd</i> is a general-purpose batch file that contains debugger commands. This batch file, shipped with the debugger, defines a 'C2x memory map. If this file isn't present when you invoke the debugger, then all memory is invalid at first. When you first start using the debugger, this memory map should be sufficient for your needs. Later, you may want to define your own memory map. For information about setting up your own memory map, refer to Chapter 7, <i>Defining a Memory Map</i> . |

† Included as part of the debugger package

2.3.2 Installing the Simulator and Debugger Software

This section explains the simple process of installing the simulator and debugger on a hard-disk system. The software package is shipped on a cartridge tape. To install the simulator and debugger, you must restore the directory from the tape.

Step 1: Insert the product tape in a cartridge tape drive.

Step 2: Create a directory named *sim2x* to contain the 'C2x simulator and debugger software:

```
mkdir sim2x
```

Step 3: Make *sim2x* the current directory:

```
cd sim2x
```

Step 4: Copy the files from tape to disk:

```
tar xvf /dev/rst8
```

Note that the *sim2x* directory will contain copies of both the Sun3 (*Sun3*) and Sun4 (*Sun4*) versions of the simulator. Simply delete the version you don't need.

2.3.3 Restrictions Associated With the Sun Version of the Simulator

Several restrictions are associated with the Sun version of the debugger interface. These restrictions, listed below, override the information presented in Parts II and III of this manual, as well as the information presented in the tutorial.

- This version of the debugger works with monochrome monitors only. Descriptions of color monitors and color features do not apply.
- Breakpointed lines are not highlighted in any way.
- The C source debugger has a very flexible command-entry system; there are usually a variety of ways to perform any specific action. For example, you may be able to perform the same action by typing in a command, using the mouse, or using a function key. However, the function key and alternate-key sequences described in this manual apply to PCs. No keyboard mapping is provided for Sun systems. Therefore, you should look for methods that use the mouse or a command.
- The simulator is an instruction-level simulator that does not simulate the pipeline.

2.4 Invoking the Debugger

Here's the basic format for the command that invokes the debugger:



```

sim2x [filename] [-options]
```

- sim2x** is the command that invokes the debugger.
- filename* is an optional parameter that names an object file that the debugger will load into memory during invocation. The debugger looks for the file in the current directory; if the file isn't in the current directory, you must supply the entire path-name.
- options* supply the debugger with additional information (see Table 2-1).

With the PC version of the simulator or the PC version under Microsoft Windows, you can also specify filename and option information with the D_OPTIONS environment variable.

Table 2-1. Debugger Options

Option	Description																								
-b[bbbb]	<p>By default, the debugger uses an 80-character-by-25-line screen. If you are using the PC or Sun version of the simulator, you can use the -b screen-size options to choose a larger screen size. With the PC version, you must also have a special graphics card installed.</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;">Option</th> <th style="text-align: left;">Characters/Lines</th> <th style="text-align: left;">Notes</th> </tr> </thead> <tbody> <tr> <td><i>none</i></td> <td>80 by 25</td> <td><i>This is the default display</i></td> </tr> <tr> <td>-b</td> <td>80 by 39 (PC under Microsoft Windows)</td> <td></td> </tr> <tr> <td></td> <td>80 by 43 (PC with EGA; Sun)</td> <td><i>PC without Microsoft Windows:</i></td> </tr> <tr> <td></td> <td>80 by 50 (PC with VGA)</td> <td><i>Use any EGA or VGA card</i></td> </tr> <tr> <td>-bb</td> <td>120 by 43</td> <td rowspan="4" style="vertical-align: middle;">} <i>PC without Microsoft Windows only; currently, the debugger supports these modes on a Video Seven VEGA Deluxe card.</i></td> </tr> <tr> <td>-bbb</td> <td>132 by 43</td> </tr> <tr> <td>-bbbb</td> <td>80 by 60</td> </tr> <tr> <td>-bbbbb</td> <td>100 by 60</td> </tr> </tbody> </table>	Option	Characters/Lines	Notes	<i>none</i>	80 by 25	<i>This is the default display</i>	-b	80 by 39 (PC under Microsoft Windows)			80 by 43 (PC with EGA; Sun)	<i>PC without Microsoft Windows:</i>		80 by 50 (PC with VGA)	<i>Use any EGA or VGA card</i>	-bb	120 by 43	} <i>PC without Microsoft Windows only; currently, the debugger supports these modes on a Video Seven VEGA Deluxe card.</i>	-bbb	132 by 43	-bbbb	80 by 60	-bbbbb	100 by 60
Option	Characters/Lines	Notes																							
<i>none</i>	80 by 25	<i>This is the default display</i>																							
-b	80 by 39 (PC under Microsoft Windows)																								
	80 by 43 (PC with EGA; Sun)	<i>PC without Microsoft Windows:</i>																							
	80 by 50 (PC with VGA)	<i>Use any EGA or VGA card</i>																							
-bb	120 by 43	} <i>PC without Microsoft Windows only; currently, the debugger supports these modes on a Video Seven VEGA Deluxe card.</i>																							
-bbb	132 by 43																								
-bbbb	80 by 60																								
-bbbbb	100 by 60																								
-c	Sets memory reserved for uninitialized data to all zeros.																								
-i <i>pathname</i>	<p>-i identifies additional directories that contain your source files. Replace <i>pathname</i> with an appropriate directory name. You can specify several pathnames; use the -i option as many times as necessary:</p> <p>sim2x -i <i>path</i>₁ -i <i>path</i>₂ -i <i>path</i>₃ . . .</p> <p>Using -i is similar to using the D_SRC environment variable (described on page 2-5). If you name directories with both -i and D_SRC, the debugger first searches through directories named with -i.</p>																								

Table 2–1. Debugger Options (Continued)

Option	Description
<code>-mvversion</code>	The <code>-mv</code> options tells the simulator to simulate the operation of 'C25 or 'C26 DSPs: <code>-mv25</code> tells the simulator to simulate the 'C25 DSP operation (default). <code>-mv26</code> tells the simulator to simulate the 'C26 DSP operation. If you don't use the <code>-mv</code> option, the simulator simulates the 'C25 DSP operation.
<code>-mmode</code>	The <code>-mm</code> option tells the simulator to operate in either the microprocessor or micro-computer mode: <code>-mm0</code> tells the simulator to operate in the microcomputer mode. <code>-mm1</code> tells the simulator to operate in the microprocessor mode (default). If you don't use the <code>-mm</code> option, the simulator operates in the microprocessor mode.
<code>-s</code>	If you supply a <i>filename</i> when you invoke the debugger, you can use the <code>-s</code> option to tell the debugger to load only the file's symbol table (without the file's object code). This is similar to the debugger's SLOAD command.
<code>-t filename</code>	<code>-t</code> allows you to specify an initialization command file other than <code>siminit.cmd</code> . If <code>-t</code> is present on the command line, the file specified by <i>filename</i> will be invoked as the command file instead of <code>siminit.cmd</code> .
<code>-v</code>	This command 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 space. The <code>-v</code> option affects all loads, including those performed when you invoke the debugger and those performed with the LOAD command within the debugger environment.
<code>-x</code>	<code>-x</code> tells the debugger to ignore any information supplied with <code>D_OPTIONS</code> .

2.5 Exiting the Debugger

To exit the debugger and return to the operating system, enter this command:

```
quit 
```

You don't need to worry about where the cursor is or which window is active—just type. If a program is running, press `(ESC)` to halt program execution before you quit the debugger.

If you are using the PC version under Microsoft Windows, you can also exit the debugger by selecting the exit option from the Microsoft Windows menu bar.

An Introductory Tutorial to the C Source Debugger

This chapter provides a step-by-step, hands-on demonstration of the 'C2x C source debugger's basic features. This is not the kind of tutorial that you can take home to read—this tutorial is effective only if you're sitting at your PC, performing the lessons in the order that they're presented. This tutorial contains two sets of lessons (10 in the first, 13 in the second) and takes about one hour to complete.


Synopsis Page	Topic	Page
<i>Reading these sections will help you get the most out of the tutorial.</i>	How to use this tutorial	3-2
	A note about entering commands	3-3
	An escape route (just in case)	3-3
<i>The first set of lessons introduces you to basic debugger operation. You'll learn how to invoke the debugger and load object code, and you'll become acquainted with the main features of the debugger display. You'll also learn how to view a C source file and how to select one of the three debugging modes.</i>	Invoke the debugger and load the sample program's object code	3-4
	Take a look at the display...	3-5
	What's in the DISASSEMBLY window?	3-6
	Select the active window	3-6
	Resize the active window	3-8
	Move the active window	3-9
	Scroll through a window's contents	3-10
	Display the C source version of the sample file	3-11
	Execute some code	3-11
	Become familiar with the three debugging modes	3-12
<i>The second set of lessons shows you how to execute your programs and concentrates on the debugger's advanced features: setting breakpoints, benchmarking code, and observing the effects of program execution on selected variables, memory locations, and registers.</i>	Open another text file, then redisplay a C source file	3-14
	Use the basic RUN command	3-15
	Set some breakpoints	3-15
	Benchmark a section of code (simulator)	3-17
	Watch some values and single-step through code	3-18
	Run code conditionally	3-20
	WHATIS that?	3-21
	Clear the COMMAND window display area	3-22
	Display the contents of an aggregate data type	3-22
	Display data in another format	3-25
	Change some values	3-26
	Define a memory map	3-27
	Close the debugger	3-28

How to use this tutorial

This tutorial contains three basic types of information:

Primary actions

Primary actions identify the main lessons in the tutorial; they're boxed so you can find them easily. A primary action looks like this:

```
Make the CPU window the active window:  
win CPU 
```

Important information

In addition to primary actions, important information ensures that the tutorial works correctly. Important information is marked like this:

Important! The CPU window should still be active from the previous step.

Alternative actions

Alternative actions show additional methods for performing the primary actions. Alternative actions are marked like this:

Try This: Another way to display the current code in MEMORY is to show memory beginning from the current PC. . .

You can use this information in several ways:

- To use this information as a tutorial, perform the primary actions and pay close attention to the important information.
- If you want to be able to use the debugger like a real pro, then perform the alternative actions, too. (Don't worry—if you skip any of the alternative actions, the debugger won't blow up.)
- If all you're interested in is a quick demonstration of how the debugger works, just perform the primary actions.

Important! This tutorial assumes that you have correctly and completely installed your development board or emulator (including invoking any files or DOS commands as instructed in the installation chapters).

A note about entering commands

Whenever this tutorial tells you to type a debugger command, just type—the debugger automatically places the text on the command line. You don't have to worry about moving the cursor to the command line; the debugger takes care of this for you. (There are a few instances when this isn't true—for example, when you're editing data in the CPU or MEMORY window—but this is explained later in the tutorial.)

Also, you don't have to worry about typing commands in uppercase or lowercase—either is fine. There are a few instances when a command's *parameters* must be entered in uppercase, and the tutorial points this out.

An escape route (just in case)

The steps in this tutorial create a path for you to follow. The tutorial won't purposely lead you off the path. But sometimes when people use new products, they accidentally press the wrong key, push the wrong mouse button, or mistype a command. Suddenly, they're off the path without any idea of where they are or how they got there.

This probably won't happen to you. But, if it does, you can almost always get back to familiar ground by pressing `(ESC)`. If you were running a program when you pressed `(ESC)`, you should also type `RESTART` `(R)`. Then go back to the beginning of whatever lesson you were in and try again.

Invoke the debugger and load the sample program's object code

Included with the debugger is a demonstration program named *sample*. This lesson shows you how to invoke the debugger and load the sample program. You will use the `-b` option so that the debugger uses a larger display.

Important! This step assumes that you are using the default memory segment jumpers or that you have identified the memory segment jumpers with the `D_OPTIONS` environment variable (as described in the installation instructions in Chapter 1).

Invoke the debugger and load the sample program:

- For the **SWDS**, enter:

```
db2x -b c:\c2xh11\sample 
```

- For the **simulator**, enter:

```
sim2x -b c:\c2xh11\sample 
```

Take a look at the display. . .

Now you should see a display similar to this (it may not be exactly the same display, but it should be close).

The screenshot displays a debugger interface with several panels:

- Menu Bar:** Load, Brea, Watch, Memory, Color, MoDe, Run=F5, Step=F8, Next=F10
- DISASSEMBLY:** A list of assembly instructions with addresses and hex values. The current instruction is highlighted at address 1124: `d000 c_int0: LRLK AR0,#4e1h`.
- CPU:** A table of CPU registers and their values.

ACC	00000000
PREG	00000000
TIM	ffff
PRD	ffff
PC	1124
TOS	0000
ST0	0600
ST1	07f0
IMR	ffc0
IFR	0000
TREG	0000
RTPC	0000
AR0	0000
AR1	0000
AR2	0000
AR3	0000
AR4	0000
AR5	0000
AR6	0000
AR7	0000
BIO	0001
- COMMAND window display area:** Shows the debugger version (TMS320C2x Debugger Version 1.00) and copyright information.
- memory contents:** A table showing memory addresses and their corresponding hex values.

0000	0000	0000	ffff	ffff	ffc0	ff00	0000
0007	0000	0000	0000	0000	0000	0000	0000
strume	000e	0000	0000	0000	0000	0000	0000
TMS320C2x	0015	0000	0000	0000	0000	0000	0000
Simulator Version 3.0	001c	0000	0000	0000	0000	0000	0000
	0023	0000	0000	0000	0000	0000	0000
- command line:** Shows the prompt `>>>` with a cursor.

- If you **don't** see a display, then your debugger or board may not be installed properly. Go back through the installation instructions and be sure that you followed each step correctly; then reinvoke the debugger.
- If you **do** see a display, *check the first few lines of the DISASSEMBLY window*. If these lines aren't the same—if, for example, they show ADD instructions or say Invalid address—then enter the following commands on the debugger command line. (Just type; you don't have to worry about where the cursor is.)

1) Reset the 'C2x processor:

```
reset 
```

2) Load the sample program again:

```
load c:\c2xh11\sample 
```

What's in the DISASSEMBLY window?

The DISASSEMBLY window always shows the reverse assembly of memory contents; in this case, it shows an assembly language version of sample.out. The MEMORY window displays the current contents of memory. Because you loaded the object file sample.out when you invoked the debugger, memory contains the object code version of the sample file.

This tutorial step demonstrates that the code shown in the DISASSEMBLY window corresponds to memory contents. Initially, memory is displayed starting at address 0; if you look at the first line of the DISASSEMBLY window, you'll see that its display starts at address 0x1124.

Modify the MEMORY display to show the same object code that is displayed in the DISASSEMBLY window:

```
mem 0x1124@prog 
```

Notice that the first column in the DISASSEMBLY window corresponds to the addresses in the MEMORY window; the second column in the DISASSEMBLY window corresponds to the memory contents displayed in the MEMORY window.

Try This: The 'C2x has separate program and data spaces. You can access either program or data memory by following the location with **@prog** for program memory or **@data** for data memory. If you'd like to see the contents of location 0x1124 in data memory, enter:

```
mem 0x1124@data 
```

Try This: Another way to display the current code in MEMORY is to show memory beginning from the current PC:

```
mem PC@prog 
```

Select the active window

This lesson shows you how to make a window into the *active window*. You can move and resize any window; you can close some windows. Whenever you type a command or press a function key to move, resize, or close a window, the debugger must have some method of understanding which window you want to affect. The debugger does this by designating one window at a time to be the *active window*. Any window can be the active window, but only one window at a time can be active.



Make the CPU window the active window:

`win CPU` 

Important! If this didn't work, look at the way you entered the command. Did you enter **CPU** in uppercase letters? For this command, it's important that you enter the parameter in uppercase as shown.

Important! Notice the appearance of the CPU window (especially its borders) in contrast to the other, inactive windows! This is how you can tell which window is active.



Try This: Press the `F6` key to "hop" through the windows in the display, making each one active in turn. Press `F6` as many times as necessary until the CPU window becomes the active window.



Try This: You can also use the mouse to make a window active:



1) Point to any location on the window's border.



2) Click the left mouse button.

Be careful! If you point *inside* the window, the window becomes active when you press the mouse button, but something else may happen as well:

- If you're pointing inside the CPU window, then the register you're pointing at becomes active. The debugger then treats the text you type as a new value for that register. Similarly, if you're pointing inside the MEMORY window, the address you're pointing at becomes active.

Press `ESC` to get out of this.

- If you're pointing inside the DISASSEMBLY or FILE window, you'll set a breakpoint on the statement that you were pointing to.

Point to the same statement; press the button again to delete the breakpoint.

Resize the active window

This lesson shows you how to resize the active window.

Important! The CPU window should still be active from the previous step.




Make the CPU window as small as possible:

`size 4,3` 

This tells the debugger to make the window 4 characters by 3 lines, which is the smallest a window can be. (If it were any smaller, the debugger wouldn't be able to display all four corners of the window.) If you try to enter smaller values, the debugger will warn you that you've entered an *Invalid window size*. The maximum width and length depend on which `-b` option you used when you invoked the debugger. (If you'd like a complete list of the limits, see Table 5-1 on page 5-21.)



Make the CPU window larger:

`size` 

Enter the SIZE command without parameters



Make the window 3 lines longer

Make the window 4 characters wider




Press this key when you finish sizing the window

You can also use  to make the window shorter and  to make the window narrower.



Try This: You can also use the mouse to resize the window (note that this process forces the selected window to become the active window).

-  1) If you examine any window, you'll see a highlighted, backwards "L" in the lower right corner. Point to the lower right corner of the CPU window.

- ☰ 2) Press the left mouse button, but don't release it; move the mouse while you're holding in the button. This resizes the window.
- ☐ 3) Release the mouse button when the window reaches the desired size.

Move the active window

This lesson shows you how to move the active window.

Important! The CPU window should still be active from the previous steps.



Move the CPU window to the upper left portion of the screen:

`move 0,1`

The debugger doesn't let you move the window to the very top—that would hide the menu bar

The MOVE command's first parameter identifies the window's new X position on the screen. The second parameter identifies the window's new Y position on the screen. The maximum X and Y positions depend on which `-b` option you used when you invoked the debugger. (For a complete list of the limits, see Table 5-2 on page 5-23.)



Try This: You can use the MOVE command with no parameters and then use arrow keys to move the window:

`move`

Press until the CPU window is back where it was (it may seem like only the border is moving—this is normal)

`ESC`

Press `ESC` when you finish moving the window

You can also use to move the window up, to move the window down, and to move the window left.



Try This: You can also use the mouse to move the window (note that this process forces the selected window to become the active window).

- 1) Point to the top edge or left edge of the window border.
- ☰ 2) Press the left mouse button, but don't release the button; move the mouse while you're holding in the button.

- 3) Release the mouse button when the window reaches the desired position.

Scroll through a window's contents

Many of the windows contain more information than can possibly be displayed at one time. You can view hidden information by moving through a window's contents. The easiest way to do this is to use the mouse to scroll the display up or down.



If you examine most windows, you'll see an up arrow near the top of the right border and a down arrow near the bottom of the right border. These are scroll arrows.

Scroll through the contents of the DISASSEMBLY window:

- 1) Point to the up or down scroll arrow.
- 2) Press the left mouse button; continue pressing it until the display has scrolled several lines.
- 3) Release the button.



Try This: You can also use several of the keys to modify the display in the active window.

Make the MEMORY window the active window:

`win MEMORY` 

Now try pressing these keys; observe their effects on the window's contents.



These keys don't work the same for all windows; Section 12.3 (page 12-37) summarizes the functions of all the special keys, key sequences, and how their effects vary for the different windows.

Display the C source version of the sample file

Now that you can find your way around the debugger interface, you can become familiar with some of the debugger's more significant features. It's time to load some C code.

Display the contents of a C source file:

```
file sample.c 
```

This opens a FILE window that displays the contents of the file `sample.c` (`sample.c` was one of the files that contributed to making the sample object file). You can always tell which file you're displaying by the label in the FILE window. Right now, the label should say `FILE: sample.c`.

Execute some code

Let's run some code—not the whole program, just a portion of it.

Execute a portion of the sample program:

```
go main 
```

You've just executed your program up to the point where `main()` is declared. Notice how the display has changed:

- The current PC is highlighted in both the DISASSEMBLY and FILE windows.
- The addresses and object code of the first several statements in the DISASSEMBLY window are highlighted; this is because these statements are associated with the current C statement (highlighted in the FILE window).
- The CALLS window, which tracks functions as they're called, now points to `main()`.
- The values of the PC and SP (and possibly some additional registers) are highlighted in the CPU window because they were changed by program execution.

Become familiar with the three debugging modes

The debugger has three basic debugging modes:




- Mixed mode** shows both disassembly and C at the same time.
- Auto mode** shows disassembly or C, depending on what part of your program happens to be running.
- Assembly mode** shows only the disassembly, no C, even if you're executing C code.

When you opened the FILE window in a previous step, the debugger switched to mixed mode; you should be in mixed mode now. (You can tell that you're in mixed mode if both the FILE and DISASSEMBLY windows are displayed.)

The following steps show you how to switch debugging modes.





Use the **MoDe** menu to select assembly mode:

- 1) Look at the top of the display: the first line shows a row of pull-down menu selections.
-  2) Point to the word MoDe on the menu bar.
-  3) Press the left mouse button, but don't release it; drag the mouse downward until Asm (the second entry) is highlighted.
-  4) Release the button.

This switches to assembly mode. You should see the DISASSEMBLY window, but not the FILE window.

Switch to auto mode:

- 1) Press **ALT** **D**. This displays and freezes the MoDe menu.
- 2) Now select C(auto). Choose one of these methods for doing this:
 - Press the arrow keys to move up/down through the menu; when C(auto) is highlighted, press .
 - Type **c** .
 - Point the mouse cursor at C(auto), then click the left mouse button.

You should be in auto mode now, and you should see the FILE window but not the DISASSEMBLY window (because your program is in C code). Auto mode automatically switches between an assembly or a C display, depending on where you are in your program. Here's a demonstration of that:

Run to a point in your program that executes assembly language code:

```
go meminit 
```

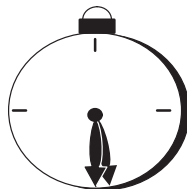
You're still in auto mode, but you should now see the DISASSEMBLY window. The current PC should be at the statement that defines the meminit label.



Try This: You can also switch modes by typing one of these commands:


asm switches to assembly-only mode
c switches to auto mode
mix switches to mixed mode

Switch back to mixed mode.



Halfway Point

You've finished the first half of the tutorial and the first set of lessons.

If you're lucky enough to be going to lunch or going home at this point, you may want to close the debugger down. To do this, just type **QUIT** . When you come back, reinvoke the debugger and load the sample program (page 3-4). Then turn to page 3-14 and continue with the second set of lessons.

Still here? Turn the page.

Open another text file, then redisplay a C source file

In addition to what you already know about the FILE window and the FILE command, you should also know that:

- You can display any text file in the FILE window.
- If you enter any command that requires the debugger to display a C source file, it automatically displays that code in the FILE window (regardless of whether the window is open or not and regardless of what is already displayed in the FILE window).

Display a file that isn't a C source file:

```
file ..\autoexec.bat 
```

This replaces sample.c in the FILE window with your autoexec.

Remember, you can tell which file you're displaying by the label in the FILE window. Right now, the label should say `FILE: autoexec.bat`.

Redisplay another C source file (sample1.c):


```
func call 
```

Now the FILE window label should say `FILE: sample1.c` because the `call()` function is in `sample1.c`.

Use the basic run command

The debugger provides you with several ways of running code, but it has one basic run command.

Run your entire program:

`run` 

Entered this way, the command basically means “run forever”. You may not have that much time!

This isn't very exciting: halt program execution:

`ESC`

Set some breakpoints

When you halted execution in the previous step, you should have seen changes in the display similar to the changes you saw when you entered `go main` earlier in the tutorial. When you pressed `ESC`, you had little control over where the program stopped. Knowing that information changed was nice, but what part of the program affected the information?

This information would be much more useful if you picked an explicit stopping point before running the program. Then, when the information changed, you'd have a better understanding of what caused the changes. You can stop program execution in this way by setting *breakpoints*.

Important!



This lesson assumes that you're displaying the contents of `sample.c` in the FILE window. If you aren't, enter:

`file sample.c` 

lesson continues on the next page →

Set a breakpoint and run your program:


1) Scroll to line 51 in the FILE window (the meminit() statement) and set a breakpoint at that line:

-  a) Point the mouse cursor at the statement on line 51.
-  b) Click the left mouse button. *Notice how the line is highlighted; this identifies a breakpointed statement.*

2) Reset the program entry point:


`restart` 


3) Enter the run command:

`run`  *Program execution halts at the breakpoint*

Once again, you should see that some statements are highlighted in the CPU window, showing that they were changed by program execution. But this time, you know that the changes were caused by code from the beginning of the program to line 51 in the FILE window.

Clear the breakpoint:

 1) Point the mouse cursor at the statement on line 51. (It should still be highlighted from setting the breakpoint.)

 2) Click the left mouse button. *The line is no longer highlighted.*

Benchmark a section of code (*simulator*)

If you're using the 'C2x simulator, you can use breakpoints to help you benchmark a section of code. This means that you'll ask the debugger to count the number of CPU clock cycles that are consumed by a certain portion of code.


Benchmark some code:

1) In sample.c (displayed in the FILE window), set two breakpoints: one at line 56 (the call (i) statement) and one at line 59 (the if (!(i&0xFFFA)) statement).

2) Reset the program entry point:

`restart` 

3) Enter the run command:

`run` 

This runs to the first breakpoint

4) Enter the runb command:

`runb` 

This runs to the second breakpoint

5) Now use the ? command to examine the contents of the CLK pseudo-register:

`? clk` 

simulator

The debugger now shows a number in the display area; this is the number of CPU clock cycles consumed by the portion of code between the two breakpointed C statements.

Important!

The value in the CLK pseudoregister is valid *only* when you execute the RUNB command and when that execution is halted on breakpointed statements.

Delete both breakpoints:

`br` 



The BR (breakpoint reset) command deletes all breakpoints that were set

Watch some values and single-step through code

Now you know how to update the display without running your entire program; you can set breakpoints to obtain information at specific points in your program. But what if you want to update the display after each statement? No, you don't have to set a breakpoint at every statement—you can use single-step execution.

For this lesson, you have to be at a specific point in the program—let's go there before we do anything else.




Set up for single-step example:

```
restart   
go main 
```

The debugger has another type of window called a WATCH window that's very useful in combination with single-step execution. What's a WATCH window for? Suppose you are interested in only a few specific register values, not *all* of the registers shown in the CPU window. Or suppose you are interested in a particular memory location or in the value of some variable. You can observe these data items in a WATCH window.

Set up the WATCH window before you start the single-step execution.

Open a WATCH window:

```
wa TOS, Stack Pointer   
wa pc   
wa i 
```

You may have noticed that the WA (watch add) command can have one or two parameters. The first parameter is the item that you're watching. The second parameter is an optional label.


Now try out the single-step commands. **Hint:** Watch the PC in the FILE and DISASSEMBLY windows; watch the value of `i` in the WATCH window.

Single-step through the sample program:

```
step 50 
```

Try This: Notice that the step command single-stepped each assembly language statement (in fact, you single-stepped through 50 assembly language statements). Did you also notice that the FILE window displayed the source for the `call()` function when it was called? The debugger supports more single-step commands that have a slightly different flavor.

For example, if you enter:

```
cstep 50 
```

you'll single-step 50 *C statements*, not assembly language statements (notice how the PC "jumps" in the DISASSEMBLY window).

Reset the program entry point and run to `main()`.

```
restart 
```

```
go main 
```

Now enter the NEXT command, as shown below. You'll be single-stepping 50 assembly language statements, *but the FILE window doesn't display the source for the `call()` function when `call()` is executed.*

```
next 50 
```

(There's also a CNEXT command that "nexts" in terms of C statements.)


Run code conditionally

Let's execute this loop one more time. Take a look at this code; its doing a lot of work with a variable named `i`. You may want to check the value of `i` at specific points instead of after each statement. To do this, you set breakpoints at the statements you're interested in and then initiate a conditional run.

First, clear out the WATCH window so that you won't be distracted by any superfluous data items.

Delete the first three data items from the WATCH window (don't watch them anymore)

`wd 3` 

`wd 1` 

`i` was the third item added to the WATCH window in the previous tutorial step, and it should now be the only remaining item in the window.

Set up for the conditional run examples

1) Set breakpoints at lines 56 and 58.


2) Reset the program entry point:

`restart` 

3) Run the first part of the program

`go main` 

4) Reset the value of `i`:

`?i=0` 


Now initiate the conditional run:

`run i<100` 

This causes the debugger to run through the loop as long as the value of `i` is less than 100. Each time the debugger encounters the breakpoints in the loop, it updates the value of `i` in the WATCH window.

When the conditional run completes, close the WATCH window.

Close the WATCH window:







wr 

WHATIS that?

At some point, you might like to obtain some information about the types of data in your C program. Maybe things won't be working quite the way you'd planned, and you'll find yourself saying something like "... but isn't that supposed to point to an integer?" Here's how you can check on this kind of information: be sure to watch the COMMAND window display area as you enter these commands.

Use the WHATIS command to find the types of some of the variables declared in the sample program:

```

whatis genum 
    enum yy genum;                                genum is an enumerated type
whatis tiny6 
    struct {                                       tiny6 is a structure
        int u;
        int v;
        int x;
        int y;
        int z;
    } tiny6;
whatis call 
    int call();                                    call is a function that returns an integer
whatis s 
    short s;                                       s is a short unsigned integer
whatis zzz 
    struct zzz {                                   zzz is a very long structure
        int b1;
        int b2;
    }
    Press  to halt long listings
  
```

Clear the COMMAND window display area

After displaying all of these types, you may want to clear them away. This is easy to do.

Clear the COMMAND window display area:

```
cls
```

Try This:

CLS isn't the only system-type command that the debugger supports.

```
cd ..  
dir  
cd c2xh11
```

Change back to the main directory
Show a listing of the current directory
Change back to the debugger directory

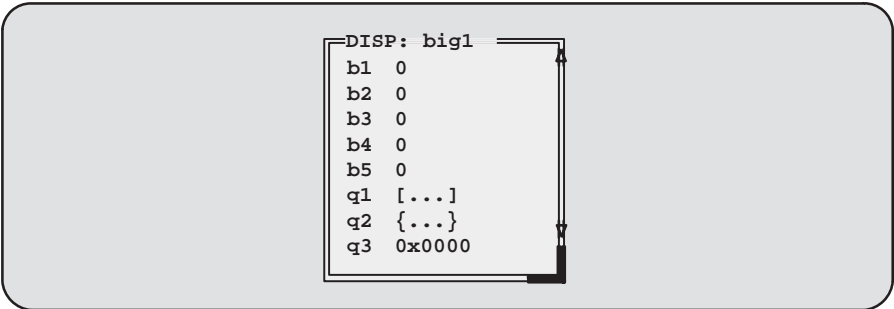
Display the contents of an aggregate data type

The WATCH window is convenient for watching single, or *scalar*, values. When you're debugging a C program, though, you may need to observe values that aren't scalar; for example, you might need to observe the effects of program execution on an array. The debugger provides another type of window called a DISP window where you can display the individual members of an array or structure.

Show another structure in a DISP window:

```
disp big1
```

Now you should see a display like the one below. The newly opened DISP window becomes the active window. Like the FILE window, you can always tell what's being displayed because of the way the DISP window is labeled. Right now, it should say `DISP: big1`.





```
DISP: big1  
b1 0  
b2 0  
b3 0  
b4 0  
b5 0  
q1 [...]  
q2 {...}  
q3 0x0000
```

- Members b1, b2, b3, b4, and b5 are ints; you can tell because they're displayed as integers (shown as plain numbers without prefixes).
- Member q1 is an array; you can tell because q1 shows [. . .] instead of a value.
- Member q2 is another structure; you can tell because q2 shows { . . . } instead of a value.
- Member q3 is a pointer; you can tell because it is displayed as a hexadecimal address (indicated by a 0x prefix) instead of an integer value.

If a member of a structure or an array is itself a structure or an array, or even a pointer, you can display its members (or the data it points to) in additional DISP windows (referred to as the original DISP window's *children*).



Display what q3 is pointing to:

-  1) Point at the address displayed next to the q3 label in big1's display.
-  2) Click the left mouse button.

This opens a second DISP window, named big1.q3, that shows what q3 is pointing to (it's pointing to another structure). Close this DISP window or move it out of the way.



Display array q1 in another DISP window:

-  1) Point at the [. . .] displayed next to the q1 label in big1's display.
-  2) Click the left mouse button.

This opens another DISP window labeled DISP: big1.q1.

Important! q1 is actually a 2-member array of structures. To view the two different structures, use **(CONTROL) (PAGE DOWN)** and **(CONTROL) (PAGE UP)**. (Look at the name of this DISP window when you're switching.)

lesson continues on the next page →



Try This: Display structure q2 in another DISP window.

- 1) Close the additional DISP windows or move them out of the way so that you can clearly see the original DISP window that you opened to display big1.
- 2) Make big1's DISP window the active window.
- ⏴ ⏵ 3) Use these arrow keys to move the field cursor (`_`) through the list of big1's members until the cursor points to q2.
- ⏴ 4) Now press ⏴.

Close all of the DISP windows:

- 1) Make big1's DISP window the active window.
- 2) Press ⏴.

When you close the main DISP window, the debugger closes all of its children as well.

Display data in another format

Usually when you add an item to the WATCH window or open a DISP window, the data is shown in its *natural format*. This means that ints are shown as integers, floats are shown as floating-point values, etc. Occasionally, however, you may wish to view data in a different format.

This is especially important if you want to show memory contents in another format. For example, suppose you want to see the contents of the stack in integer format:

Display memory contents in an integer format:

```
disp *(int *)TOS 
```

Notice that this shows memory contents in the DISP window in an array format. The “array” member identifiers don’t necessarily correspond to actual addresses—they’re relative to the first address you request with the DISP command. In this case, the item displayed as item [0] is the contents of the stack pointer— *it isn’t memory location 0*. Note that you can scroll through the memory displayed in the DISP window; item [1] is at &TOS + 1, item [-1] is at &TOS-1.

Try This: You might also want to display memory contents in floating-point format. For example, you can display the contents of location 0x0 in floating-point format:

```
disp *(float *)0x0 
```

To get ready for the next step, close any DISP windows that are open.

Change some values

You can edit the values displayed in the MEMORY, CPU, WATCH, and DISP windows.



Change a value in memory:

- 1) Move or close the WATCH window if it's obscuring the MEMORY window, then display memory beginning with address 0x042b:

mem 0x042b

- 2) Point to the contents of memory location 0x042b.
- 3) Click the left mouse button. This highlights the field to identify it as the field that will be edited.
- 4) Type 1111.
- 5) Press to enter the new value.
- 6) Press to conclude editing.



Try This: Here's another method for editing data that lets you edit a few more values at once.

- 1) Make the CPU window the active window:
win CPU
- 2) Press the arrow keys until the field cursor (`_`) points to the PC contents.
- 3) Press .
- 4) Type 0107.
- 5) Press 8 times. You should now be pointing at the contents of register AR0.
- 6) Type ffff.
- 7) Press to enter the new value.
- 8) Press to conclude editing.

Define a memory map

You can set up a memory map to tell the debugger which areas of memory it can and can't access. This is called *memory mapping*. When you invoked the debugger for this tutorial, the debugger automatically read a default memory map from a batch file included in the c2xhll directory. For the purposes of the sample program, that's fine (which is why this lesson was saved for next-to-last).

View the default memory map settings:

```
m1 
```

Look in the COMMAND window display area—you'll see a listing of the areas that are currently mapped. The 'C2x supports separate program and data spaces. Page 0 in the memory map is for program memory; page 1 is for data memory.

It's easy to add new ranges to the map or delete existing ranges.

Change the memory map:


- 1) Use the MD (memory delete) command to delete the block of data memory:

```
md 0x0,1 
```

This deletes the block of memory beginning at address 0 in data memory.

- 2) Use the MA (memory add) command to define a new block of program memory and a new block of data memory:


```
ma 0x2000,0,0x20,ROM 
```

```
ma 0x4000,1,0xffff,RAM 
```

Close the debugger

This is the end of the tutorial—close the debugger.

Close the debugger and return to DOS:

`quit` 

Overview of a Code Development and Debugging System

The 'C2x C source debugger is an advanced software interface that helps you to develop, test, and refine 'C2x C programs (compiled with the 'C2x optimizing ANSI C compiler) and assembly language programs. The debugger is the interface to TI's 'C2x SWDS and simulator. This chapter provides an overview of the C source debugger and describes the 'C2x code development environment.

Topic	Page	
<i>The chapter provides an overview of the debugger and the debugging process and describes how the debugging process fits in with the overall code development process.</i>	4.1 Description of the 'C2x Debugger	4-2
	Key features of the debugger	4-3
	4.2 Developing Code for the 'C2x	4-5
	4.3 Preparing Your Program for Debugging	4-8
	4.4 Debugging 'C2x Programs	4-10

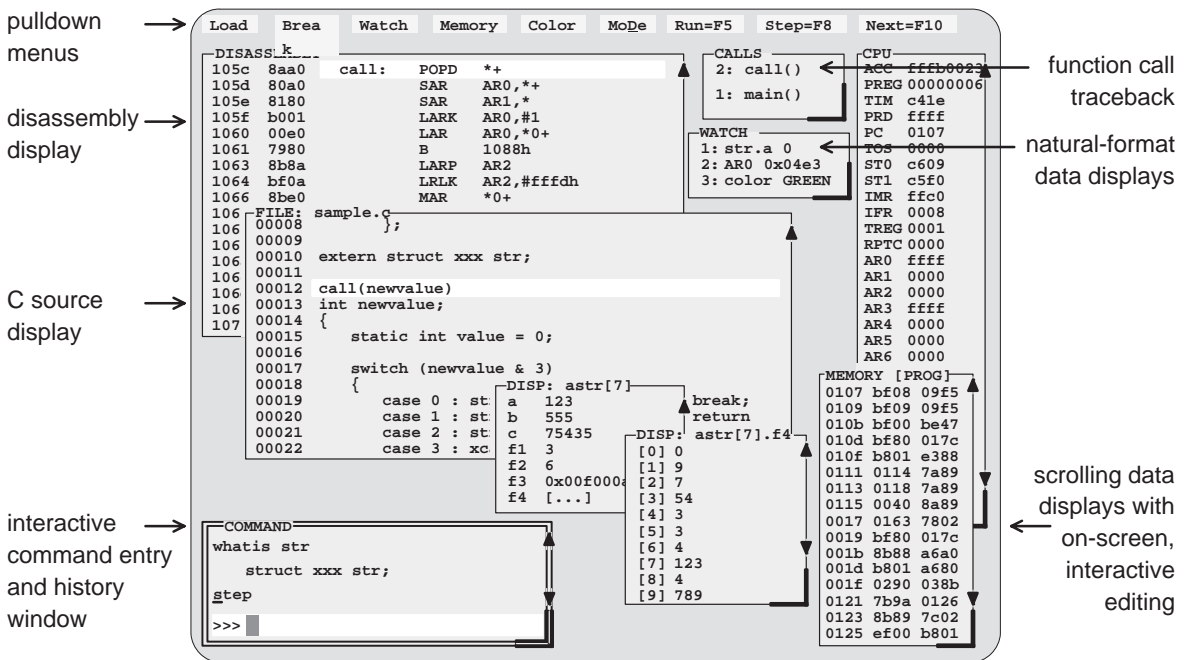
4.1 Description of the 'C2x C Source Debugger

The 'C2x C source debugger 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, or both. And, unlike many other debuggers, the 'C2x debugger's higher level features are available even when you're debugging assembly language code.

The debugger is easy to learn and use. Its friendly window-, mouse-, and menu-oriented interface reduces learning time and eliminates the need to memorize complex commands. The debugger's customizable displays and flexible command entry let you develop a debugging environment that suits your needs—you won't be locked into a rigid environment. A shortened learning curve and increased productivity reduce the software development cycle, so you'll get to market faster.

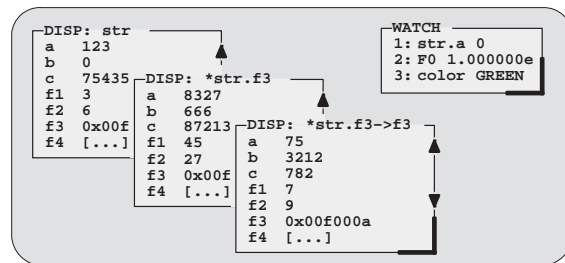
Figure 4–1 identifies several features of the debugger display.

Figure 4–1. The Debugger Display



Key features of the debugger

- Multilevel debugging.** The debugger allows you to debug both C and assembly language code. If you're debugging a C program, you can choose to view just 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.
- Fully configurable, state-of-the-art, window-oriented interface.** The C source debugger separates code, data, and commands into manageable portions. Use any of the default displays. Or, select the windows you want to display, size them, and move them where you want them.
- 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.** Change any data value displayed in any window—just point the mouse, click, and type.
- Continuous update.** The debugger continuously updates information on the screen, highlighting changed values.
- Powerful command set.** Unlike many other debugging systems, this debugger doesn't force you to learn a large, intricate command set. The 'C2x C source debugger supports a small but powerful command set that makes full use of C expressions. One debugger command performs actions that would take several commands in another system.

- **Flexible command entry.** There are a variety of ways to enter commands. You can type commands or use a mouse, function keys, or the pulldown menus; choose the method that you like best. Want to re-enter a command? No need to retype it—simply use the command history.



- **Create your own debugger.** The debugger display is completely configurable, allowing you to create the interface that is best suited for your use.
 - If you're using a color display, you can change the colors of any area on the screen.
 - You can change the physical appearance of display features such as window borders.
 - You can interactively set the size and position of windows in the display.

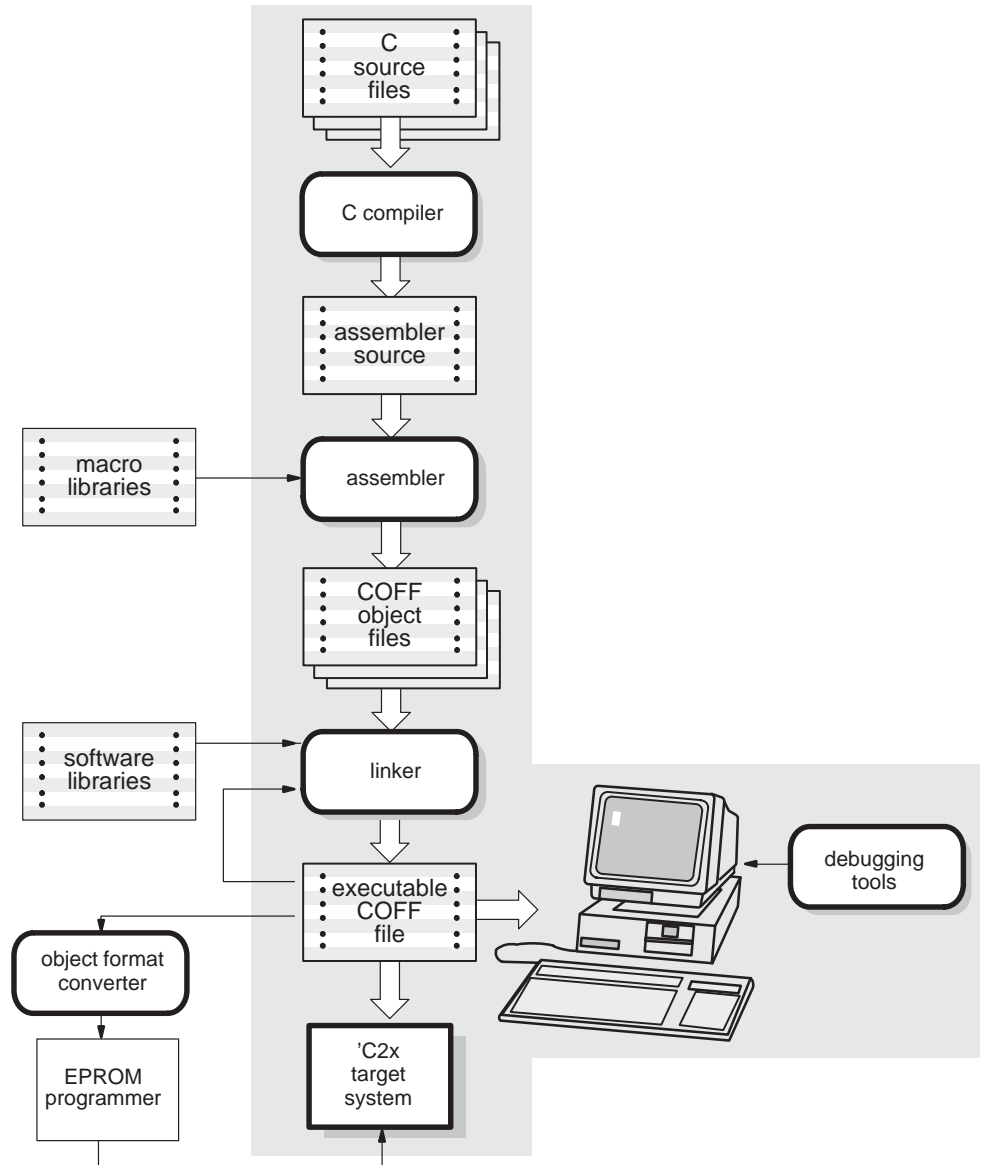
Create and save as many custom configurations as you like, or use the defaults. Use the debugger with a color display or a black-and-white display. A color display is preferable; the various types of information on the display are easier to distinguish when they are highlighted with color.

- **Variety of screen sizes.** The debugger's default configuration is set up for a typical PC display, with 25 lines by 80 characters. If you use a sophisticated graphics card, you can take advantage of the debugger's additional screen sizes. A larger screen size allows you to display more information and provides you with more screen space for organizing the display—bringing the benefits of workstation displays to your PC.
- **Plus 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 or by typing commands. You can define a memory map that identifies the portions of target memory that the debugger can access. You can choose to load only the symbol table portion of an object file to work with systems that have code in ROM. The debugger can execute commands from a batch file, providing you with an easy method for entering often-used command sequences.

4.2 Developing Code for the 'C2x

The 'C2x is supported by a complete set of hardware and software development tools, including a C compiler, assembler, and linker. Figure 4–2 illustrates the 'C2x code development flow. The figure highlights the most common paths of software development; the other portions are optional.

Figure 4–2. 'C2x Software Development Flow



These tools use common object file format (COFF), which encourages modular programming. 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 4–2.

C compiler

The 'C2x **optimizing ANSI C compiler** is a full-featured optimizing compiler that translates standard ANSI C programs into 'C2x assembly language source. Key characteristics include:

- *Standard ANSI C.* The ANSI standard is a precise definition of the C language, agreed upon by the C community. The standard encompasses most of the recent extensions to C. To an increasing degree, ANSI conformance is a requirement for C compilers in the DSP community.
- *Optimization.* The compiler uses several advanced techniques for generating efficient, compact code from C source.
- *Assembly language output.* The compiler generates assembly language source that you can inspect (and modify, if desired).
- *ANSI standard runtime support.* The compiler package comes with a complete runtime library that conforms to the ANSI C library standard. The library includes functions for string manipulation, dynamic memory allocation, data conversion, timekeeping, trigonometry, exponential, and hyperbolic operations. Functions for I/O and signal handling are not included because they are application specific.
- *Flexible assembly language interface.* The compiler has straightforward calling conventions, allowing you to easily write assembly and C functions that call each other.
- *Shell program.* The compiler package includes a shell program that enables you to compile, assemble, and link programs in a single step.
- *Source interlist utility.* The compiler package includes a utility that interlists your original C source statements into the assembly language output of the compiler. This utility provides you with an easy method for inspecting the assembly code generated for each C statement.

assembler

The **assembler** translates 'C2x assembly language source files into machine language object files.

archiver

The **archiver** allows you to collect a group of files into a library. It also allows you to modify a library by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is to build a library of object modules. Several object libraries and a source library are included with the C compiler.

linker

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

debugging
tools

The main purpose of the development process is to produce a module that can be executed in a '**C2x target system**. You can use one of several **debugging tools** to refine and correct your code. Available products include:

- A software development system (**SWDS**), and
- A software **simulator**.

Each of these tools uses the 'C2x debugger as a software interface.

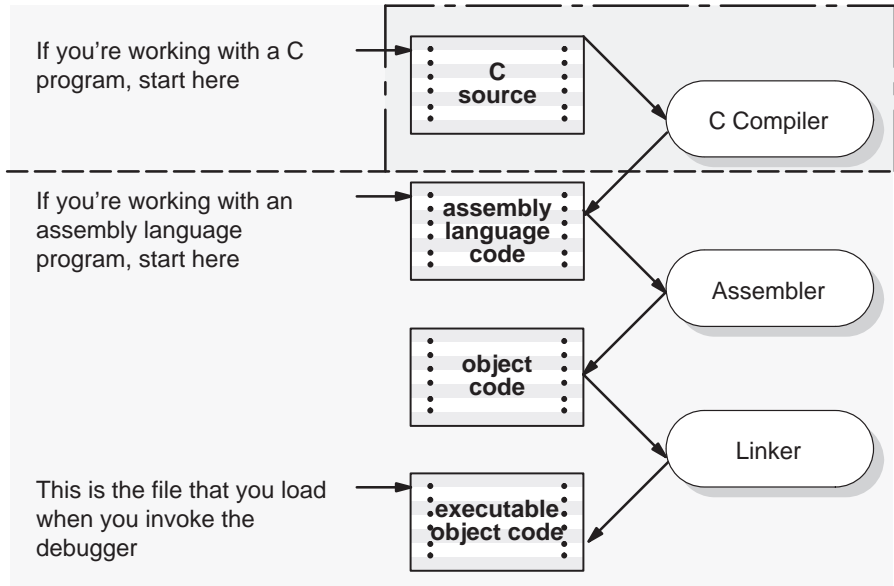
object
format
converter

An **object format converter** is also available; it converts a COFF object file into an Intel, Tektronix, or TI-tagged object-format file that can be downloaded to an EPROM programmer.

4.3 Preparing Your Program for Debugging

Figure 4–3 illustrates the steps you must go through to prepare a program for debugging.

Figure 4–3. Steps You Go Through to Prepare a Program



If you're preparing to debug a C program. . .

- 1) Compile the program; **use the `-g` option.**
- 2) Assemble the resulting assembly language program.
- 3) Link the resulting object file.

This produces an object file that you can load into the debugger.

If you're preparing to debug an assembly language program. . .

- 1) Assemble the assembly language source file.
- 2) Link the resulting object file.

This produces an object file that you can load into the debugger.

You can compile, assemble, and link a program by invoking the compiler, assembler, and linker in separate steps; or, you can perform all three actions in a single step by using the CL2x shell program. The *TMS320 Floating-Point DSP Assembly Language Tools User's Guide* and *TMS320C2x C Compiler Reference Guide* contain complete instructions for invoking the tools individually and for using the shell program.

For your convenience, here's the command for invoking the shell program when preparing a program for debugging:

```
dspcl [-options] -g [filenames] [-z [link options]]
```

- dspcl** is the command that invokes the compiler and assembler.
- options* affect the way the shell processes input files.
- filenames* are one or more C source files, assembly language source files, or object files. Filenames are not case sensitive.
- g** is an option that tells the C compiler to produce symbolic debugging information. When preparing a C program for debugging, you must use the **-g** option.
- z** is an option that invokes the linker. After compiling/assembling your programs, you can invoke the linker in a separate step. If you want the shell to automatically invoke the linker, however, use **-z**.
- link options* affect the way the linker processes input files; use these options only when you use **-z**.

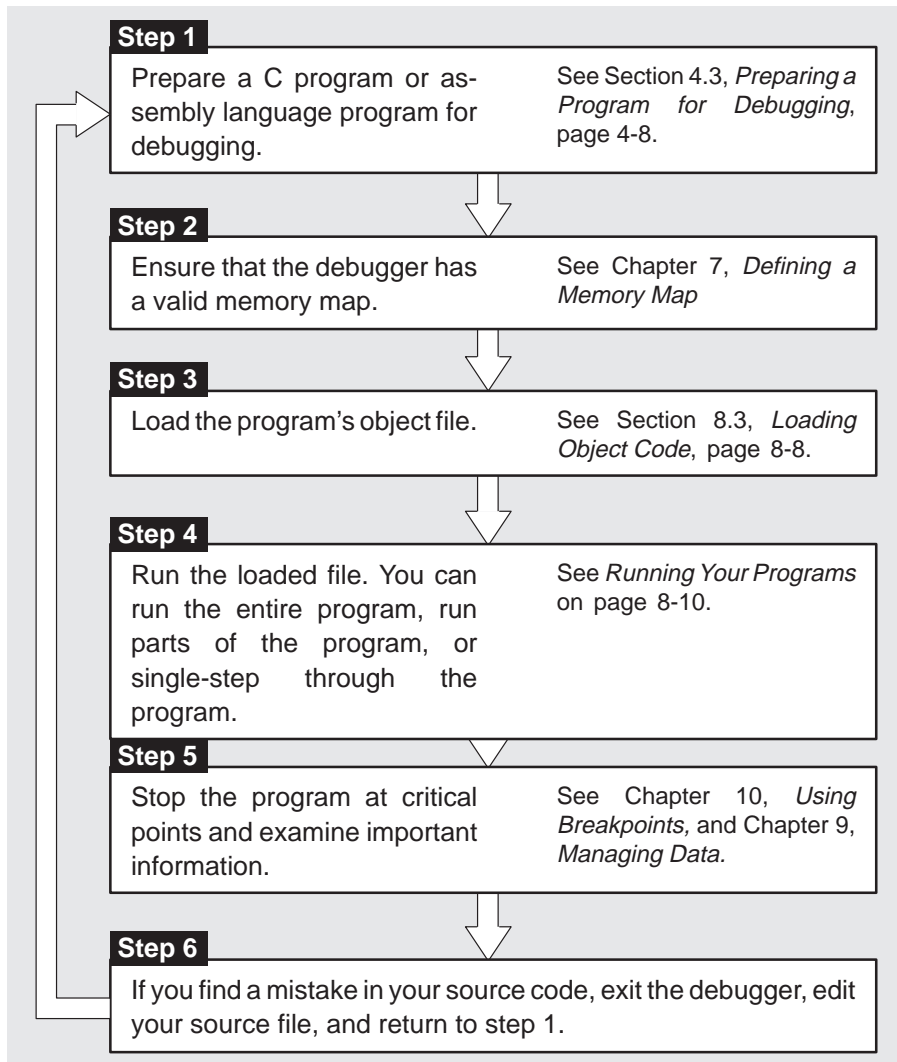
Options and filenames can be specified in any order on the command line, but if you use **-z**, it must follow all C/assembly language source filenames and compiler options.

The shell identifies a file's type by the filename's extension.

Extension	File Type	File Description
.c	C source	compiled, assembled, and linked
.asm	assembly language source	assembled and linked
.s* (any extension that begins with s)	assembly language source	assembled and linked
.o* (extension begins with o)	object file	linked
none (.c assumed)	C source	compiled, assembled, and linked

4.4 Debugging 'C2x Programs

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



The Debugger Display

The 'C2x C source debugger has a window-oriented display. This chapter shows what windows can look like and describes the basic types of windows that you'll use.

Topic	Page
<i>The debugger's three modes use a set of three default displays. These modes control the types of information that you can display and the types of actions that you can perform.</i>	5.1 Debugging Modes and Default Displays 5-2
	Auto mode 5-2
	Assembly mode 5-3
	Mixed mode 5-4
	Restrictions associated with debugging modes 5-4
<i>The debugger can display eight different types of windows. Each has a unique purpose.</i>	5.2 Descriptions of the Different Kinds of Windows and Their Contents 5-5
	COMMAND window 5-6
	DISASSEMBLY window 5-6
	FILE window 5-8
	CALLS window 5-9
	MEMORY window 5-11
	CPU window 5-13
	DISP windows 5-14
	WATCH window 5-15
<i>The windows in the debugger display aren't fixed in position or size. You can resize, move, and, in some cases, close windows. The window that you're going to move, resize, or close must be the active window.</i>	5.3 Cursors 5-16
	5.4 The Active Window 5-17
	Identifying the active window 5-17
	Selecting the active window 5-18
	5.5 Manipulating Windows 5-20
	Resizing a window 5-20
	Moving a window 5-22
	5.6 Manipulating a Window's Contents 5-25
	Scrolling through a window's contents 5-25
	Editing the data displayed in windows 5-26
5.7 Closing a Window 5-28	

5.1 Debugging Modes and Default Displays

The debugger has three debugging modes:

- Auto mode
- Assembly mode
- Mixed mode

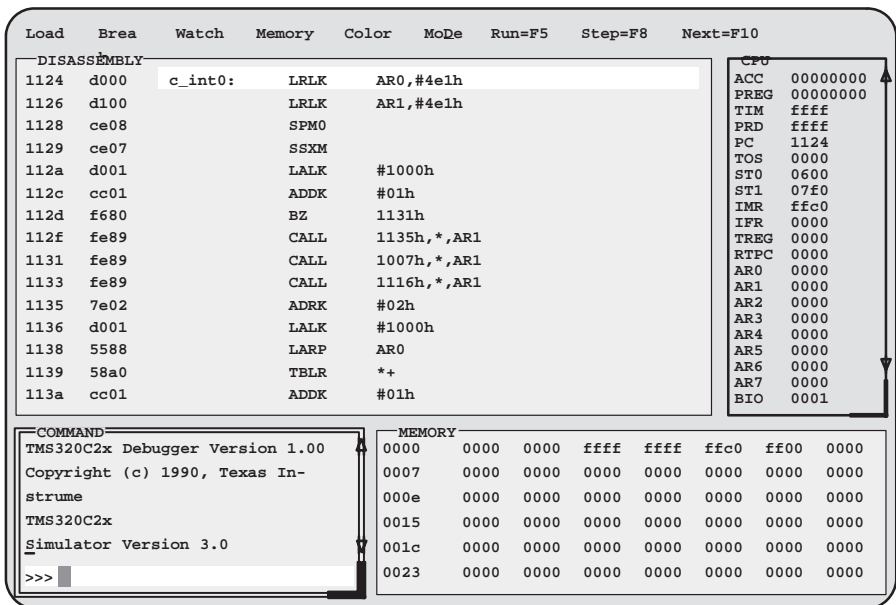
Each mode changes the debugger display by adding or hiding specific windows. Some windows, such as the COMMAND window, may be present in all modes. The following figures show the default displays for these modes and show the windows that the debugger automatically displays for these modes. In addition to the default windows shown in these illustrations, you can also display DISP windows and the WATCH window (see Section 5.2, page 5-5).

Auto mode

In **auto mode**, the debugger automatically displays whatever type of code is currently running—assembly language or C. This is the default mode; when you first invoke the debugger, you'll see a display similar to Figure 5-1. Auto mode has two types of displays:

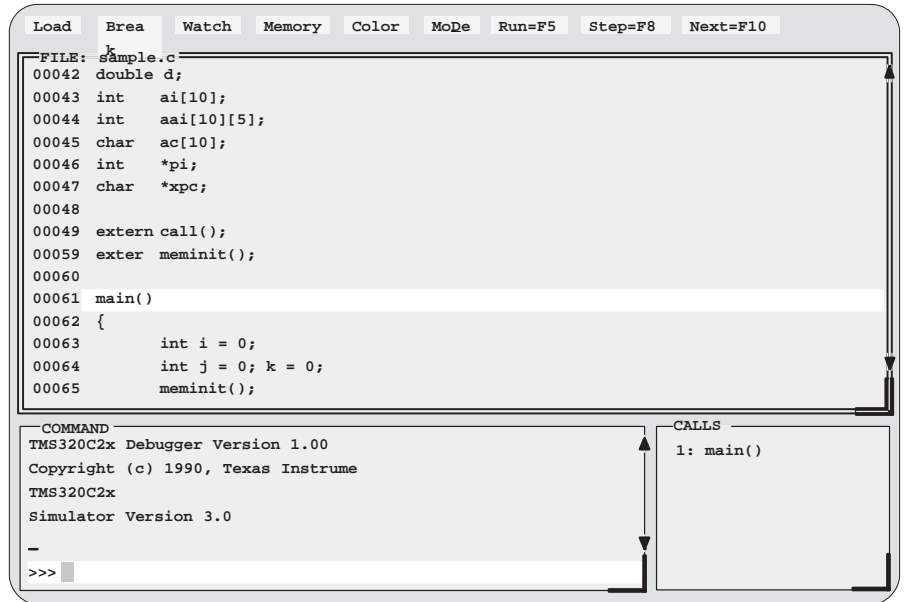
- When the debugger is running assembly language code, you'll see an assembly display similar to the one in Figure 5-1. The DISASSEMBLY window displays the reverse assembly of memory contents.

Figure 5-1. Typical Assembly Display (for Auto Mode and Assembly Mode)



- When the debugger is running C code, you'll see a C display similar to the one in Figure 5–2. (This assumes that the debugger can find your C source file to display in the FILE window. If the debugger can't find your source, then it switches to mixed mode.)

Figure 5–2. Typical C Display (for Auto Mode Only)



When you're running assembly language code, the debugger automatically displays windows as described for assembly mode.

When you're running C code, the debugger automatically displays the COMMAND, CALLS, and FILE windows. If desired, you can also open a WATCH window and DISP windows.

Assembly mode

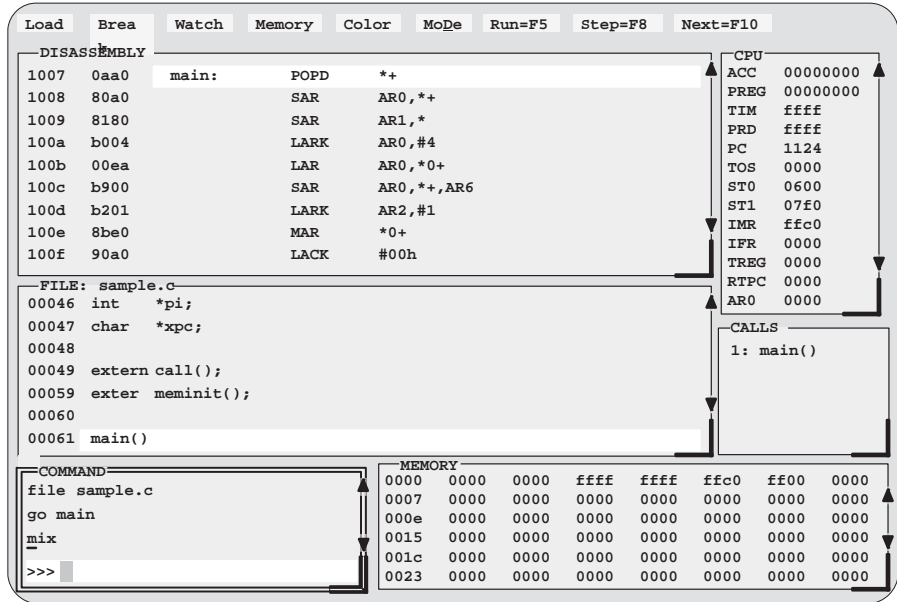
Assembly mode is for viewing assembly language programs only. In this mode, you'll see a display similar to the one shown in Figure 5–1. When you're in assembly mode, you'll always see the assembly display, regardless of whether C or assembly language is currently running.

Windows that are automatically displayed in assembly mode include the MEMORY window, the DISASSEMBLY of memory contents window, the CPU register window, and the COMMAND window. If you choose, you can also open a WATCH window in assembly mode.

Mixed mode

Mixed mode is for viewing assembly language and C code at the same time. Figure 5–3 shows the default display for mixed mode.

Figure 5–3. Typical Mixed Display (for Mixed Mode Only)



In mixed mode, the debugger displays all windows that can be displayed in auto and assembly modes—regardless of whether you’re currently running assembly language or C code. This is useful for finding bugs in C programs that exploit specific architectural features of the 'C2x.

Restrictions associated with debugging modes

The assembly language code that the debugger shows you is the disassembly (reverse assembly) of memory’s contents. If you load object code into memory, then the assembly language code is the disassembly of that object code. If you don’t load an object file, then the disassembly won’t 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. This applies to these commands:

- | | | |
|-------|------|------|
| dasm | func | mem |
| calls | file | disp |

5.2 Descriptions of the Different Kinds of Windows and Their Contents

The debugger can show several types of windows. This section lists the various types of windows and describes their characteristics.

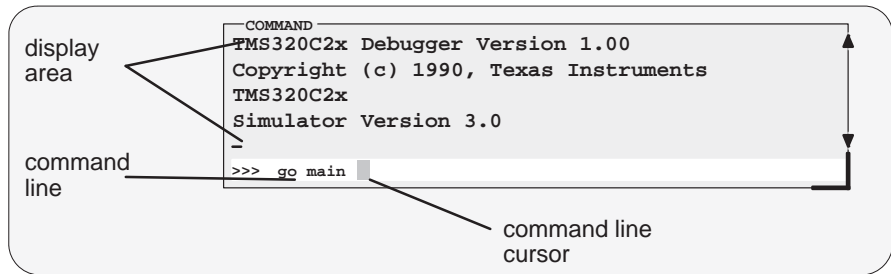
Every window is identified by a name in its upper left corner. Each type of window serves a specific purpose and has unique characteristics. There are eight different windows, divided into three general categories:

- The **COMMAND window** provides an area for typing in commands and for displaying various types of information such as progress messages, error messages, or command output.
- **Code-display windows** are for displaying assembly language or C code. There are three code-display windows:
 - The **DISASSEMBLY** window displays the disassembly (assembly language version) of memory contents.
 - The **FILE** window displays any text file that you want to display; its main purpose, however, is to display C source code.
 - The **CALLS** window identifies the current function traceback (when C code is running).
- **Data-display windows** are for observing and modifying various types of data. There are four data-display windows:
 - The **MEMORY** window displays the contents of a range of memory.
 - The **CPU** window displays the contents of 'C2x registers.
 - A **DISP** window displays the contents of an aggregate type such as an array or structure, showing the values of the individual members. You can display up to 120 DISP windows at one time.
 - A **WATCH** window displays selected data such as variables, specific registers, or memory locations.

You can move or resize any of these windows; you can also edit any value in a data-display window. Before you can perform any of these actions, however, you must select the window you want to move, resize, or edit, and make it *the active window*. For more information about making a window active, see Section 5.4, *The Active Window*, on page 5-17.

The remainder of this section describes the individual windows.

COMMAND window



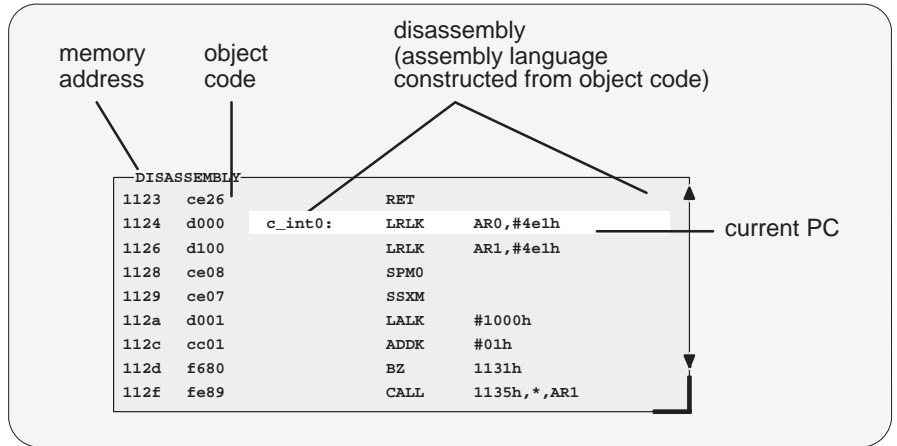
<i>Purpose</i>	Provides an area for entering commands Provides an area for echoing commands and displaying command output, errors, and messages
<i>Editable?</i>	Command line is editable; command output isn't
<i>Modes</i>	All modes
<i>Created</i>	Automatically
<i>Affected by</i>	All commands entered on the command line All commands that display output in the display area Any input that creates an error

The COMMAND window has two parts:

- Command line.** This is where you enter commands. When you want to enter a command, just type—no matter which window is active. The debugger keeps a list of the last 50 commands that you entered. You can select and re-enter commands from the list without retyping them.
- Display area.** This area of the COMMAND window echoes the command that you entered, shows any output from the command, and displays debugger messages.

For more information about the COMMAND window and entering commands, refer to Chapter 6, *Entering and Using Commands*.

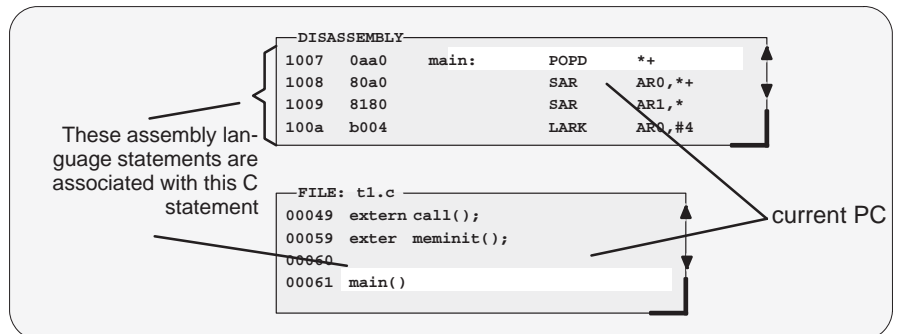
DISASSEMBLY window



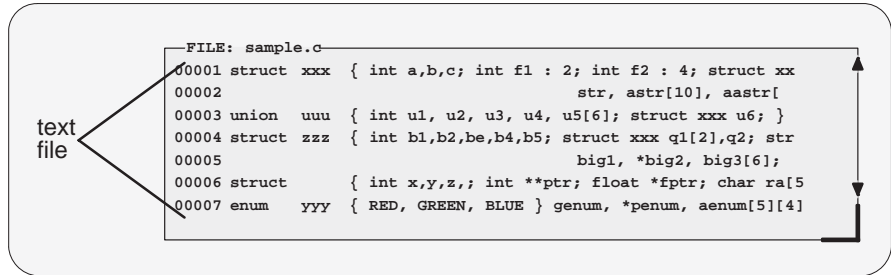
- Purpose** Displays the disassembly (or reverse assembly) of memory contents
- Editable?** No; pressing the edit key (**F9**) or the left mouse button sets a breakpoint on an assembly language statement
- Modes** Auto (assembly display only), assembly, and mixed
- Created** Automatically
- Affected by** DASM and ADDR commands
Breakpoint and run commands

Within the DISASSEMBLY window, the debugger highlights:

- The statement that the PC is pointing to (if that line is in the current display)
- Any breakpointed statements
- The address and object code fields for all statements associated with the current C statement, as shown below



FILE window



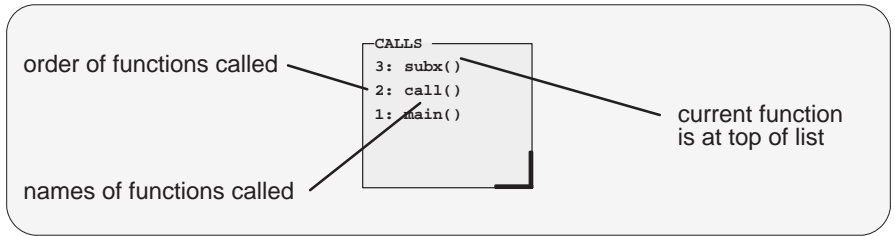
<i>Purpose</i>	Shows any text file you want to display
<i>Editable?</i>	No; if the FILE window displays C code, pressing the edit key (F9) or the left mouse button sets a breakpoint on a C statement
<i>Modes</i>	Auto (C display only) and mixed
<i>Created</i>	With the FILE command Automatically when you're in auto or mixed mode and your program begins executing C code
<i>Affected by</i>	FILE, FUNC, and ADDR commands Breakpoint and run commands

You can use the FILE command to display the contents of any file within the FILE window, but this window is especially useful for viewing C source files. Whenever you single-step a program or run a program and halt execution, the FILE window automatically displays the C source associated with the current point in your program. This overwrites any other file that may have been displayed in the window.

Within the FILE window, the debugger highlights:

- The statement that the PC is pointing to (if that line is in the current display)
- Any statements where you've set a breakpoint

CALLS window



<i>Purpose</i>	Lists the function you're in, its caller, and its caller, etc., as long as each function is a C function
<i>Editable?</i>	No; pressing the edit key (F9) or the left mouse button changes the FILE display to show the source associated with the called function
<i>Modes</i>	Auto (C display only) and mixed
<i>Created</i>	Automatically when you're displaying C code With the CALLS command if you closed the window
<i>Affected by</i>	Run and single-step commands

The display in the CALLS window changes automatically to reflect the latest function call.

If you haven't run any code, then no functions have been called yet. You'll also see this if you're running code but are not currently running a C function.

```
CALLS
1: **UNKNOWN
```

In C programs, the first C function is main.

```
CALLS
1: main()
```

As your program runs, the contents of the CALLS window change to reflect the current routine that you're in and where the routine was called from. When you exit a routine, its name is popped from the CALLS list.

```
CALLS
2: xcall()
1: main()
```

```
CALLS
1: main()
```

If a function name is listed in the CALLS window, you can easily display the function in the FILE window:



-
- 1) Point the mouse cursor at the appropriate function name that is listed in the CALLS window.
 - 2) Click the left mouse button. This displays the selected function in the FILE window.



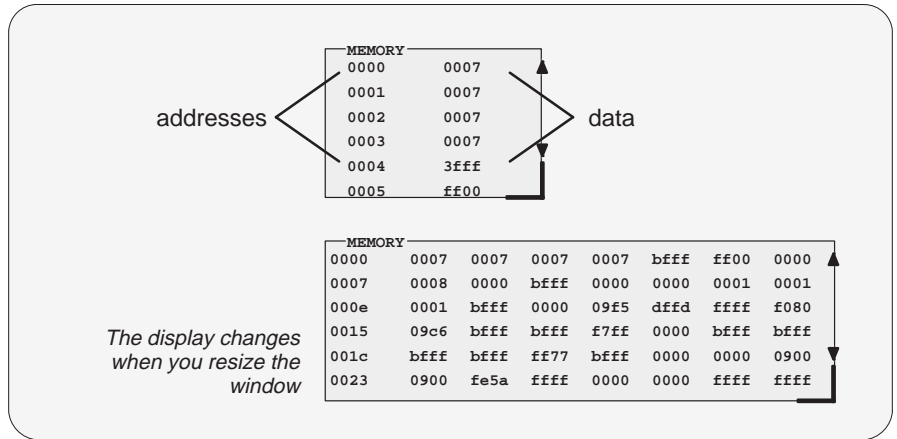
-
- 1) Make the CALLS window the active window (see Section 5.4, *The Active Window*, page 5-17).
 - 2) Use the arrow keys to move up/down through the list of function names until the appropriate function is indicated.
 - 3) Press **F9**. This displays the selected function in the FILE window.

You can close and reopen the CALLS window.

- Closing the window is a two-step process:
 - 1) Make the CALLS window the active window.
 - 2) Press **F4**.
- To reopen the CALLS window after you've closed it, enter the CALLS command. The format for this command is:

calls

MEMORY window



<i>Purpose</i>	Displays the contents of memory
<i>Editable?</i>	Yes—you can edit the data (but not the addresses)
<i>Modes</i>	Auto (assembly display only), assembly, and mixed
<i>Created</i>	Automatically
<i>Affected by</i>	The MEM command

The MEMORY window has two parts:

- Addresses.** The first column of numbers identifies the addresses of the first column of displayed data. No matter how many columns of data you display, only one address column is displayed. Each address in this column identifies the address of the data immediately to its right.
- Data.** The remaining columns display the values at the listed addresses. You can display more data by making the window wider and/or longer.

The first MEMORY window above has one column of data, so each new address is incremented by one. Although the second window shows four columns of data, there is still only one column of addresses; the first value is at address 0x0000, the second at address 0x0001, etc.; the fifth value (first value in the second row) is at address 0x0007, the sixth at address 0x0008, etc.

As you run programs, some memory values change as the result of program execution. The debugger highlights changed values. Depending on how you configure memory for your application, some locations may be invalid/unconfigured. The debugger also highlights these locations (by default, it shows these locations in red).

If you want to view different memory locations, use the MEM command to display a different block of memory. The basic syntax for this command is:

mem *address*

When you enter this command, the debugger changes the memory display so that *address* becomes the first displayed location (it's displayed in row 1, column 1).

The 'C2x has separate data and program spaces. By default, the MEMORY window shows data memory. If you want to display program memory, you can enter the MEM command like this:

mem *address@prog*

The **@prog** suffix identifies the *address* as a program memory address. (You can also use **@data** to display data memory, but since data memory is the default, the @data is unnecessary).

When you display program memory, the MEMORY window's label changes to remind you that you are no longer displaying data memory:

The MEMORY label changes to MEMORY [PROG]

Address	Hex Value	Hex Value	Hex Value	Hex Value	Hex Value	Hex Value	Hex Value
0000	ff80	1000	0000	0000	0000	0000	0000
0007	0000	0000	0000	0000	0000	0000	0000
000e	0000	0000	0000	0000	0000	0000	0000
0015	0000	0000	0000	0000	0000	0000	0000
001c	fefa	fdcf	7175	1454	57d3	5555	ffff

CPU window

register name

register contents

The display changes when you resize the window

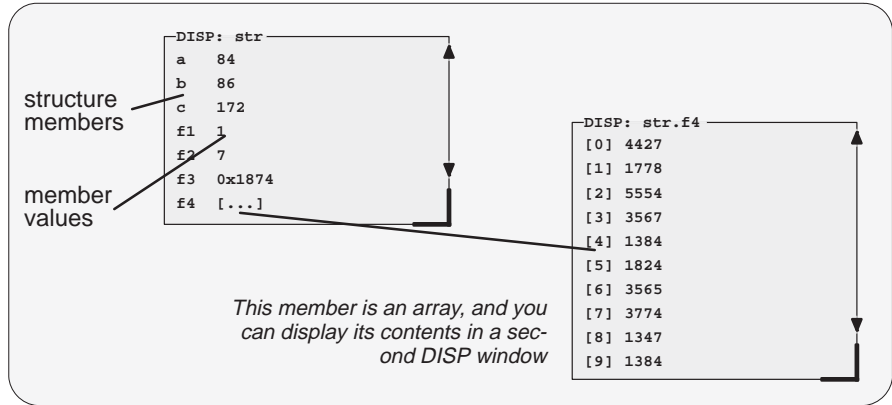
CPU	
ACC	00000000
PREG	00000000
TIM	ffff
PRD	ffff
PC	1124
TOS	0000
ST0	0600
ST1	07E0
IMR	ffc0
IFR	0000
TREG	0000
RTPC	0000
AR0	0000
AR1	0000
AR2	0000
AR3	0000
AR4	0000
AR5	0000

CPU					
ACC	00000000	PREG	00000000		
TIM	ffff	PRD	ffff		
PC	1124	TOS	0000	ST0	0600
IMR	ffc0	IFR	0000	TREG	0000
AR0	0000	AR1	0000	AR2	0000
				AR3	0000

- Purpose* Shows the contents of the 'C2x registers
- Editable?* Yes—you can edit the value of any displayed register
- Modes* Auto (assembly display only), assembly, and mixed
- Created* Automatically
- Affected by* Data-management commands

As you run programs, some values displayed in the CPU window change as the result of program execution. The debugger highlights changed values.

DISP windows



<i>Purpose</i>	Displays the members of a selected structure, array or pointer, and the value of each member
<i>Editable?</i>	Yes—you can edit individual values
<i>Modes</i>	Auto (C display only) and mixed
<i>Created</i>	With the DISP command
<i>Affected by</i>	The DISP command

A DISP window is similar to a WATCH window, but it shows the values of an entire array or structure instead of a single value. Use the DISP command to open a DISP window; the syntax is:

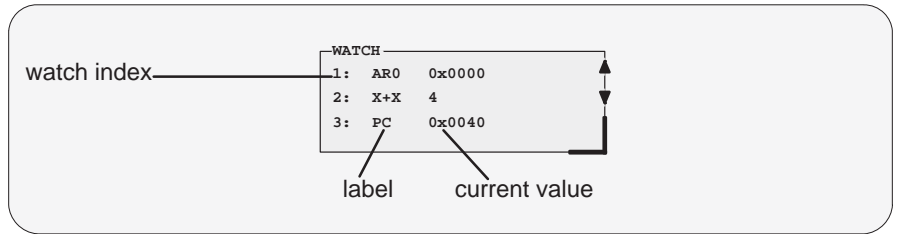
disp *expression*

Data is displayed in its natural format:

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

If any of the displayed members are arrays, structures, or pointers, you can bring up additional DISP windows to display their contents—up to 120 DISP windows can be open at once.

WATCH window



- Purpose* Displays the values of selected expressions
- Editable?* Yes—you can edit the value of any expression whose value specifies a storage location (in registers or memory). In the window above, for example, you could edit the value of PC but couldn't edit the value of X+X.
- Modes* Auto, assembly, and mixed
- Created* With the WA command
- Affected by* WA, WD, and WR commands

The WATCH window helps you to track the values of arbitrary expressions, variables, and registers. Use the WA command for this; the syntax is:

wa *expression* [, *label*]

WA adds *expression* to the WATCH window. (If there's no WATCH window, then WA also opens a WATCH window).

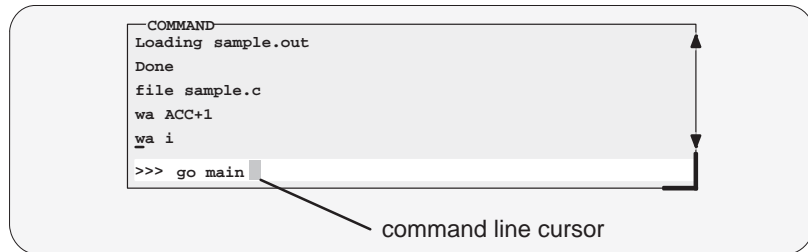
To delete individual entries from the WATCH window, use the WD command. To delete all entries at once and close the WATCH window, use the WR command.

Although the CPU window displays register contents, you may not be interested in the values of all these registers. In this situation, it is convenient to use the WATCH window to track the values of the specific registers you're interested in.

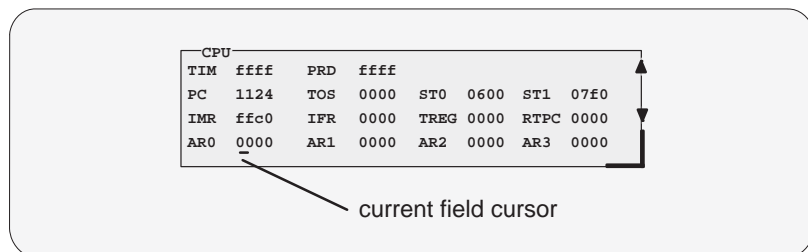
5.3 Cursors

The debugger display has three types of cursors:

- ❑ The **command-line cursor** is a block-shaped cursor that identifies the current character position on the command line. Arrow keys *do not* affect the position of this cursor.



- ❑ The **mouse cursor** is a block-shaped cursor that tracks mouse movements over the entire display. This cursor is controlled by the mouse driver installed on your system; if you haven't installed a mouse, you won't see a mouse cursor on the debugger display.
- ❑ The **current-field cursor** identifies the current field in the active window. This is the hardware cursor that is associated with your EGA card. Arrow keys *do* affect this cursor's movement.



5.4 The Active Window

The windows in the debugger display aren't fixed in their position or in their size. You can resize them, move them around, and, in some cases, close them. The window that you're going to move, resize, or close must be **active**.

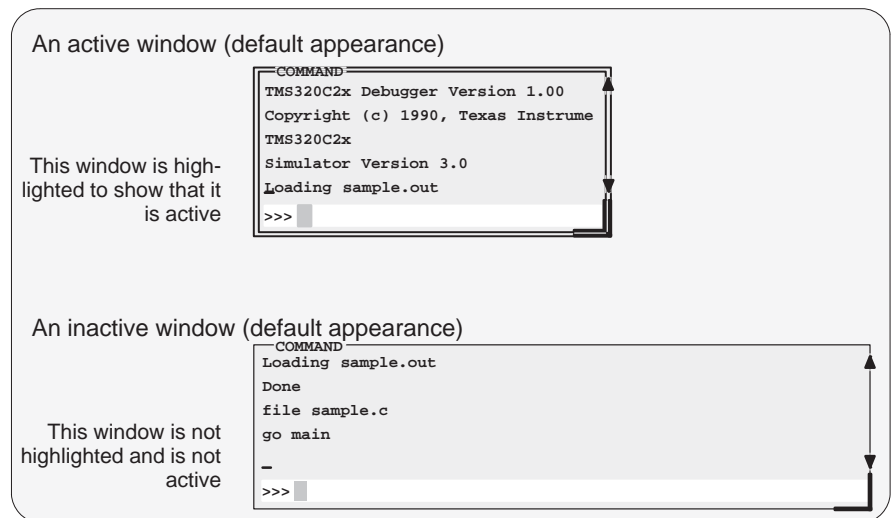
You can move, resize, or close *only one window at a time*; thus, only one window at a time can be the **active window**. Whether or not a window is active doesn't affect the debugger's ability to update information in a window—it affects only your ability to manipulate a window.

Identifying the active window

The debugger highlights the active window. When windows overlap on your display, the debugger pops the active window to be on top of other windows.

You can alter the active window's border style and colors if you wish; Figure 5–4 illustrates the default appearance of an active window and an inactive window.

Figure 5–4. Default Appearance of an Active and an Inactive Window



Note: On **monochrome monitors**, the border and selection corner are highlighted as shown in the illustration. On **color monitors**, the border and selection corner are highlighted as shown in the illustration, but they also change color (by default, they change from white to yellow).

Selecting the active window

You can use one of several methods for selecting the active window:



1) Point to any location within the boundaries or on any border of the desired window.



2) Click the left mouse button.

Note that if you point within the window, you might also select the current field. For example:

- If you point inside the CPU window, then the register you're pointing at becomes active and the debugger treats any text that you type as a new register value. If you point inside the MEMORY window, then the address value you're pointing at becomes active, and the debugger treats any text that you type as a new memory value.

*Press **ESC** to get out of this.*

- If you point inside the DISASSEMBLY or FILE window, you'll set a breakpoint on the statement you're pointing to.

Press the button again to clear the breakpoint.



This key hops through the windows on your display, making each one active in turn and making the previously active window inactive. Pressing this key highlights one of the windows, showing you that the window is active. Pressing **F6** again makes a different window active. Press **F6** as many times as necessary until the desired window becomes the active window.





win The WIN command allows you to select the active window by name. The format of this command is:

win *WINDOW NAME*

Note that the *WINDOW NAME* is in uppercase (matching the name exactly as displayed). You can spell out the entire window name, but you really need specify only enough letters to identify the window.

For example, to select the DISASSEMBLY window as the active window, you could enter either of these two commands:

win DISASSEMBLY 
or **win DISA** 

If several windows of the same type are visible on the screen, don't use the WIN command to select one of them.

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 doesn't find the window you asked for (because you closed the window or misspelled the name), then the WIN command has no effect.

5.5 Manipulating Windows

A window's size and its position in the debugger display aren't fixed—you can resize and move windows.

Note: Which Windows Can Be Resized?

You can resize or move any window, but first the window must be **active**. For information about selecting the active window, refer to Section 5.4 (page 5-17).

Resizing a window

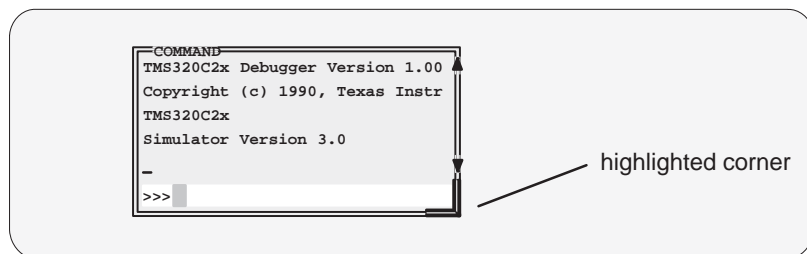
The minimum window size is three lines by four characters. The maximum window size varies, depending on which screen size you're using, but you can't make a window larger than the screen.

There are two basic ways to resize a window:

- You can resize a window by using the mouse.
- You can resize a window by using the SIZE command.



- 1) Point to the lower right corner of the window. This corner is highlighted—here's what it looks like:



- 2) Grab the highlighted corner by pressing one of the mouse buttons; while pressing the button, move the mouse in any direction. This resizes the window.
- 3) Release the mouse button when the window reaches the desired size.



size The SIZE command allows you to size the active window. The format of this command is:

size [*width, length*]

You can use the SIZE command in one of two ways:

Method 1 Supply a specific *width* and *length*

Method 2 Omit the *width* and *length* parameters and use arrow keys to interactively resize the window.

SIZE, method 1: Use *width* and *length* parameters. Valid values for the width and length depend on the screen size and the window position on the screen. Table 5–1 lists the minimum and maximum window sizes.

Table 5–1. *Width and Length Limits for Window Sizes*

Screen size	Debugger option	Valid widths	Valid lengths
80 characters by 25 lines	none	4 through 80	3 through 24
80 characters by 39 lines [†]	–b	4 through 80	3 through 38
80 characters by 43 lines [‡]			3 through 42
80 characters by 50 lines [§]			3 through 49
120 characters by 43 lines	–bb	4 through 120	3 through 42
132 characters by 43 lines	–bbb	4 through 132	3 through 42
80 characters by 60 lines	–bbbb	4 through 80	3 through 59
100 characters by 60 lines	–bbbbb	4 through 100	3 through 59

[†] PC version of simulator running under Microsoft Windows



[‡] PC with EGA card; Sun

[§] PC with VGA card

Note: To use a larger screen size, you must invoke the debugger with one of the –b options.

The maximum sizes assume that the window is in the upper left corner (beneath the menu bar). If a window is in the middle of the display for example, you can't size it to the maximum height and width—you can size it only to the right and bottom screen borders.

For example, if you want to use commands to make the CALLS window 8 characters wide by 20 lines long, you could enter:

```
win CALLS 
size 8, 20 
```

SIZE, method 2: Use arrow keys to interactively resize the window. If you enter the SIZE command without *width* and *length* parameters, you can use arrow keys to size the window:

- ⏴ Makes the active window one line longer.
- ⏵ Makes the active window one line shorter.
- ⏪ Makes the active window one character narrower.
- ⏩ Makes the active window one character wider.

When you're finished using the cursor keys, you *must* press `ESC` or `↵`.

For example, if you want to make the CPU window three lines longer and two characters narrower, you can enter:

```
win CPU ↵  
size ↵  
⏴ ⏴ ⏴ ⏪ ⏪ ESC
```

Moving a window

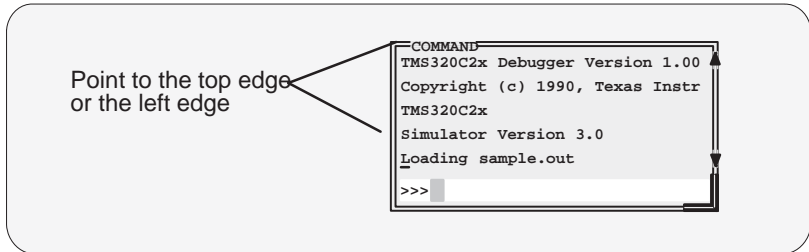
The windows in the debugger display don't have fixed positions—you can move them around.

There are two ways to move a window:

- You can move a window by using the mouse
- You can move a window by using the MOVE command



- 1) Point to the left or top edge of the window.



- 2) Press the left mouse button, but don't release it; now move the mouse in any direction.
- 3) Release the mouse button when the window is in the desired position.



move The MOVE command allows you to move the active window. The format of this command is:

```
move [X position, Y position [, width, length ] ]
```

You can use the MOVE command in one of two ways:

Method 1 Supply a specific *X position* and *Y position*

Method 2 Omit the *X position* and *Y position* parameters and use arrow keys to interactively resize the window

MOVE, method 1: Use the *X position* and *Y position* parameters. You can move a window by defining a new XY position for the window's upper left corner. Valid X and Y positions depend on the screen size and the window size. Table 5–2 lists the minimum and maximum XY positions.

Table 5–2. Minimum and Maximum Limits for Window Positions

Screen size	Debugger option	Valid X positions	Valid Y positions
80 characters by 25 lines	none	0 through 76	1 through 22
80 characters by 39 lines [†]	–b	0 through 76	1 through 36
80 characters by 43 lines [‡]			1 through 40
80 characters by 50 lines [§]			1 through 47
120 characters by 43 lines	–bb	0 through 116	1 through 40
132 characters by 43 lines	–bbb	0 through 128	1 through 40
80 characters by 60 lines	–bbbb	0 through 76	1 through 57
100 characters by 60 lines	–bbbbb	0 through 106	1 through 57

[†] PC version of simulator running under Microsoft Windows

[‡] PC with EGA card; Sun

[§] PC with VGA card

Note: To use a larger screen size, you must invoke the debugger with one of the –b options.

The maximum values assume that the window is as small as possible; for example, if a window is half as tall as the screen, you won't be able to move its upper left corner to an X position on the bottom half of the screen.

For example, if you want to use commands to move the DISASSEMBLY position to a place in the upper left area of the display, you might enter:

```
win DISASSEMBLY   
move 5, 6 
```

MOVE, method 2: Use arrow keys to interactively move the window. If you enter the MOVE command without *X position* and *Y position* parameters, you can use arrow keys to move the window:

- ⬇ Moves the active window down one line.
- ⬆ Moves the active window up one line.
- ⬅ Moves the active window left one character position.
- ➡ Moves the active window right one character position.

When you're finished using the cursor keys, you *must* press `ESC` or `↵`.

For example, if you want to move the COMMAND window up two lines and right five characters, you can enter:

```
win COM ↵  
move ↵  
⬆ ⬆ ➡ ➡ ➡ ➡ ➡ ESC
```

Note: Resizing the Window as You Move the Window

If you choose, you can resize a window at the same time you move it. To do this, use the *width* and *length* parameters in the same way that they are used for the SIZE command.

5.6 Manipulating a Window's Contents

Although you may be concerned with changing the way windows appear in the display—where they are and how big/small they are—you'll usually be interested in something much more important: *what's in the windows*. Some windows contain more information than can be displayed on a screen; others contain information that you'd like to change. This section tells you how to view the hidden portions of data within a window and which data can be edited.

Note: Which Windows Can Be Scrolled and Edited?

You can scroll and edit only the **active window**. For information about selecting the active window, refer to Section 5.4 (page 5-17).

Scrolling through a window's contents

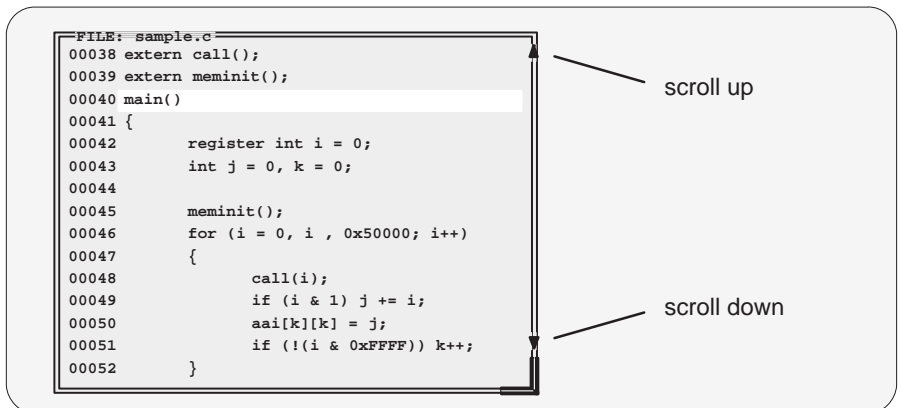
If you resize a window to make it smaller, you may hide information. Sometimes, a window may contain more information than can be displayed on a screen. In these cases, the debugger allows you to scroll information up and down within the window.

There are two ways to view hidden portions of a window's contents:

- You can use the mouse to scroll the contents of the window.
- You can use function keys and arrow keys.



You can use the mouse to point to the scroll arrows on the righthand side of the active window. This is what the scroll arrows look like:



To scroll window contents up or down:

- 1) Point to the appropriate scroll arrow.
- 2) Press the left mouse button; continue to press it until the information you're interested in is displayed within the window.
- 3) Release the mouse button when you're finished scrolling.

You can scroll up/down one line at a time by pressing the mouse button and releasing it immediately.



In addition to scrolling, the debugger supports the following methods for moving through a window's contents.

PAGE UP

The page-up key scrolls up through the window contents, one window length at a time. You can use **CONTROL** **PAGE UP** to scroll up through an array of structures displayed in a DISP window.

PAGE DOWN

The page-down key scrolls down through the window contents, one window length at a time. You can use **CONTROL** **PAGE DOWN** to scroll down through an array of structures displayed in a DISP window.

HOME

When the FILE window is active, pressing **HOME** adjusts the window's contents so that the first line of the text file is at the top of the window. You can't use **HOME** outside of the FILE window.

END

When the FILE window is active, pressing **END** adjusts the window's contents so that the last line of the file is at the bottom of the window. You can't use **END** outside of the FILE window.



Moves the field cursor up one line at a time.



Moves the field cursor down one line at a time.



In the FILE window, scrolls the display left eight characters at a time. In other windows, moves the field cursor left one field; at the first field on a line, wraps back to the last fully displayed field on the previous line.



In the FILE window, scrolls the display right eight characters at a time. In other windows, moves the field cursor right one field; at the last field on a line, wraps around to the first field on the next line.

Editing the data displayed in windows

You can edit the data displayed in the MEMORY, CPU, DISP, and WATCH windows by using an overwrite "click and type" method or by using commands

that change the values. (This is described in detail in Section 9.3, *Basic Methods for Changing Data Values*, page 9-4.)

Note: “Editing” the FILE, DISASSEMBLY, and CALLS Windows

In these windows, the “click and type” method of selecting data for editing—pointing at a line and pressing (F9) or the left mouse button—does not allow you to modify data.

In the FILE and DISASSEMBLY windows, pressing (F9) or the mouse button sets or clears a breakpoint on any line of code that you select. You can't modify text in a FILE or DISASSEMBLY window.

In the CALLS window, pressing (F9) or the mouse button shows the source for the function named on the selected line.

5.7 Closing a Window

The debugger opens various windows on the display according to the debugging mode you select. When you switch modes, the debugger may close some windows and open others. Additionally, you may choose to open DISP and WATCH windows.

Most of the windows remain open—you can't close them. However, you can close the CALLS, DISP, and WATCH windows.

To close the CALLS window:

- 1) Make the CALLS window the active window.
- 2) Press **F4**.

To close a DISP window:

- 1) Make the appropriate DISP window the active window.
- 2) Press **F4**.

If the DISP window that you close has any children, they are closed also.

To close the WATCH window, enter:

WT **F4**

Entering and Using Commands

The debugger provides you with several methods for entering commands and accomplishing other tasks within the debugger environment. There are several ways to enter commands: from the command line, from pulldown menus, with a mouse, and with function keys. Mouse and function key use differ from situation to situation, and their use is described throughout this book whenever applicable. Certain specific rules apply to entering commands and using pulldown menus, however, and this chapter includes this information.

Some restrictions apply to command entry for VAX and Sun versions of the simulator. For descriptions of these restrictions, refer to subsection 2.2.3, page 2-10 (VAX) or subsection 2.3.3, page 2-13 (Sun).

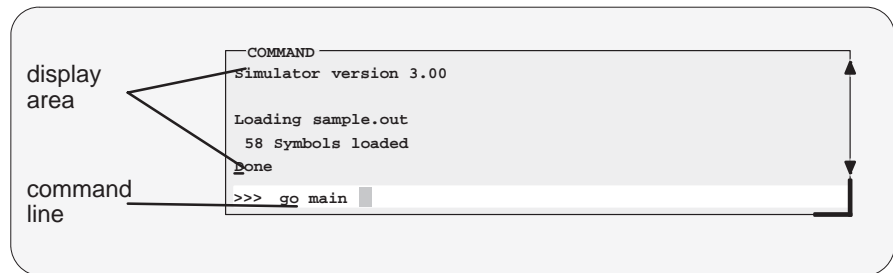
Topic	Page
<i>Some of the alternative methods for entering commands don't apply to all commands—however, entering the command from the command line is a method that works for all commands.</i>	6.1 Entering Commands From the Command Line 6-2
	How to type in and enter commands 6-3
	Sometimes, you can't type a command 6-4
	Using the command history 6-4
	Clearing the display area 6-5
<i>The pulldown menus and dialog boxes provide you with another easy method for entering commands. You can use this method even if you don't have a mouse.</i>	6.2 Using the Menu Bar and the Pulldown Menu 6-6
	Using the pulldown menu 6-7
	Escaping from the pulldown menu 6-8
	Entering parameters in a dialog box 6-9
	Using menu bar selections that don't have pulldown menus 6-10
	How the menu selections correspond to commands 6-11
<i>The debugger allows you to execute often-needed command sequences by keeping the commands in a batch file. The debugger also allows you to perform some simple system commands from within the debugger environment.</i>	6.3 Entering Commands From a Batch File 6-13
	6.4 Additional System Commands 6-14

6.1 Entering Commands From the Command Line

The debugger supports a complete set of commands that help you to control and monitor program execution, customize the display, and perform other tasks. These commands are discussed in various sections throughout this book, as they apply to the current topic. Chapter 12 summarizes all of the debugger commands with an alphabetical reference.

Although there are a variety of methods for entering most of the commands, *all* of the commands can be entered by typing them on the command line in the COMMAND window. Figure 6–1 shows the COMMAND window.

Figure 6–1. The COMMAND Window



The COMMAND window serves two purposes:

- The **command line** portion of the window provides you with an area for entering commands. For example, the command line in Figure 6–1 shows that a GO command was typed in (but not yet entered).
- The **display area** provides the debugger with an area for echoing commands, displaying command output, or displaying errors and messages for you to read. For example, the command output in Figure 6–1 shows the messages that are displayed when you first bring up the debugger and also shows that a GO MAIN command was entered.

If you enter a command by using an alternate method (using the mouse, a pulldown menu, or function keys), the COMMAND window doesn't echo the entered command.









How to type in and enter commands

You can type a command at almost any time; the debugger automatically places the text on the command line when you type. When you want to enter a command, just type—no matter which window is active. You don't have to worry about making the COMMAND window active or moving the field cursor to the command line. When you start to type, the debugger usually assumes that you're typing a command and puts the text on the command line (except under certain circumstances, which are explained on the next page). Commands themselves are not case sensitive, although some parameters (such as window names) are.



To execute a command that you've typed, just press . The debugger then:

- 1) Echoes the command to the display area,
- 2) Executes the command and displays any resulting output, and
- 3) Clears the command line when command execution completes.

Once you've typed a command, you can edit the text on the command line with these keystrokes:

To...	Press...
Move back over text without erasing characters	  OR 
Move forward through text without erasing characters	 
Move back over text while erasing characters	
Move forward through text while erasing characters	
Insert text into the characters that are already on the command line	

Note: Several Points About Typing Commands on the Command Line

- You cannot use the arrow keys to move through or edit text on the command line.
- Typing a command doesn't make the COMMAND window the active window.
- If you press  when the cursor is in the middle of text, the debugger truncates the input text at the point where you press .

Sometimes, you can't type a command

At most times, you can press any alphanumeric or punctuation key on your keyboard (any printable character); the debugger interprets this as part of a command and displays the character on the command line. In a few instances, however, pressing an alphanumeric key is not interpreted as information for the command line.

- When you're pressing the **ALT** key, typing certain letters causes the debugger to display a pulldown menu.
- When a pulldown menu is displayed, typing a letter causes the debugger to execute a selection from the menu.
- When you're pressing the **CONTROL** key, pressing **H** or **L** moves the command-line cursor backward or forward through the text on the command line.
- When you're editing a field, typing enters a new value in the field.
- When you're using the **MOVE** or **SIZE** command interactively, pressing keys affects the size or position of the active window. Before you can enter any more commands, you must press **ESC** to terminate the interactive moving or sizing.
- When you've brought up a dialog box, typing enters a parameter value for the current field in the box.

Using the command history

The debugger keeps an internal list, or **command history**, of the commands that you enter. It remembers the last 100 commands that you entered. If you want to re-enter a command, you can move through this list, select a command that you've already executed, and re-execute it.

Use these keystrokes to move through the command history.

To...	Press...
Move forward through the list of executed commands, one by one	SHIFT TAB
Move backward through the list of executed commands, one by one	TAB
Execute the last command in the list	F2

As you move through the command history, the debugger displays the commands, one by one, on the command line. When you see a command that you want to execute, simply press **↵** to execute the command. You can also edit these displayed commands in the same manner that you can edit new commands.

Clearing the display area

Occasionally, you may want to completely blank out the display area of the COMMAND window; the debugger provides a command for this:



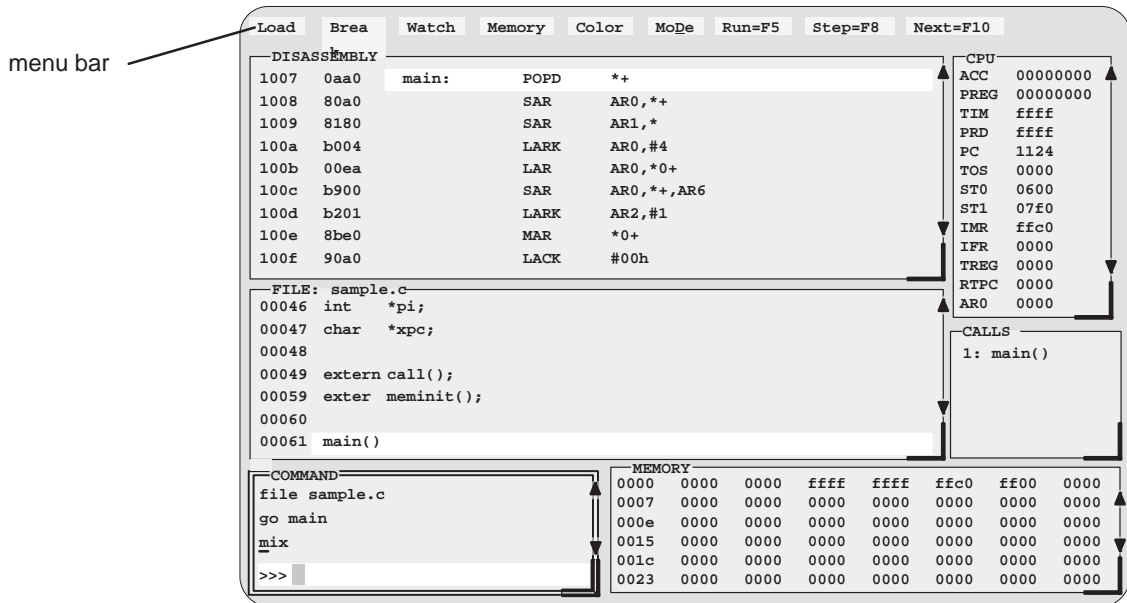
cls Use the CLS command to clear all displayed information from the display area. The format for this command is:

cls

6.2 Using the Menu Bar and the Pulldown Menus

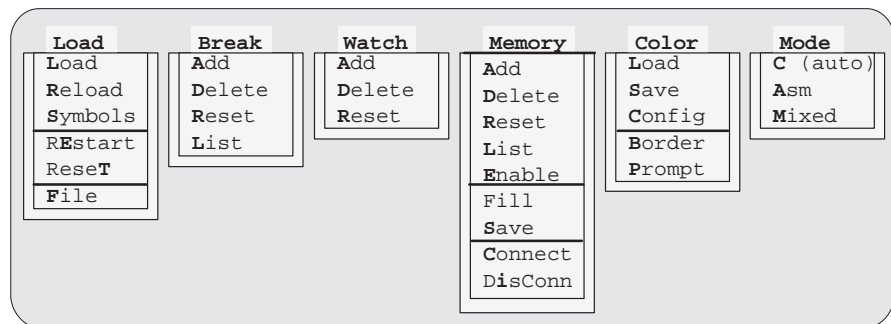
In all three of the debugger displays, you'll see a menu bar at the top of the screen. The menu selections offer you an alternative method for entering many of the debugger commands. Figure 6–2 points out the menu bar in a mixed-mode display. There are several ways to use the selections on the menu bar, depending on whether the selection has a pull-down menu or not.

Figure 6–2. The Menu Bar in the Debugger Display



Several of the selections on the menu bar have pull-down menus; if they could all be pulled down at once, they'd look like Figure 6–3.

Figure 6–3. All of the Pull-down Menus



Note that the menu bar and associated pulldown menus occupy fixed positions on the display. Unlike windows, you can't move, resize, or cover the menu bar or pulldown menus.

Using the pulldown menus





There are several ways to display the pulldown menus and then execute your selections from them. Executing a command from a menu is similar to executing a command by typing it in.

- If you select a command that has no parameters, then the debugger executes the command as soon as you select it.
- If you select a command that has one or more parameters, the debugger displays a **dialog box** when you make your selection. A dialog box offers you the chance to type in the parameter values for the command.





The following paragraphs describe several methods for selecting commands from the pulldown menus.



Mouse method 1

- 1)  Point the mouse cursor at one of the appropriate selections in the menu bar.
- 2)  Press the left mouse button, but don't release the button.
- 3)  While pressing the mouse button, move the mouse downward until your selection is highlighted on the menu.
- 4)  When your selection is highlighted, release the mouse button.

Mouse method 2

- 1)  Point the cursor at one of the appropriate selections in the menu bar.
- 2)  Click the left mouse button. This displays the menu until you are ready to make a selection.
- 3)  Point the mouse cursor at your selection on the pulldown menu.
- 4)  When your selection is highlighted, click the left mouse button.



Keyboard method 1

- 1) Press the **ALT** key; don't release it.
- 2) Press the key that corresponds to the highlighted letter in the selection name; release both keys. This displays the menu and freezes it.
- 3) Press and release the key that corresponds to the highlighted letter of your selection in the menu.

Keyboard method 2

- 1) Press the **ALT** key; don't release it.
- 2) Press the key that corresponds to the highlighted letter in the selection name; release both keys. This displays the menu and freezes it.
- 3) Use the arrow keys to move up and down through the menu.
- 4) When your selection is highlighted, press **↵**.



Escaping from the pulldown menus

- If you display a menu and then decide that you don't want to make a selection from this menu, you can:
 - Press **ESC**
 - or**
 - Point the mouse outside of the menu; press and then release the left mouse button.
- If you pull down a menu and see that it is not the menu you wanted, you can point the mouse at another entry and press the left mouse button, or you can use the **←** and **→** keys to display adjacent menus.

Entering parameters in a dialog box

Many of the debugger commands have parameters. When you execute these commands from menus, you must have some way of providing parameter values. The debugger allows you to do this by displaying a **dialog box** that asks for these values.

Entering parameter values in a dialog box is much like entering commands on the command line:

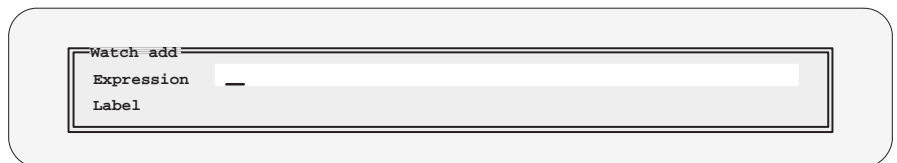
- If you press  in the middle of a string of text, the debugger truncates the string at that point.
- When you display a dialog box for the first time during a debugging session, the parameter fields are empty. When you bring up the same dialog box again, though, the box displays the last values that you entered. (This is similar to having a command history.) If you want to use the same value, just press .
- You can edit what you type (or values that remain from previous entry) in the same way that you can edit text on the command line.

When the dialog box displays more than one parameter, you can use the arrow keys to move from parameter to parameter. You can omit entries for optional parameters, but the debugger won't allow you to skip required parameters.


For example, the Add entry on the Watch menu is equivalent to the WA command. This command has two parameters:

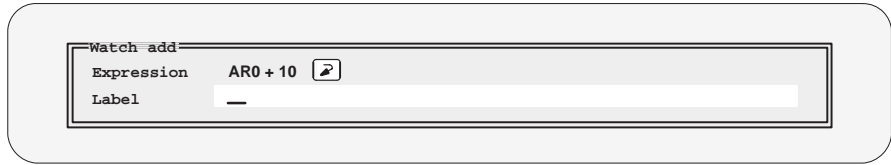
wa *expression* [, *label*]


When you select Add from the menu, the debugger displays a dialog box that asks you for this parameter information. The dialog box looks like this:



The image shows a dialog box with a title bar that says "Watch add". Inside the dialog box, there are three labels with corresponding input fields: "Expression" with a text input field containing a cursor, and "Label" with a text input field.

You can type in an *expression* just as you'd type in an expression if you were typing the WA command, and then press . The cursor moves down to the next parameter:



In this case, the next parameter (*label*) is optional. If you want to enter a parameter, you may do so; if you don't want to use this parameter, don't type anything in the field. When you've entered your final choices, press . The debugger closes the dialog box and executes the command with the parameter values you supplied.

Using menu bar selections that don't have pulldown menus

These three menu bar selections are single-level entries without pulldown menus:






There are two ways to execute these choices.



- 1) Point the cursor at one of these selections in the menu bar.
- 2) Click the left mouse button.

This executes your choice in the same manner as typing in the associated command without its optional *expression* parameter.

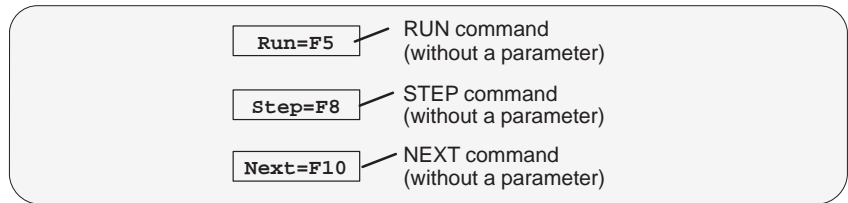


-  Pressing this key is equivalent to typing in the RUN command without an *expression* parameter.
-  Pressing this key is equivalent to typing in the STEP command without an *expression* parameter.
-  Pressing this key is equivalent to typing in the NEXT command without an *expression* parameter.

How the menu selections correspond to commands

The following sample screens illustrate the relationship of the debugger commands to the menu bar and pulldown menus.

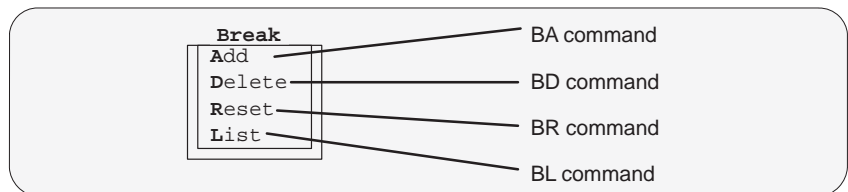
program execution (run) commands



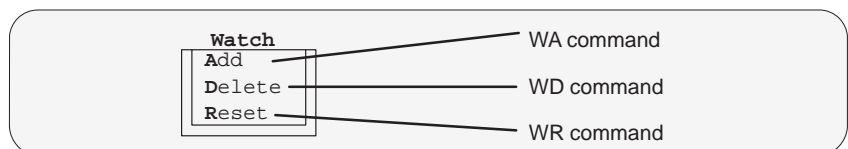
file/load commands



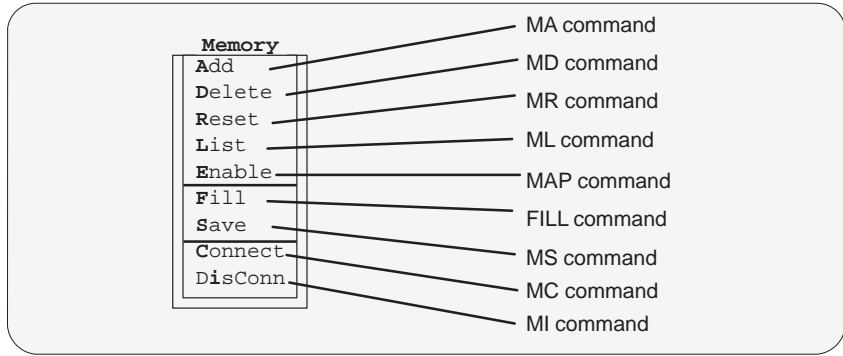
breakpoint commands



watch commands



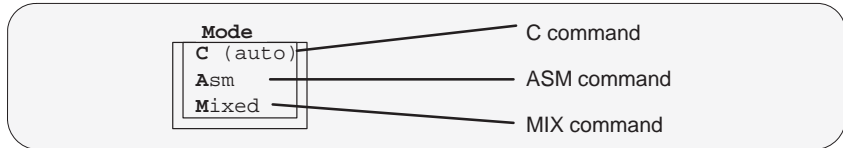
memory commands



screen-configuration commands



mode commands



6.3 Entering Commands From a Batch File

You can place debugger commands in a batch file and execute the file from within the debugger environment. This is useful, for example, for setting up a memory map that contains several MA commands followed by a MAP command that enables memory mapping.



take Use the TAKE command to tell the debugger to read and execute commands from a batch file. A batch file can call another batch file; they can be nested in this manner up to 10 deep. To halt the debugger's execution of a batch file, press **ESC**.

The format for the TAKE command is:

take *batch filename* [, *suppress echo flag*]

- The *batch filename* parameter identifies the file that contains commands.
 - If you supply path information with the *filename*, the debugger looks for the file only in the specified directory.
 - If you don't supply path information with the *filename*, the debugger looks for the file in the current directory.
 - If the debugger can't find the file in the current directory, it looks in any directories that you identified with the D_DIR environment variable. You can set D_DIR within the DOS environment; the command for doing this is:

```
SET D_DIR=C:\pathname;C:\pathname
```

This allows you to name several directories that the debugger can search. Remember that if you use D_DIR, you must set it *before you invoke the debugger*—the debugger doesn't recognize the DOS SET command. If you often use the same directories, it may be convenient to set D_DIR in your autoexec.bat file.

- By default, the debugger echoes the commands in the COMMAND window display area and updates the display as it reads commands from the batch file.
 - If you don't use the *suppress echo flag* parameter, or if you use it but supply a nonzero value, then the debugger behaves in the default manner.
 - If you would like to suppress the echoing and updating, use the value 0 for the *suppress echo flag* parameter.

6.4 Additional System Commands

In addition to the commands that the debugger supports for controlling and monitoring program execution and maintaining the display, the debugger supports several system-manipulation commands that perform DOS-like functions.



dir Use the DIR command to list the contents of a directory. The format for this command is:

dir [*directory name*]

This 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 don't use this parameter, the debugger lists the contents of the current directory.

You can use wildcards as part of the *directory name*.

cd Use the CHDIR (CD) command to change the current working directory. The format for this command is:

chdir *directory name*

or **cd** *directory name*

This changes the current directory to the specified *directory name*. You can use relative pathnames as part of the directory name. Note that this command can affect any command whose parameter is a filename (such as the FILE, LOAD, and TAKE commands).

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 can't access. Note that the commands described in this chapter can also be entered using the Memory pulldown menu.

Topic	Page
<i>This chapter shows you how to set up a memory map for your system.</i>	7.1 The Memory Map: What It Is and Why You Must Define It 7-2
	7.2 A Sample Memory Map 7-3
	Defining a memory map for the simulator 7-3
	Defining a memory map for the SWDS 7-4
	7.3 Identifying Usable Memory Ranges 7-6
	7.4 Enabling Memory Mapping 7-7
	7.5 Checking the Memory Map 7-7
<i>When you are using either the simulator or the SWDS, you can add simulated I/O ports to the memory map and associate the ports with input or output files.</i>	7.6 Modifying the Memory Map During a Debugging Session 7-8
	Returning to the original memory map 7-9
	7.7 Simulating I/O Space 7-10
	Connecting an I/O port 7-10
	Observing serial port data 7-11
Configuring memory to use serial port simulation 7-12	
Disconnecting an I/O port 7-13	

7.1 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 can't access. Memory maps vary, depending on the application. Typically, the map matches the MEMORY definition in your linker command file. (For information about the MEMORY directive and setting up a linker command file, see the *TMS320 Floating-Point DSP Assembly Language Tools User's Guide*.)

When memory mapping is enabled, the debugger checks each of its memory accesses against the provided memory map. If you attempt to access an undefined or protected area, the debugger displays an error message.

Note: Accessing Nonexistent Memory

If you're using the SWDS

The debugger won't allow you to access nonexistent memory. Attempting to access nonexistent program memory may actually cause the debugger to access data memory, and accessing nonexistent data memory may cause access to board control registers.

If you're using the simulator

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.

The debugger provides a complete set of memory-mapping commands. You can define the memory map interactively by entering these commands while you're using the debugger. This can be inconvenient because in most cases, you'll set up one memory map before you begin debugging and will use this map for all of your debugging sessions. The easiest method for doing this is to put the memory-mapping commands in a batch file.

Whenever you invoke the debugger, it looks for a batch file named `dbinit.cmd` (SWDS) or `siminit.cmd` (simulator). If it finds the file, the debugger automatically reads and executes the commands in the file. If you plan to use the same memory map many times, then it may be convenient for you to define your memory map in this batch file. However, you aren't required to use `dbinit.cmd` or `siminit.cmd` file. You can define the memory map in any batch file and load the file after you invoke the debugger.

The `dbinit.cmd` and `siminit.cmd` files shipped with the SWDS and simulator, respectively, define default memory maps for these tools. These default maps may be sufficient when you begin using the debugger. If you don't define the memory map in `dbinit.cmd` or `siminit.cmd`, then the debugger is initially unable

to access any target memory locations. Invalid memory addresses and their contents are highlighted in the data-display windows. (On color monitors, invalid memory locations, by default, are displayed in red.)

7.2 A Sample Memory Map

Whenever you invoke the debugger, it looks for an initialization file named dbinit.cmd (SWDS) or siminit.cmd (simulator). If the debugger finds this file, it reads commands from the file during the initialization process.

Because you must define a memory map before you can run any programs, it's convenient to define the memory map in the initialization file. The method for doing this differs between the simulator and the SWDS. Choose the correct method for the tool that you're using.

Defining a memory map for the simulator

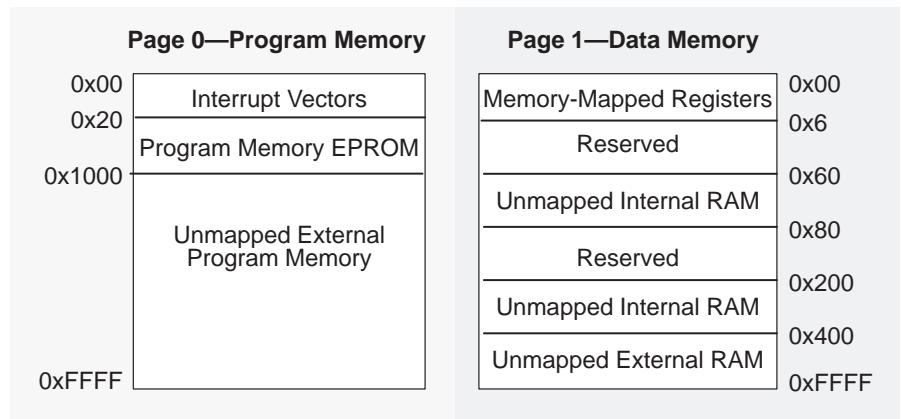
Figure 7–1 shows the memory map commands that are defined in siminit.cmd. You can use the file as is, edit it, or create your own command file.

Figure 7–1. Definition of On-Chip Memory Maps

```
MA 0x0, 0,0x20, ROM ;Interrupt vectors and reserved
MA 0x20, 0,0xF80,RAM ;On-chip program memory EPROM
MA 0x0, 1,0x6, RAM ;On-chip data memory-mapped registers
```

The MA (map add) commands define valid memory ranges and identify the read/write characteristics of the memory ranges. Figure 7–2 illustrates the memory map defined by the default siminit.cmd file.

Figure 7–2. Initial Memory Map Defined by the siminit.cmd File



Note: Memory Map Restrictions Associated With the Simulator

- 1) You *must* map in the first six words of data memory for on-chip memory-mapped registers. (Refer to the third line of code in Figure 7–1.)
- 2) *Do not* map in the following two reserved areas in data memory:

0x6 through 0x5F and
0x80 through 0x1FF

Figure 7–3 shows another example of a possible memory map for the simulator.

Figure 7–3. Example of a Memory Map That You Could Define for the Simulator

```
MA 0x0, 0, 0x20, ROM ;on-chip registers
MA 0x400, 1, 0x4000, RAM ;External data memory
```

Defining a memory map for the SWDS

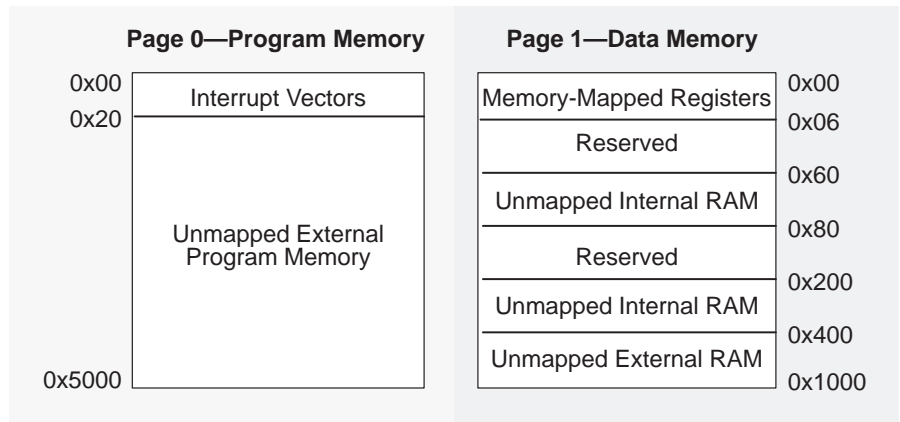
Figure 7–4 shows the memory map commands that are defined in dbinit.cmd. You can use the file as is, edit it, or create your own command file.

Figure 7–4. Definition of On-Chip Memory Maps

```
MA 0x0, 0, 0x20, RAM ;Interrupt vectors and reserved
MA 0x20, 0, 0x4FC0, RAM ;On-chip program memory EPROM
MA 0x0, 1, 0x6, RAM ;On-chip data memory-mapped registers
```

The MA (map add) commands define valid memory ranges and identify the read/write characteristics of the memory ranges. Figure 7–5 illustrates the memory map defined by the default dbinit.cmd file (configuration 0 from Table 7–1).

Figure 7-5. Initial Memory Map Defined by the dbinit.cmd File



The SWDS program and data memory space is limited to 24K words. You can define your own memory map in one of the four configurations shown in Table 7-1.

Table 7-1. Acceptable Memory Map Configurations for the SWDS

Configuration	Program Memory	Data Memory
0	0x0 to 0x5000	0x0 to 0x1000
1	0x0 to 0x4000	0x0 to 0x2000
2	0x0 to 0x3000	0x0 to 0x3000
3	0x0 to 0x2000	0x0 to 0x4000

Note: Memory Map Restrictions Associated with the SWDS

The total memory available in any configuration is 0x6000, and each memory space (program or data) begins at 0x0000. The SWDS configuration allows you to map over the normally reserved areas of the 'C2x; however, these reserved areas are still undefined. You should avoid these areas.

The SWDS memory configuration is determined from the *program* memory configuration. For example, if you defined the program memory upper limit to be somewhere between 0x4000 and 0x5000, the SWDS configuration will be set to 0. In this case, data memory addresses higher than 0x1000 will produce a memory error.

7.3 Identifying Usable Memory Ranges



ma The debugger's MA (memory add) command identifies valid ranges of target memory. The syntax for this command is:

ma *address, page, 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.

A new memory map must not overlap an existing entry. If you define a range that overlaps an existing range, the debugger ignores the new range and displays this error message in the COMMAND window display area: `Conflicting map range.`

- The *page* parameter is a 1-digit number that identifies the type of memory (program, data, or I/O) that the range occupies:

To identify this page,	Use this value as the <i>page</i> parameter
Program memory	0
Data memory	1
I/O space	2

- 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, ROM, or READONLY
Write-only memory	W, WOM, or WRITEONLY
Read/write memory	WR or RAM
No-access memory	PROTECT
Input port	IPOINT or IN PORT‡
Output port	OPOINT or OUT PORT‡
Input/output port	IOPORT‡

‡ Simulator only

7.4 Enabling Memory Mapping



map By default, mapping is enabled when you invoke the debugger. In some instances, you may want to explicitly enable or disable memory. You can use the MAP command to do this; the syntax for this command is:

map on
or **map off**

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

Note: Accessing Invalid Memory Locations

When memory mapping is enabled, you cannot:

- Access memory locations that are not defined by an MA command
- Modify 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 COMMAND window display area:

```
Error in expression
```

7.5 Checking the Memory Map



ml If you want to see which memory ranges are defined, use the ML command. The syntax for this command is:

ml

The ML command lists the page, starting address, ending address, and read/write characteristics of each defined memory range. For example, if you're using the SWDS default memory map and you enter the ML command, the debugger displays this:

Page	Memory range	Attributes
0	0000 - efff	READ WRITE
1	0000 - efff	READ WRITE

Page 0 = program memory
 Page 1 = data memory

| starting address ending address

7.6 Modifying the Memory Map During a Debugging Session



If you need to modify the memory map during a debugging session, use these commands.

md To delete a range of memory from the memory map, use the MD (memory delete) command. The syntax for this command is:

md *address, page*

- The *address* parameter identifies the starting address of the range of program, data, or I/O memory. If you supply an *address* that is not the starting address of a range, the debugger displays this error message in the COMMAND window display area:

```
Specified map not found
```

- The *page* parameter is a 1-digit number that identifies the type of memory (program, data, or I/O) that the range occupies:

To identify this page,	Use this value as the <i>page</i> parameter
Program memory	0
Data memory	1
I/O space	2

Note: Deleting a Simulated I/O Port

If you are using the simulator and want to use the MD command to remove a simulated I/O port, you must first disconnect the port with the MI command. Refer to Section 7.7, page 7-10.

mr If you want to delete all defined memory ranges from the memory map, use the MR (memory reset) command. The syntax for this command is:

mr

This resets the debugger memory map.

ma If you want to add a memory range to the memory map, use the MA (memory add) command. The syntax for this command is:

ma *address, page, length, type*

The MA command is described in detail on page 7-6.

Returning to the original memory map

If you modify the memory map, 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 TAKE command to read in your original memory map from a batch file.

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

```
mr  Reset the memory map  
take mem.map  Reread the default memory map
```

The MR command resets the memory map. (Note that you could put the MR command in the batch file, preceding the commands that define the memory map.) The TAKE command tells the debugger to execute commands from the specified batch file.

7.7 Simulating I/O Space

Both the SWDS and simulator provide simulation of I/O ports.

In addition to adding memory ranges to the memory map, you can use the MA command to add I/O ports to the memory map. To do this, use IPORT (input port), OPORT (output port), or IOPORT (input/output port) as the memory type. Use page 1 to simulate serial ports; use page 2 to simulate parallel ports. Then, you can use the MC command to connect a port to an input or output file. This 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.

Note: Simulating I/O Space With the SWDS

The SWDS cannot monitor the instruction execution while it's running; so, in addition to using the MC and MI commands, you *must* set a breakpoint on the IN and OUT instructions to enable the I/O simulation.

If the I/O port is connected to a file and a breakpoint is set on the corresponding I/O instruction, the debugger performs the I/O operation but *does not break* at that breakpoint. If you want to break at the I/O instruction while connected to a port, precede the I/O instruction with a NOP instruction, and set breakpoints on both instructions.

Connecting an I/O port



mc The MC (memory connect) command connects IPORT, OPORT, or IOPORT to an input or output file. The syntax for this command is:

mc *port address, page, filename, {READ | WRITE}*

- The *port address* parameter defines the address of the I/O port. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.
- The *page* parameter is a 1-digit number that identifies the page that the port occupies. Parallel ports are on page 2 (the I/O space), and serial ports are on page 1 (data space).
- The *filename* 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.
- The final parameter is specified as **READ** or **WRITE** and defines how the file will be used (for input or output, respectively).

The file is accessed during an IN or OUT instruction to the associated port address. 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 will read or write to the associated file.

Example 7–1 shows how an input port can be connected to an input file named in.dat.

Example 7–1. Connecting an Input Port to an Input File

Assume that the file in.dat contains words of data in hexadecimal format, one per line, like this:

```
0A00
1000
2000
.
.
.
```

These two debugger instructions set up and connect an input port:

MA	0x5, 2, 0x1, IPORT	<i>Configure port address 5h as an input port</i>
MC	0x5, 2, in.dat, READ	<i>Open file in.dat and connect to port address 5h</i>

Assume that these two 'C2x instructions are part of your 'C2x program. They read from the file in.dat.

IN	*, 5h	<i>IN instruction reads a word from the file attached to location 5h and stores it in data memory location indicated by the current auxiliary register.</i>
----	-------	---

Observing serial port data

The SWDS and simulator provide simulation for the serial port. The mode of operation depends primarily on three bits: FO, FSM, and TXM. You can connect the ports to input and output files with the MC command. You can observe the data moving in and out of the ports with the WA command or by watching locations 0 and 1 in data memory.

You can use four pseudoregisters to generate external synchronization pulses when you're operating with the FSM mode set to one: XIRP, XIRT, RIRP, RIRT. For more information about these pseudoregisters, refer to Table 7–2.

Table 7–2. Serial Port Pseudoregisters

Pseudo-Register	Description	Default
XIRP	<i>Transmit Interrupt Period Register</i> Defines the machine cycles between transmit interrupts. The period is loaded into the transmit interrupt timer register (XIRT) when the serial ports are reset or when the XIRT decrements to zero. Once the XIRT register counter reaches zero, data is transferred from the DXR register to the file which is connected to this register. A transmit interrupt is set in the interrupt flag register. If the interrupt is masked in the interrupt mask register, or if the global interrupt mode (INTM) is disabled, the transfer will not occur. Note that the minimum period value should be five. When both XIRT and XIRP are zero, any load to XIRP will also load XIRT. A write of zero to XIRP will disable the synchronization pulse generation after XIRT decrements to zero.	64
XIRT	<i>Transmit Interrupt Timer Register</i> Contains the current cycle count time to the next transmit interrupt. The counter decrements at the machine cycle rate.	Loaded from XIRP
RIRP	<i>Receive Interrupt Period Register</i> Defines the machine cycles between receive interrupts. The period is loaded into the receive interrupt timer register (RIRT) when the serial ports are reset or when the RIRT decrements to zero. Once the RIRT counter reaches zero, data is transferred from the file which is connected to the register to the data memory location. A receive interrupt is set in the interrupt flag register. If the interrupt is masked in the interrupt mask register, or if the global interrupt mode (INTM) is disabled, the transfer will not occur. Note that the minimum period value should be five. When both IRT and RIRP are zero, any load to RIRP will also load RIRT. A write of zero to RIRP will disable the synchronization pulse generation after RIRT decrements to zero.	64
RIRT	<i>Receive Interrupt Timer Register</i> Contains the current cycle count time to the next receive interrupt. The counter decrements at the machine cycle rate.	Loaded from XIRT

Configuring memory to use serial port simulation

If you want to use serial port simulation, you must configure memory with MA and MC. The code in Example 7–2 adds the transmit and receive registers to the memory map, then connects them to the input and output files.

Example 7–2. Adding Transmit and Receive Registers; Connecting Their Input and Output to a File

```

ma 0x0, 1, 1, IPORT    ;Configure DRR in data space as input port
ma 0x1, 1, 1, OPORT    ;Configure DXR in data space as output port
mc 0x0, 1, rdat, read  ;Open file rdat and connect to port address 0h
mc 0x1, 1, xdat, write ;Open file xdat and connect to port address 1h

```

The following commands configure the period registers for the transmit and receive operations to occur every 64 cycles for the standard serial port:

```

?rirp=64
?xirp=64

```

The input and output file formats for the standard serial port operation require one hexadecimal number per line. The following is an acceptable format for an input file to the standard serial port:

```

0000
a445
099f
.
.
.

```

Disconnecting an I/O port

Before you can use the MD command to delete a port from the memory map, you must use the MI command to disconnect the port.



mi The MI (memory disconnect) command disconnects a file from an I/O port. The syntax for this command is:

```
mi port address, page, {READ | WRITE}
```

The *port address* and *page* identify the port that will be closed. The read/write characteristics must match the parameter used when the port was connected.

Loading, Displaying, and Running 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. Note that many of the commands described in this chapter can also be executed from the Load pulldown menu.

Topic	Page
<i>Depending on the debugging mode you choose, the debugger shows you assembly language only, C code only, or both at the same time. You can also tell the debugger to automatically display whatever type of code is currently running.</i>	8.1 Code-Display Windows: Viewing Assembly Language Code, C Code, or Both 8-3
	Selecting a debugging mode 8-4
<i>To debug a program, you must load the program's object code into memory. You'll also need to see the associated source code.</i>	8.2 Displaying Your Source Programs (or Other Text Files) 8-5
	Displaying assembly language code 8-5
	Displaying C code 8-7
	Displaying other text files 8-8
	8.3 Loading Object Code 8-9
	Loading code while invoking the debugger 8-9
	Loading code after invoking the debugger 8-9
8.4 Where the Debugger Looks for Source Files 8-10	

Once you've loaded an object file, there are several ways of running the program during a debugging session.

8.5	Running Your Programs	8-11
	Defining the starting point for program execution	8-11
	Running code	8-12
	Single-stepping through code	8-13
	Running code while disconnected from the target system	8-15
	Running code conditionally	8-16
8.6	Halting Program Execution	8-17
8.7	Benchmarking	8-18
	Benchmarking with the simulator	8-18
	Benchmarking with the SWDS	8-19

8.1 Code-Display Windows: Viewing Assembly Language Code, C Code, or Both

The debugger has three code-display windows:

- The DISASSEMBLY window displays the reverse assembly of program memory contents.
- The FILE window displays any text file; its main purpose is to display C source files.
- The CALLS window identifies the current function (when C code is running).

You can view code in several different ways. The debugger has three different code displays that are associated with the three debugging modes. The debugger's selection of the appropriate display is based on two factors:

- The mode you select and
- Whether your program is currently executing assembly language code or C code.

Here's a summary of the modes and displays; for a complete description of the three debugging modes, refer to Section 5.1, *Debugging Modes and Default Displays* (page 5-2).

Use this mode	To view	The debugger uses these code-display windows
assembly mode	<i>assembly language code only</i> (even if your program is executing C code)	<input type="checkbox"/> DISASSEMBLY
auto mode	<i>assembly language code</i> (when that's what your program is running)	<input type="checkbox"/> DISASSEMBLY
auto mode	<i>C code only</i> (when that's what your program is running)	<input type="checkbox"/> FILE <input type="checkbox"/> CALLS
mixed mode	<i>both assembly language and C code</i>	<input type="checkbox"/> DISASSEMBLY <input type="checkbox"/> FILE <input type="checkbox"/> CALLS

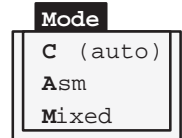
You can switch freely between the modes. If you choose auto mode, then the debugger displays C code *or* assembly language code, depending on the type of code that is currently executing.

Selecting a debugging mode

When you first invoke the debugger, it automatically comes up in auto mode. You can then choose assembly or mixed mode. There are several ways to do this.



The Mode pulldown menu provides an easy method for switching modes. There are several ways to use the pulldown menus; here's one method:

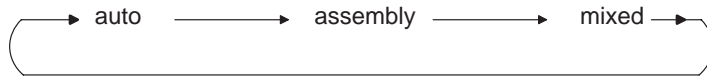


- 1) Point to the menu name.
- 2) Press the left mouse button; do not release the button. Move the mouse down the menu until your choice is highlighted.
- 3) Release the mouse button.

For more information about the pulldown menus, refer to Section 6.2, *Using the Pulldown Menus*, on page 6-6.



F3 Pressing this key causes the debugger to switch modes in this order:



Enter any of these commands to switch to the desired debugging mode:

- c** Changes from the current mode to auto mode.
- asm** Changes from the current mode to assembly mode.
- mix** Changes from the current mode to mixed mode.

If you are already in the desired mode when you enter a mode command, then the command has no effect.

8.2 Displaying Your Source Programs (or Other Text Files)

The debugger displays two types of code:

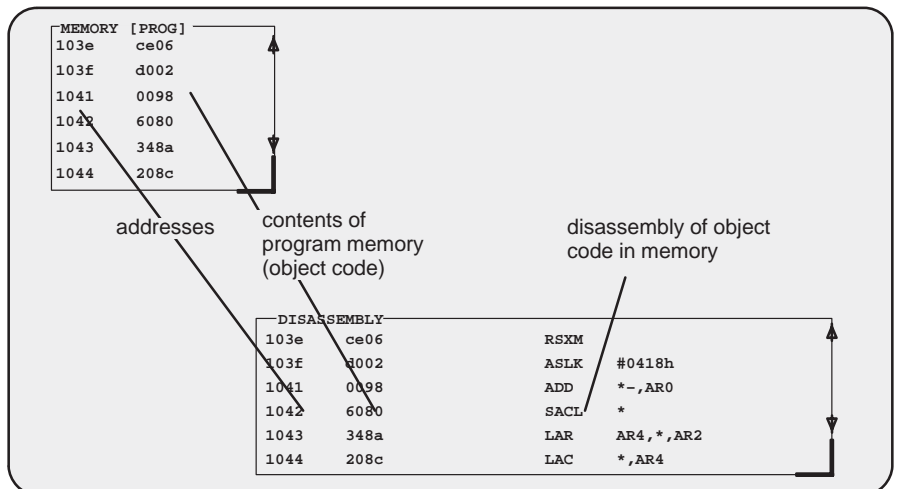
- It displays **assembly language code** in the DISASSEMBLY window in auto, assembly, or mixed mode.
- It displays **C code** in the FILE window in auto and mixed modes.

The DISASSEMBLY and FILE windows are primarily intended for displaying code that the PC points to. By default, the FILE window displays the C source for the current function (if any), and the DISASSEMBLY window shows the current disassembly.

Sometimes it's useful to display other files or different parts of the same file; for example, you may want to set a breakpoint at an undisplayed line. The DISASSEMBLY and FILE windows are not large enough to show the entire contents of most assembly language and C files. You can scroll through the windows. You can also tell the debugger to display specific portions of the disassembly or C source.

Displaying assembly language code

The assembly language code in the DISASSEMBLY window is the reverse assembly of program-memory contents. (This code doesn't 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, then the DISASSEMBLY window displays the reverse assembly of the object file that's loaded into memory. If you don't load an object file, the DISASSEMBLY window shows the reverse assembly of whatever happens to be in memory.



In assembly and mixed modes, you can use these commands to display a different portion of code in the DISASSEMBLY window.

dasm Use the DASM command to display code beginning at a specific point. The syntax for this command is:

dasm *address*
or **dasm** *function name*

This command modifies the display so that *address* or *function name* is displayed within the DISASSEMBLY window. The debugger continues to display this portion of the code until you run a program and halt it.

addr Use the ADDR command to display assembly language code beginning at a specific point. The syntax for this command is:

addr *address*
or **addr** *function name*

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 mixed mode, ADDR affects both the DISASSEMBLY and FILE windows.

Displaying C code

Unlike assembly language code, C code isn't 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 entering a FILE, FUNC, or ADDR command.
- In auto and mixed modes, the debugger automatically opens a FILE window if you're currently running C code.



These commands are valid in C and mixed modes:

file Use the FILE command to display the contents of any text file. The syntax for this command is:

file *filename*

This uses the FILE window to display the contents of *filename*. The debugger continues to display this file until you run a program and halt in a C function. Although this command is most useful for viewing C code, you can use the FILE command for displaying any text file. You can view only one text file at a time. Note that you can also access this command from the Load pulldown menu.

(Displaying a file *doesn't* 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 8.3 on page 8-9.)

func Use the FUNC command to display a specific C function. The syntax for this command is:

func *function name*

or **func** *address*

FUNC modifies the display so that *function name* or *address* is displayed within the 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.

Note that FUNC works similarly to FILE, but you don't need to identify the name of the file that contains the function.

addr Use the ADDR command to display C or assembly code beginning at a specific point. The syntax for this command is:

addr *address*
or **addr** *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.



Whenever the CALLS window is open, you can use the mouse or function keys 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.

- 1) In the CALLS window, point to the name of the C function.
- 2) Click the left mouse button.

(If the CALLS window is active, you can also use the arrow keys and **F9** to display the function; see the *CALLS window* discussion on page 5-9 for details.)

Displaying other text files

The DISASSEMBLY window always displays the reverse assembly of memory contents, no matter what is in memory.

The FILE window is primarily for displaying C code, but you can use the FILE command to display any text file within the FILE window. You may, for example, wish 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.

You are restricted to displaying files that are 65,518 bytes long or less.

8.3 Loading Object Code

In order to debug a program, you must load the program's object code into memory. You can do this as you're invoking the debugger, or you can do it after you've invoked the debugger. (Note that you create an object file by compiling, assembling, and linking your source files; see Section 4.3, *Preparing Your Program for Debugging*, on page 4-8.)

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 debugger's LOAD command). To do this, enter the appropriate command along with the name of the object file.

If you want to load a file's symbol table only, use the `-s` option (this has the same effect as using the debugger's SLOAD command). To do this, enter the appropriate command, the name of the object file, and specify `-s`.

Loading code after invoking the debugger

After you invoke the debugger, you can use one of three commands to load object code and/or the symbol table associated with an object file. Use these commands as described below, or use them from the Load pulldown menu.

load Use the LOAD command to load both an object file and its associated symbol table. In effect, the LOAD command performs both a RELOAD and an SLOAD. The format for this command is:

load *object filename*

reload Use the RELOAD command to load only an object file *without* loading its associated symbol table. This is useful for reloading a program when memory has been corrupted. The format for this command is:

reload *object filename*

sload Use the SLOAD command to load only a symbol table. The format for this command is:

sload *object filename*

SLOAD is useful in a debugging environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). SLOAD clears the existing symbol table before loading the new one but does not modify memory or set the program entry point.

8.4 Where the Debugger Looks for Source Files

Some commands (FILE, LOAD, RELOAD, and SLOAD) 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 don't supply path information, though, the debugger must search for the file. The debugger first looks for these files in the current directory. You may, however, have your files in several different directories.

□ If you're using LOAD, RELOAD, or SLOAD, you have only two choices for supplying the path information:

■ Specify the path as part of the filename.

cd Alternatively, you can use the CD command to change the current directory from within the debugger. The format for this command is:

cd *directory name*

□ If you're using the FILE command, you have several options:

■ Within the DOS environment, you can name additional directories with the D_SRC environment variable. The format for doing this is:

SET D_SRC=C:\pathname;C:\pathname

This allows you to name several directories that the debugger can search. If you use the same directories often, it may be convenient to set the D_SRC environment variable in your autoexec.bat file. If you do this, then the list of directories is always available when you're using the debugger.

■ 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.

use 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.

8.5 Running Your Programs

To debug your programs, you must execute them on one of the two TMS320C2x debugging tools (SWDS, 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. There are several ways to halt execution:
 - Set a breakpoint.
 - When you issue a run command, define a specific stopping point.
 - Press `(ESC)`.
 - Press the left mouse button.
- Single-step** commands execute assembly language or C code, one statement at a time, and update the display after each execution.

Defining the starting point for program execution

All run and single-step commands begin executing from the current PC (program counter). 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
- or**
- Finding the appropriately highlighted line in the FILE or DISASSEMBLY window. You can do this by executing 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. There are two ways to do this:

rest If you executed some code and would like to rerun the program from the original program entry point, use the RESTART (REST) command. The format for this command is:

restart

or **rest**

Note that you can also access this command from the Load pulldown menu.

?/eval You can directly modify the PC's contents with one of these commands:

?PC=new value

or **eval pc = new value**

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

Running code

The debugger supports several run commands.



run The RUN command is the basic command for running an entire program. The format for this command is:

run [*expression*]

The command's behavior depends on the type of parameter you supply:

- If you don't supply an *expression*, the program executes until it encounters a breakpoint or until you press **ESC** or the left mouse button.
- If you supply a logical or relational *expression*, this becomes a conditional run (see page 8-16).
- 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.



F5 Pressing this key runs code from the current PC. This is similar to entering a RUN command without an *expression* parameter.



go Use the GO command to execute code up to a specific point in your program. The format for this command is:

go [*address*]

If you don't supply an *address* parameter, then GO acts like a RUN command without an *expression* parameter.

ret The RETURN (RET) command executes the code in the current C function and halts when execution returns to its caller. The format for this command is:

return

or **ret**

Breakpoints do not affect this command, but you can halt execution by pressing **ESC** or the left mouse button.

Simulator

runb Use the RUNB (run benchmark) command to execute a specific section of code and count the number of clock cycles consumed by the execution. The format for this command is:

runb

Using the RUNB command to benchmark code is a multistep process, described later in this chapter (Section 8.7, *Benchmarking*, on page 8-18).

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 more than one statement; the debugger updates the display after each statement.) You can single-step through assembly language code or C code.

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

Note that the debugger ignores interrupts when you use the STEP command to single-step through assembly language code.



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

- If you don't 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 page 8-16).
- If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger single-steps *count* C or assembly language statements (depending on the type of code you're in).

step Use the STEP command to single-step through assembly language or C code. The format for this command is:

step [*expression*]

If you're 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're 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` debug option). When function execution completes, single-step execution returns to the caller. If the function wasn't compiled with the debug option, the debugger executes the function but doesn't show single-step execution of the function.

cstep The CSTEP command is similar to STEP, but CSTEP always single-steps in terms of a C statement. If you're in C code, STEP and CSTEP behave identically. In assembly language code, however, CSTEP executes all assembly language statements associated with one C statement before it updates the display. The format for this command is:

cstep [*expression*]

next The NEXT and CNEXT commands are similar to the STEP and CSTEP commands. The only difference is that NEXT/CNEXT never show single-step execution of called functions—they always step to the next consecutive statement. The formats for these commands are:



cnext

next [*expression*]

cnext [*expression*]





You can also single-step through programs by using function keys:

-  Acts as a STEP command.
-  Acts as a NEXT command.





The debugger allows you to execute several single-step commands from the selections on the menu bar.

To execute a STEP:

-  1) Point to `Step=F8` in the menu bar.
-  2) Press and release the left mouse button.

To execute a NEXT:

-  1) Point to `Next=F10` in the menu bar.
-  2) Press and release the left mouse button.

Running code while disconnected from the target system

SWDS

runf Use the RUNF command to disconnect the SWDS from the target system while code is executing. The format for this command is:

runf

When you enter RUNF, the debugger clears all breakpoints, disconnects the SWDS 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 will produce an error.

RUNF is useful in a multiprocessor system. It's also useful in a system in which several target systems share an SWDS; RUNF enables you to disconnect the SWDS from one system and connect it to another.

halt Use the HALT command to halt the target system after you've entered a RUNF command. The format for this command is:

halt

When you invoke the debugger, it automatically executes a HALT command. Thus, if you enter a RUNF, quit the debugger, and later reinvoke the debugger, you will effectively reconnect the SWDS 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.

reset The RESET command resets the target system. This is a *software* reset. The format for this command is:

reset

Running code conditionally

The RUN, GO, and single-step commands have an optional *expression* parameter that can be a relational or logical expression. This type of expression has 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 breakpoints with conditional runs; the expression is evaluated each time the debugger encounters a breakpoint. Each time the debugger evaluates the conditional expression, it updates the screen. The debugger applies this algorithm:

top:

if (*expression* == 0) go to end;

run or single-step (until breakpoint, `ESC`, or mouse button halts execution)

if (halted by breakpoint, *not* by `ESC` or mouse button) go to top

end:

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

8.6 Halting Program Execution

Whenever you're 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*). If you'd like to explicitly halt program execution, there are two ways to accomplish this:



Click the left mouse button.



Press the escape key.

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

8.7 Benchmarking

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

Benchmarking with the simulator

The debugger maintains the count in a pseudoregister named *CLK*.

Benchmarking code is a multiple-step process:

- Step 1:** Set a breakpoint at the statement that marks the beginning of the section of code you'd like to benchmark.
- Step 2:** Set a breakpoint at the statement that marks the end of the section of code you'd like to benchmark.
- Step 3:** Enter any RUN command to execute code up to the first breakpoint.
- Step 4:** Now enter the RUNB command:

```
runb 
```

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 WA command. This value is valid until you enter another RUN command.

Note: Restrictions Associated With CLK

- The value in CLK is valid only after using a RUNB command that is terminated by a breakpoint.
 - The value returned by CLK after executing a RUNB command would be higher than the actual number of cycles because the pipeline reduces the number of execution cycles for each instruction, and the simulator is an instruction-level simulator.
-

Benchmarking with the SWDS

Note:

If you're using serial ports, this information does not apply.

The debugger maintains the cycle count in the timer period register (TIM).

Usually, when you invoke the debugger, the TIM is set to 0. This value corresponds to the period of CLKIN/4 which is the same as the processor's cycle time.

TIM is shown in the CPU window. You can also view it (in integer format) in the WATCH window by entering this command:

```
wa tim 
```


Managing Data

The debugger allows you to examine and modify many different types of data related to the 'C2x and to your program. You can display and modify the values of :

Individual memory locations or a range of memory

'C2x registers

Variables, including scalar types (ints, chars, etc.) and aggregate types (arrays, structures, etc.)

This chapter tells you how to display and change data.

Topic	Page
<i>The chapter begins by describing basic commands and editing methods that apply to managing all forms of data.</i>	9.1 Where Data Is Displayed 9-2
	9.2 Basic Commands for Managing Data 9-2
	9.3 Basic Methods for Changing Data Values 9-4
	Editing data displayed in a window 9-4 Advanced “editing”—using expressions with side effects 9-5
<i>These sections discuss unique details about displaying and changing specific types of data.</i>	9.4 Managing Data in Memory 9-6
	Displaying memory contents 9-6
	Displaying program memory 9-7
	Displaying memory contents while you’re debugging C 9-8
	Saving memory values in a file 9-9
	Filling a block of memory 9-10
	9.5 Managing Register Data 9-11
	Displaying register contents 9-11
	9.6 Managing Data in a DISP (Display) Window 9-13
	Displaying data in a DISP window 9-13 Closing a DISP window 9-15
9.7 Managing Data in a WATCH Window 9-16	
Displaying data in a WATCH window 9-16 Deleting watched values 9-17 and closing the WATCH window	
<i>If you are using the simulator, you can also observe the status of the pipeline and of the BIO pin.</i>	9.8 Managing Signal Information (Simulator Only) 9-18
	Monitoring the BIO pin 9-18

9.1 Where Data Is Displayed

Four windows are dedicated to displaying the various types of data.

Type of data	Window name and purpose
memory locations	MEMORY window Displays the contents of a range of data memory or program memory
register values	CPU window Displays the contents of 'C2x registers
pointer data or selected variables of an aggregate type	DISP windows Display the contents of aggregate types and shows the values of individual members
selected variables (scalar types or individual members of aggregate types) and specific memory locations or registers	WATCH window Displays selected data

This group of windows is referred to as **data-display windows**.

9.2 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.



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

whatis *symbol*

This lists *symbol's* data type in the COMMAND window display area. The *symbol* can be any variable (local, global, or static), a function name, structure tag, typedef name, or enumeration constant.

Command	Result displayed in the COMMAND window
whatis <code>big1</code>	<code>struct zzz big1;</code>
whatis <code>xxx</code>	<code>struct xxx { int a; int b; int c; int f1 : 2; int f2 : 4; struct xxx *f3; int f4[10]; }</code>

- ? The ? (evaluate expression) command evaluates an expression and shows the result in the COMMAND window display area. 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, then the debugger displays the result as a decimal value in the COMMAND window. If *expression* is a structure or array, ? 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
? big1	big1[0].b1 4365 big1[0].b2 -7910 big1[0].b3 1952 big1[0].b4 -1555 etc.
? j	4194425
? j=0x5a	90

Note that the DISP command (described in detail on page 9-13) behaves like the ? command when its *expression* parameter does not identify an aggregate type.

- eval** The EVAL (evaluate expression) command behaves like the ? command *but does not show the result* in the COMMAND window display area. 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's not necessary to display the result).

9.3 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 the MEMORY window
- Elements displayed in a DISP window
- Values displayed in the WATCH window






There are two similar methods for overwriting displayed data:



This method is sometimes referred to as the “click and type” method.

-  1) Point to the data item that you want to modify.
-  2) Click the left button. The debugger highlights the selected field. (Note that the window containing this field becomes active when you press the mouse button.)
-  3) Type the new information. If you make a mistake or change your mind, press  or move the mouse outside the field and press/release the left button; this resets the field to its original value.
-  4) When you finish typing the new information, press  or any arrow key. This replaces the original value with the new value.



-
- 1) Select the window that contains the field you'd like to modify; make this the active window. (Use the mouse, the WIN command, or . For more detail, see Section 5.4, *The Active Window*, on page 5-17.)
 - 2) Use arrow keys to move the cursor to the field you'd like to edit.
 -  Moves up 1 field at a time.
 -  Moves down 1 field at a time.
 -  Moves left 1 field at a time.
 -  Moves right 1 field at a time.

- 3) When the field you'd like to edit is highlighted, press `F9`. The debugger highlights the field that the cursor is pointing to.
- 4) Type the new information. If you make a mistake or change your mind, press `ESC`; this resets the field to its original value.
- 5) When you finish typing the new information, press `↵` or any arrow key. This replaces the original value with the new value.

Advanced “editing”—using expressions with side effects

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

This means that you can coerce many commands into changing values for you. Specifically, it's most helpful to use `?` and `EVAL` to change data as well as display it.

For example, if you want to see what's in auxiliary register AR3, you can enter:

```
? AR3
```

You can also use this type of command to modify AR3's contents. Here are some examples of how you might do this:

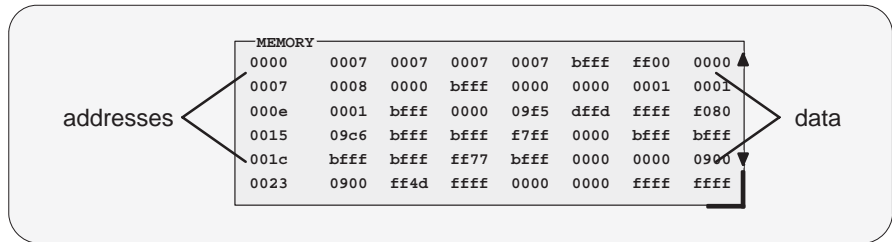
<code>? AR3++</code>	<i>Side effect: increments the contents of AR3 by 1</i>
<code>eval --AR3</code>	<i>Side effect: decrements the contents of AR3 by 1</i>
<code>? AR3 = 8</code>	<i>Side effect: sets AR3 to 8</i>
<code>eval AR3/=2</code>	<i>Side effect: divides contents of AR3 by 2</i>

Note that not all expressions have side effects. For example, if you enter `? AR3+4`, the debugger displays the result of adding 4 to the contents of AR3 but does not modify AR3'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:

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

9.4 Managing Data in Memory

In mixed and assembly modes, the debugger maintains a MEMORY window that displays the contents of memory. For details concerning the MEMORY window, see the *MEMORY window* discussion (page 5-11).



By default, the MEMORY window displays data memory. The debugger has commands that show the value at a specific location, display a different range, or display program memory instead of data memory. The debugger allows you to change the values at individual locations; refer to Section 9.3, *Basic Methods for Changing Data Values* (page 9-4), for more information.

Displaying memory contents

The main way to observe memory contents is to view the display in the MEMORY window. The amount of memory that you can display is limited by the size of the MEMORY window (which is limited only by the screen size). During a debugging session, you may need to display different areas of memory within the window. The debugger provides two methods for doing this.



mem If you want to display a different memory range in the MEMORY window, use the MEM command. The basic syntax for this command is:

mem *expression*

This makes *expression* 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 (see *Resizing a window*, page 5-20, for more information).

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

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

```
mem 0x00
```

Hint: 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
```

Hint: 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 - AR0 + label
```



You can also change the display of any data-display window—including the MEMORY window—by scrolling through the window's contents. See the *Scrolling through a window's contents* discussion (page 5-25) for more details.

Displaying program memory

By default, the MEMORY window displays data memory, but you can also display program memory. To do this, follow any address parameter with **@prog**; for example, you can follow the MEM command's *expression* parameter with @prog. This suffix tells the debugger that the *expression* parameter identifies a program memory address instead of a data memory address.

If you display program memory in the MEMORY window, the debugger changes the window's label to MEMORY [PROG] so that there is no confusion about what type of memory is displayed at any given time.

Any of the examples presented in this section could be modified to display program memory:

```
mem 0x00@prog
mem &SYM@prog
mem (SP - AR0 + label)@prog
? *0x26@prog
wa *0x26@prog
disp *(float *)0x26@prog
```

You can also use the suffix `@data` to display data memory; however, since data memory is the default, the `@data` suffix is unnecessary.

Displaying memory contents while you're debugging C

If you're debugging C code in auto mode, you won't see a MEMORY window—the debugger doesn't 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 data memory location 26 (hex), you could enter:

```
? *0x26
```

The debugger displays the memory value in the COMMAND window display area.

- If you want the opportunity to observe a specific memory location over a longer period of time, you can display it in a WATCH window. Use the `WA` command to do this:

```
wa *0x26
```

- You can also use the `DISP` command to display memory contents. The `DISP` 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 in a file



ms Sometimes it's useful to save a block of memory values to a file. You can use the MS (memory save) command to do this; the files are saved in COFF format. The syntax for the MS command is:

ms *address, page, length, filename*

- The *address* parameter identifies the first address in the block.
- The *page* is a 1-digit number that identifies the type of memory (program, data, or I/O) to save:

To save this type of memory	Use this value as the <i>page</i> parameter
Program memory	0
Data memory	1

- The *length* parameter defines the length, in words, of the range. This parameter can be any C expression.
- The *filename* is a system file.

For example, to save the values in data memory locations 0x0–0x10 to a file named memsave, you could enter:

```
ms 0x0,1,0x10,memsave
```

To reload memory values that were saved in a file, use the LOAD command. For example, to reload the values that were stored in memsave, enter:

```
load memsave
```

Filling a block of memory



fill Sometimes it's useful to be able to fill an entire block of memory at once. You can do this by using the FILL command. The syntax for this command is:

fill *address* [*@ prog rr @ data*], *page*, *length*, *data*

- The *address* parameter identifies the first address in the block.
- The *length* parameter defines the number of words to fill.
- The *page* is a 1-digit number that identifies the type of memory (program, data, or I/O) to fill:

To fill this type of memory	Use this value as the <i>page</i> parameter
Program memory	0
Data memory	1

- The *data* parameter is the value that is placed in each word in the block.

For example, to fill program memory locations 0x10FF–0x110D with the value 0xABCD, you would enter:

```
fill 0x10ff @ data,0,0xf,0xabcd
```

If you want to check to see that memory has been filled as you have asked, you can enter:

```
mem 0x10ff@data
```

This changes the MEMORY window display to show the block of memory beginning at data memory address 0x10FF.

The FILL command can also be executed from the Memory pulldown menu.

Note that the syntax for the fill command can be simplified as *fill addr, page, length, data* if the first address in the block is not a reserved memory location in either program or data memory space:

```
fill 0 @ prog,0,0x100,0xabcd
```

9.5 Managing Register Data

In mixed and assembly modes, the debugger maintains a CPU window that displays the contents of individual registers. For details concerning the CPU window, see the *CPU window* discussion (page 5-13).

CPU							
ACC	00000000	PREG	00000000				
TIM	ffff	PRD	ffff				
PC	1124	TOS	0000	ST0	0600	ST1	07E0
IMR	ffc0	IFR	0000	TREG	0000	RTPC	0000
AR0	0000	AR1	0000	AR2	0000	AR3	0000
AR4	0000	AR5	04e5	AR6	04e6	AR7	0000
BIO	0001						

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. Refer to Section 9.3, *Basic Methods for Changing Data Values* (page 9-4), for more information.

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're 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 display. In this type of situation, there are several ways to observe the contents of the selected registers.

- 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 the PC, you could enter:

```
? PC
```

The debugger displays the SP's current contents in the COMMAND window display area.

- If you want to observe a register over a longer period of time, you can use the `WA` command to display the register in a WATCH window. For example, if you want to observe the status register, you could enter:

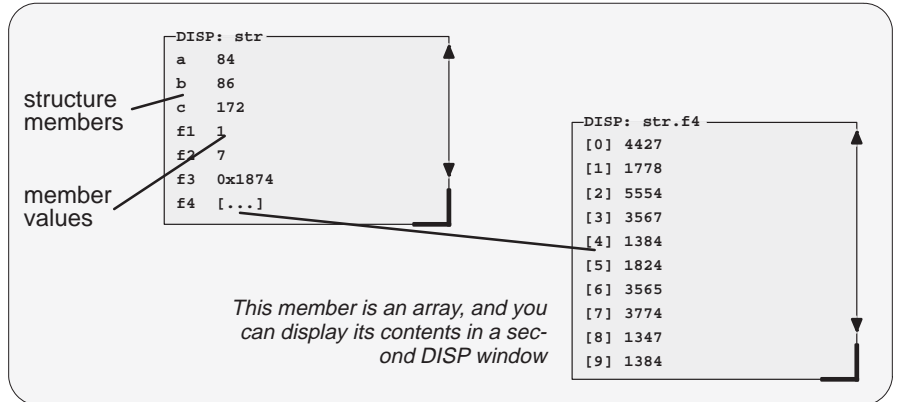
```
WA ST,Status Reg
```

This adds the `ST` to the WATCH window and labels it as *Status Reg*. The register's contents are continuously updated, just as if you were observing the register in the CPU window.

These methods are also useful when you're debugging C in auto mode because the debugger doesn't show the CPU window in the C-only display. For a list of all registers and pseudoregisters that you can display, see Appendix C.

9.6 Managing Data in a DISP (Display) Window

The main purpose of the DISP window is to display members of complex, aggregate data types such as arrays and structures. The debugger shows DISP windows *only when you specifically request to see DISP windows* with the DISP command (described below). Note that you can have up to 120 DISP windows open at once. For additional details about DISP windows, see the *DISP window* discussion (page 5-14).



Remember, you can use the data-management commands or the debugger's overwrite editing capability to modify the contents of any value displayed in a DISP window. Refer to Section 9.3, *Basic Methods for Changing Data Values* (page 9-4), for more information.

Displaying data in a DISP window



disp To open a DISP window, use the DISP command. Its syntax is:

disp *expression*

If the *expression* is not an array, structure, or pointer (of the form **pointer name*), the DISP command behaves like the ? command. However, if *expression* is one of these types, the debugger opens a DISP window to display the values of the members.

If a DISP window contains a long list of members, you can use **(PAGE DOWN)**, **(PAGE UP)**, or arrow keys to scroll through the window. If the window contains an array of structures, you can use **(CONTROL) (PAGE DOWN)** and **(CONTROL) (PAGE UP)** to scroll through the array.

Once you open a DISP window, you may find that a displayed member is another one of these types. This is how you identify the members that are arrays, structures, or pointers:

A member that is an array looks like this [. . .]
 A member that is a structure looks like this {. . .}
 A member that is a pointer looks like an address 0x00000000

You can display the additional data (the data pointed to or the members of the array or structure) in additional DISP windows (these are referred to as *children*). There are three ways to do this.



Use the DISP command again; this time, *expression* must identify the member that has additional data. For example, if the first expression identifies a structure named *str* and one of *str*'s members is an array named *f4*, you can display the contents of the array by entering this command:

```
disp str.f4
```

This opens a new DISP window that shows the contents of the array. If *str* has a member named *f3* that is a pointer, you could enter:

```
disp *str.f3
```

This opens a window to display what *str.f3* points to.



Here's another method of displaying the additional data:

- 1) Point to the member in the DISP window.
- 2) Now click the left button.



Here's the third method:

- 1) Use the arrow keys to move the cursor up and down in the list of members.
- 2) When the cursor is on the desired field, press **F9**.

When the debugger opens a second DISP window, the new window may at first be displayed on top of the original DISP window—if so, you can move the windows so that you can see both at once. If the new windows also have members that are pointers or aggregate types, you can continue to open new DISP windows.

Closing a DISP window

Closing a DISP window is a simple, two-step process.

Step 1: Make the DISP window that you want to close active (see Section 5.4, *The Active Window*, on page 5-17).

Step 2: Press **F4**.

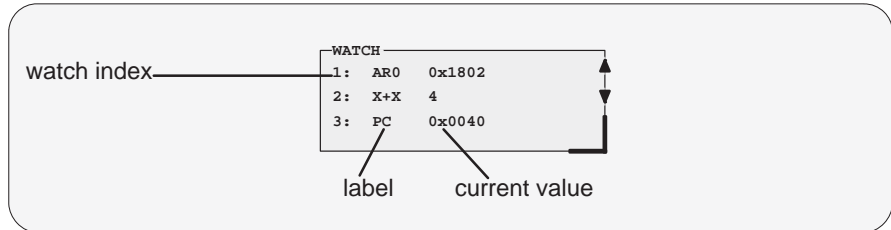
Note that you can close a window and all of its children by closing the original window.

Note: Effects of LOAD and SLOAD on DISP Windows

The debugger automatically closes any DISP windows when you execute a LOAD or SLOAD command.

9.7 Managing Data in a WATCH Window

The debugger doesn't 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 wouldn't be useful. Instead, the debugger allows you to create a WATCH window that shows you how program execution affects specific expressions, variables, registers, or memory locations.

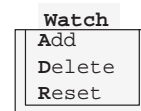


The debugger displays a WATCH window *only when you specifically request a WATCH window* with the WA command (described below). Note that there is only one WATCH window. For additional details concerning the WATCH window, see the *WATCH window* discussion (page 5-15).

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. Refer to Section 9.3, *Basic Methods for Changing Data Values* (page 9-4), for more information.

Note: Alternative Method for Entering WATCH Commands

All of the watch commands described here can also be accessed from the Watch pulldown menu. For more information about using the pulldown menus, refer to Section 6.2, *Using the Menu Bar and the Pulldown Menus* (page 6-6).



Displaying data in the WATCH window

The debugger has one command for adding items to the WATCH window.



wa To open the WATCH window, use the WA (watch add) command. The basic syntax is:

wa *expression* [, *label*]

When you first execute WA, the debugger opens the WATCH window. After that, executing WA adds additional values to the WATCH window.

The *expression* parameter can be any C expression, including an expression that has side effects. It's most useful to watch an expression whose value will change over time; constant expressions provide no useful function in the watch window.

The *label* parameter is optional. When used, it provides a label for the watched entry. If you don't use a *label*, the debugger displays the *expression* in the label field.

Deleting watched values and closing the WATCH window

The debugger supports two commands for deleting items from the WATCH window.



wr If you'd like to close the WATCH window and delete all of the items in a single step, use the WR (watch reset) command. The syntax is:

wr

wd If you'd like to delete a specific item from the WATCH window, use the WD (watch delete) command. The syntax is:

wd *index number*

Whenever you add an item to the WATCH window, the debugger assigns it an index number. (The illustration of the WATCH window on page 9-16 points to these watch indexes.) The WD command's *index number* parameter must correspond to one of the watch indexes in the WATCH window.

Note that deleting an item (depending on where it is in the list) causes the remaining index numbers to be reassigned. Deleting the last remaining item in the WATCH window closes the WATCH window.

Note: Effects of LOAD and SLOAD on WATCH Windows


The debugger automatically closes the WATCH window when you execute a LOAD or SLOAD command.

9.8 Managing Signal Information (Simulator Only)

The simulator supports an additional feature that allows you to monitor the $\overline{\text{BIO}}$ pin. For this feature, the simulator supports a pseudoregister that you can query with ? or DISP or add to the WATCH window.

Monitoring the $\overline{\text{BIO}}$ pin

The $\overline{\text{BIO}}$ pin, which is associated with 'C2x conditional instructions, is simulated as the BIO pseudoregister. You can query the value of BIO and also change it; for example, to set BIO low, you would enter:

```
?BIO = 0 
```

After a reset command, BIO is set to 1.

Using 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 **breakpoints** at critical points in your code. You can set breakpoints in assembly language code and in C code. A breakpoint halts any program execution, whether you're running or single-stepping through code.

Breakpoints are especially useful in combination with conditional execution (described on page 8-16) and benchmarking (simulator only; described on page 8-18).

Note that the commands described in this chapter can also be executed from the Break pulldown menu.

Topic	Page
<i>This chapter describes the simple processes of setting and clearing software breakpoints and of obtaining a listing of all the breakpoints that are set.</i>	
10.1 Setting a Breakpoint	10-2
10.2 Clearing a Breakpoint	10-4
10.3 Finding the Breakpoints That Are Set	10-5

10.1 Setting a Breakpoint

When you set a breakpoint, the debugger highlights the breakpointed line by showing it in a heavier or brighter font (this is the default behavior—you can change this behavior with the screen-customization commands), and adds a BP > label to the beginning of the line. If you set a breakpoint in the disassembly, the debugger also highlights the associated C statement. 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; notice how the line is highlighted. Breakpoints are also set at the associated assembly language statement—it's highlighted, too.

```

FILE: sample.c
00044
00045 BP> meminit();
00046   for (i=0; i < 0x50000; i++)
00047   {
00048   call(i);

```

```

DISASSEMBLY
00fc 7aa0 meminit: POPD  *+
00fe 70a0          SARAR0,*+
0100 7180          SARAR1,*

```

Notes: Restrictions Associated With Breakpoints

- After execution is halted by a breakpoint, you can continue program execution by reissuing any of the run or single-step commands.
- Up to 200 breakpoints can be set.
- If you're using the SWDS for I/O space simulation, you *must* set a breakpoint on I/O instructions.(For details about setting breakpoints for I/O space simulation, refer to Section 7.7 on page 7-10.

There are several ways to set a breakpoint:



-
- 1) Point to the line of assembly language code or C code where you'd like to set a breakpoint.

- 2) Click the left button.

Repeating this action clears the breakpoint.



-
- 1) Make the FILE or DISASSEMBLY window the active window.

- 2) Use the arrow keys to move the cursor to the line of code where you'd like to set a breakpoint.

- 3) Press the **F9** key.

Repeating this action clears the breakpoint.



-
- ba** If you know the address where you'd like to set a breakpoint, you can use the BA (breakpoint add) command. This command is useful because it doesn't require you to search through code to find the desired line. The syntax for the BA command is:

ba *address*

This command sets a breakpoint at *address*. This parameter can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label. You cannot set multiple breakpoints at the same statement.

10.2 Clearing a Breakpoint

There are several ways to clear a 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.



-
- 1) Point to a breakpointed assembly language or C statement.
 - 2) Click the left button.



-
- 1) Use the arrow keys or the DASM command to move the cursor to a breakpointed assembly language or C statement.
 - 2) Press the **F9** key.



br If you want to clear **all** the breakpoints that are set, use the BR (breakpoint reset) command. This command is useful because it doesn't require you to search through code to find the desired line. The syntax for the BR command is:

br

bd If you'd like to clear one specific breakpoint and you know the address of this breakpoint, you can use the BD (breakpoint delete) command. The syntax for the BD command is:

bd *address*

This command clears the breakpoint at *address*. This parameter can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label. If no breakpoint is set at *address*, the debugger ignores the command.

10.3 Finding the Breakpoints That Are Set



bl Sometimes you may need to know where breakpoints are set. For example, the BD command's *address* parameter must correspond to the address of a breakpoint that is set. The BL (breakpoint list) command provides an easy way to get a complete listing of all the breakpoints that are currently set in your program. The syntax for this command is:

bl

The BL command displays a table of breakpoints in the COMMAND window display area. BL lists all the breakpoints that are set, in the order in which you set them. Here's an example of this type of list:

<u>Address</u>	<u>Symbolic Information</u>
004d	in main, at line 60, "c:\c2xh11\sample.c"
0051	

The address is the memory address of the breakpoint. The symbolic information identifies the function, line number, and filename of the breakpointed C statement:

- If the breakpoint was set in assembly language code, you'll see only an address unless the statement defines a symbol.
- If the breakpoint was set in C code, you'll see the address together with symbolic information.

Customizing the Debugger Display

The debugger display is completely configurable; you can create the interface that is best suited for your use. Besides being able to size and position individual windows, you can change the appearance of many of the display features, such as window borders, how the current statement is highlighted, etc. In addition, if you're using a color display, you can change the colors of any area on the screen. Once you've customized the display to your liking, you can save the custom configuration for use in future debugging sessions.

Topic	Page
<i>The commands described in this section are most useful if you have a color display. If you are using a monochrome display, these commands change the shades of gray on your display.</i>	11.1 Changing the Colors of the Debugger Display 11-2
	<i>area names:</i> common display areas 11-3
	<i>area names:</i> window borders 11-4
	<i>area names:</i> COMMAND window 11-4
	<i>area names:</i> DISASSEMBLY and FILE windows 11-5
	<i>area names:</i> data-display windows 11-6
	<i>area names:</i> menu bar and pulldown menus 11-7
<i>These sections are useful with both color and monochrome displays. They tell you how to change the window border styles, save and restore custom display configurations, and customize the command-line prompt.</i>	11.2 Changing the Border Styles of the Windows 11-8
	11.3 Saving and Using Custom Displays 11-9
	Changing the default display for monochrome monitors 11-9
	Saving a custom display 11-10
	Loading a custom display 11-10
	Invoking the debugger with a custom display 11-11
	Returning to the default display 11-11
11.4 Changing the Prompt 11-12	

11.1 Changing the Colors of the Debugger Display

You can use the debugger with a color or a monochrome display; the commands described in this section are most useful if you have a color display. If you are using a monochrome display, these commands change the shades on your display. For example, if you are using a black-and-white display, these commands change the shades of gray that are used.



color
scolor

You can use the COLOR or SCOLOR command to change the colors of areas in the debugger display. The format for these commands is:

```
color  area name, attribute1 [, attribute2 [, attribute3 [, attribute4]]]
scolor area name, attribute1 [, attribute2 [, attribute3 [, attribute4]]]
```

These commands are similar. However, SCOLOR updates the screen immediately, and COLOR doesn't update the screen (the new colors/attributes take effect as soon as the debugger executes another command that updates the screen). Typically, you might use the COLOR command several times, followed by an SCOLOR command to put all of the changes into effect at once.

The *area name* parameter identifies the areas of the display that are affected. The *attributes* identify how the areas are affected. Table 11–1 lists the valid values for the *attribute* parameters.

Table 11–1. Colors and Other Attributes for the COLOR and SCOLOR Commands

(a) Colors

black	blue	green	cyan
red	magenta	yellow	white

(b) Other attributes

bright	blink
--------	-------

The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Table 11–2 lists valid values for the *area name* parameters. This is a long list; the subsections following the table further identify these areas.

Table 11-2. Summary of Area Names for the COLOR and SCOLOR Commands

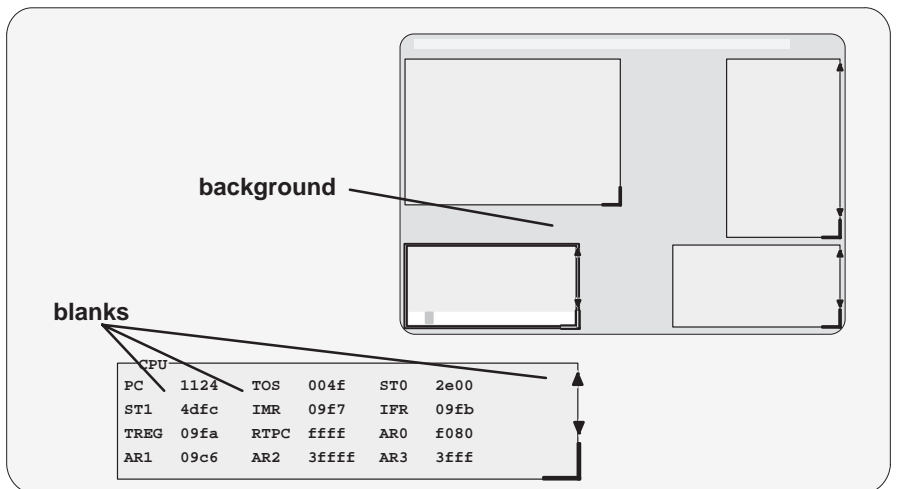
menu_bar	menu_border	menu_entry	menu_cmd
menu_hilite	menu_hicmd	win_border	win_hiborder
win_resize	field_text	field_hilite	field_edit
field_label	field_error	cmd_prompt	cmd_input
cmd_cursor	cmd_echo	asm_data	asm_cdata
asm_label	asm_clabel	background	blanks
error_msg	file_line	file_eof	file_text
file_brk	file_pc	file_pc_brk	

Note: Listing order is left to right, top to bottom.

You don't have to type an entire *attribute* or *area name*; you need type only enough letters to uniquely identify either parameter. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they're listed in Table 11-2 (left to right, top to bottom).

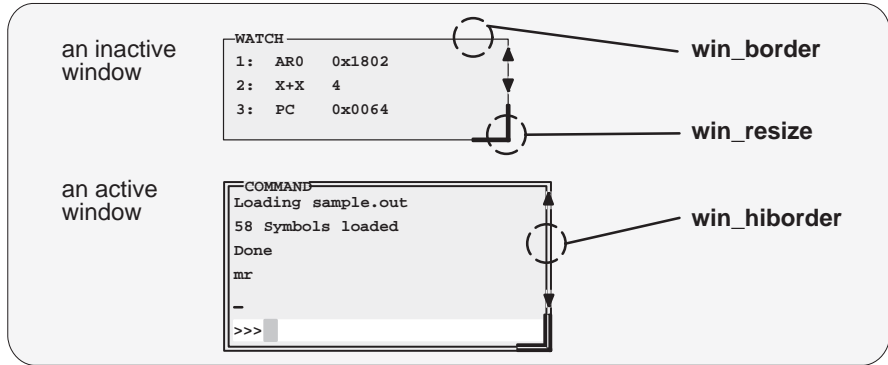
The remainder of this section identifies these areas.

area names: common display areas



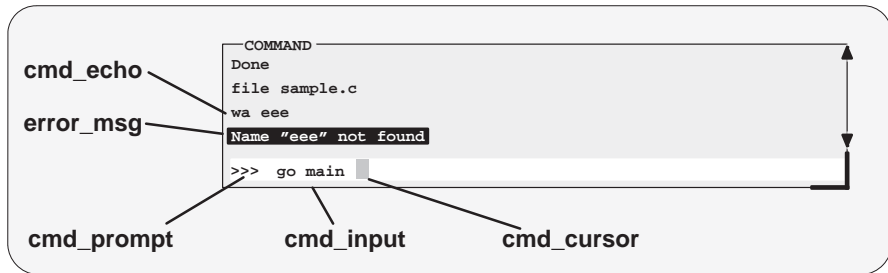
Area identification	Parameter name
Screen background (behind all windows)	background
Window background (inside windows)	blanks

area names: window borders



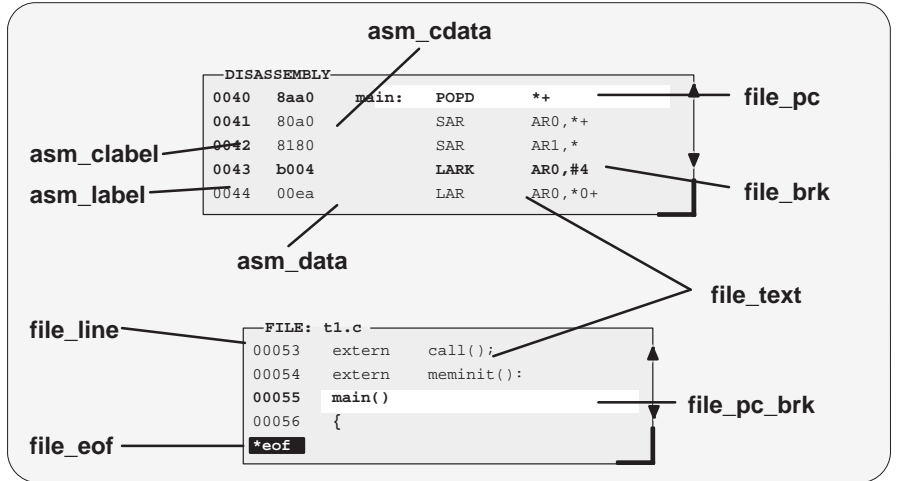
Area identification	Parameter name
Window border for any window that isn't active	win_border
The reversed "L" in the lower right corner of a resizable window	win_resize
Window border of the active window	win_hiborder

area names: COMMAND window



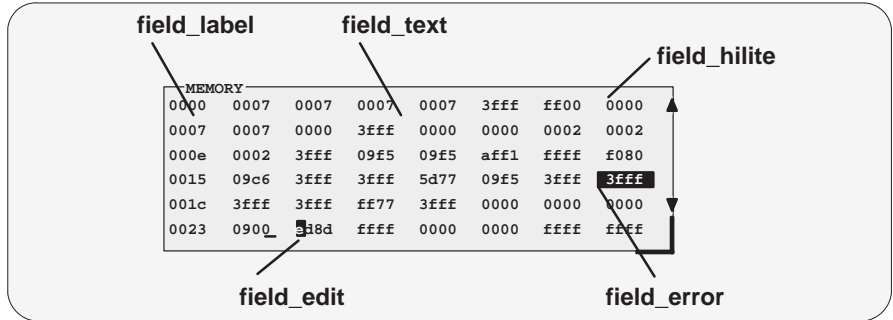
Area identification	Parameter name
Echoed commands in display area	cmd_echo
Errors shown in display area	error_msg
Command-line prompt	cmd_prompt
Text that you enter on the command line	cmd_input
Command-line cursor	cmd_cursor

area names: DISASSEMBLY and FILE windows



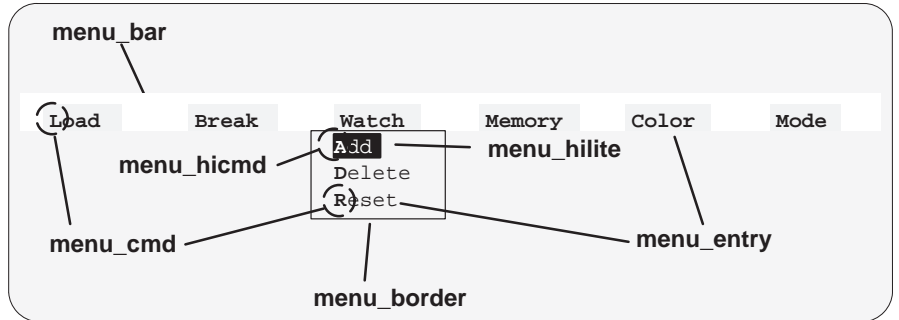
Area identification	Parameter name
Object code in DISASSEMBLY window that is associated with current C statement	asm_cdata
Object code in DISASSEMBLY window	asm_data
Addresses in DISASSEMBLY window	asm_label
Addresses in DISASSEMBLY window that are associated with current C statement	asm_clabel
Line numbers in FILE window	file_line
End-of-file marker in FILE window	file_eof
Text in FILE or DISASSEMBLY window	file_text
Breakpointed text in FILE or DISASSEMBLY window	file_brk
Current PC in FILE or DISASSEMBLY window	file_pc
Breakpoint at current PC in FILE or DISASSEMBLY window	file_pc_brk

area names: data-display windows



Area identification	Parameter name
Label of a window field (includes register names in CPU window, addresses in MEMORY window, index numbers and labels in WATCH window, member names in DISP window)	field_label
Text of a window field (includes data values for all data-display windows) and of most command output messages in command window	field_text
Text of a highlighted field	field_hilite
Text of a field that has an error (such as an invalid memory location)	field_error
Text of a field being edited (includes data values for all data-display windows)	field_edit

area names: menu bar and pulldown menus



Area identification	Parameter name
Top line of display screen; background to main menu choices	menu_bar
Border of any pulldown menu	menu_border
Text of a menu entry	menu_entry
Invocation key for a menu or menu entry	menu_cmd
Text for current (selected) menu entry	menu_hilite
Invocation key for current (selected) menu entry	menu_hicmd

11.2 Changing the Border Styles of the Windows

In addition to changing the colors of areas in the display, the debugger allows you to modify the border styles of the windows.



border Use the BORDER command to change window border styles. The format for this command is:

border [*active window style*] [, *inactive window style*] [, *resize style*]

This command can change the border styles of the active window, the inactive windows, and any window that is being resized. The debugger supports nine border styles. Each parameter for the BORDER command must be one of the numbers that identifies these styles:

Index	Style
0	Double-lined box
1	Single-lined box
2	Solid 1/2-tone top, double-lined sides and bottom
3	Solid 1/4-tone top, double-lined sides and bottom
4	Solid box, thin border
5	Solid box, heavy sides, thin top and bottom
6	Solid box, heavy borders
7	Solid 1/2-tone box
8	Solid 1/4-tone box

Here are some examples of the BORDER command. Note that you can skip parameters, if desired.

```
border 6,7,8           Change style of active, inactive, and resize windows
border 1,,2           Change style of active and resize windows
border ,3             Change style of inactive window
```

Note that you can execute the BORDER command as the Border selection on the Color pulldown menu. The debugger displays a dialog box so that you can enter the parameter values; in the dialog box, *active window style* is called *foreground*, and *inactive window style* is called *background*.

11.3 Saving and Using Custom Displays

The debugger allows you to save and use as many custom configurations as you like.

When you invoke the debugger, it looks for a screen configuration file:

- If you are using the **SWDS**, the screen configuration file is named *init.clr*.
- If you are using the **VMS** or **Sun** version of the **simulator**, the screen configuration file is named *clrs.dat*.
- If you are using the **PC** version of the **simulator**, there is no screen configuration file.

The screen configuration file defines how various areas of the display will appear. If the debugger doesn't find this file, it uses the default screen configuration. Initially, *init.clr* and *clrs.dat* define screen configurations that exactly match the default configuration.

The debugger supports two commands for saving and restoring custom screen configurations into files. The filenames that you use for restoring configurations must correspond to the filenames that you used for saving configurations. Note that these are binary files, not text files, so you can't edit the files with a text editor.

Changing the default display for monochrome monitors

The default display is most useful with color monitors. The debugger highlights changed values, messages, and other information with color; this may not be particularly helpful if you are using a monochrome monitor.

If you are using the SWDS, the debugger package includes another screen configuration file named *mono.clr* that defines a screen configuration that can be used with monochrome monitors. The best way to use this configuration is to rename the file:

- 1) Rename the original *init.clr* file—you might want to call it *color.clr*.
- 2) Now rename the *mono.clr* file. Call it *init.clr*. Now, whenever you invoke the debugger, it will automatically come up with a customized screen configuration for monochrome files.

If you aren't happy with the way that this file defines the screen configuration, you can customize it.

Saving a custom display



ssave Once you've customized the debugger display to your liking, you can use the SSAVE command to save the current screen configuration to a file. The format for this command is:

ssave [*filename*]

This saves the screen colors, window positions, window sizes, and border styles for all debugging modes. The *filename* parameter names the new screen configuration file. You can include path information (including relative pathnames); if you don't supply path information, the debugger places the file in the current directory.

If you don't supply a *filename*, then the debugger saves the current configuration into a file named `init.clr`.

Note that you can execute this command as the Save selection on the Color pulldown menu.

Loading a custom display



sconfig You can use the SCONFIG command to restore the display to a particular configuration. The format for this command is:

sconfig [*filename*]

This restores the screen colors, window positions, window sizes, and border styles that were saved with the SSAVE command into *filename*. If you don't supply a *filename*, the debugger looks for `init.clr`. The debugger searches for the file in the current directory and then in directories named with the `D_DIR` environment variable.

Note that you can execute this command as the Load selection on the Color pulldown menu.

Invoking the debugger with a custom display

If you set up the screen in a way that you like and always want to invoke the debugger with this screen configuration, you have two choices for accomplishing this:

- Save the configuration in `init.clr` (SWDS).
- Add a line to the batch file that the debugger executes at invocation time (For the SWDS, this file is `dbinit.cmd`; for the simulator, this file is `siminit.cmd`.) This line should use the `SCONFIG` command to load the custom configuration.

Returning to the default display

If you saved a custom configuration into `init.clr` or `clrs.dat` but don't want the debugger to come up in that configuration, then rename the file or delete it. If you are in the debugger, have changed the configuration, and would like to revert to the default, just execute the `SCONFIG` command without a filename.

11.4 Changing the Prompt



prompt The debugger enables you to change the command-line prompt by using the PROMPT command. The format of this command is:

prompt *new prompt*

The *new prompt* can be any string of characters, excluding semicolons and commas. (If you type a semicolon or a comma, it terminates the prompt string.)

Note that the SSAVE command doesn't save the command-line prompt as part of a custom configuration. The SCONFIG command doesn't change the command-line prompt. If you change the prompt, it stays changed until you change it again, even if you use SCONFIG to load a different screen configuration.

If you always want to use a different prompt, you can add a PROMPT statement to the batch file that the debugger executes at invocation time (for the SWDS, this file is dbinit.cmd; for the simulator, this file is siminit.cmd.)

You can also execute this command as the Prompt selection on the Color pulldown menu.

Summary of Commands and Special Keys

This chapter summarizes the debugger's commands and special key sequences.

Topic	Page
<i>The chapter begins with a description of the various categories of debugger commands and then lists the various commands that fall under these categories.</i>	12.1 Functional Summary of Debugger Commands 12-2
	Changing modes 12-3
	Managing windows 12-3
	Performing DOS-like tasks 12-3
	Managing and displaying data 12-4
	Displaying files and loading programs 12-4
	Managing breakpoints 12-4
	Customizing the screen 12-5
	Memory mapping 12-5
	Running programs 12-6
<i>The main portion of this chapter is the alphabetical command reference. Each debugger command is listed with its syntax, applicable modes, its correspondence to a pulldown menu (if any), and a short description.</i>	12.2 Alphabetical Summary of Debugger Commands 12-7
<i>The chapter ends with a summary of special keys and their functions in the debugger environment.</i>	12.3 Summary of Special Keys 12-35
	Editing text on the command line 12-35
	Using the command history 12-35
	Switching modes 12-36
	Halting or escaping from an action 12-36
	Displaying the pulldown menus 12-36
	Running code 12-37
	Selecting or closing a window 12-37
	Moving or sizing a window 12-37
	Scrolling through a window's contents 12-38
	Editing data or selecting the active field 12-38

12.1 Functional Summary of Debugger Commands

This section summarizes the debugger commands according to these categories:

- Changing modes.** These commands enable you to switch freely among the three debugging modes (auto, mixed, and assembly). You can select these commands from the Mode pulldown menu, also.
- Managing windows.** These commands enable you to select the active window and move or resize the active window. You can perform these functions with the mouse, also.
- Performing DOS-like tasks.** These commands enable you to perform several DOS-like functions and provide you with some control over the target system.
- Displaying and changing data.** These commands enable you to display and evaluate a variety of data items. Some of these commands are available on the Watch pulldown menu, also.
- Displaying files and loading programs.** These commands enable you to change the displays in the FILE and DISASSEMBLY windows and to load object files into memory. Several of these commands are available on the Load pulldown menu.
- Managing breakpoints.** These commands provide you with a command-line method for controlling software breakpoints. These commands are available through the Break pulldown menu. You can also set/clear breakpoints interactively.
- Customizing the screen.** These commands allow you to customize the debugger display, then save and later reuse the customized displays. These commands are available from the Color pulldown menu, also.
- Memory mapping.** These commands enable you to define the areas of target memory that the debugger can access or to fill a memory range with an initial value. These commands are available on the Memory pulldown menu, also.
- Running programs.** These commands provide you with a variety of methods for running your programs in the debugger environment. The basic run and single-step commands are available on the menu bar, also.

Changing modes

To do this	Use this command	See page
Put the debugger in assembly mode	asm	12-8
Put the debugger in auto mode for debugging C code	c	12-10
Put the debugger in mixed mode	mix	12-21

Managing windows

To do this	Use this command	See page
Select the active window	win	12-33
Reposition the active window	move	12-22
Resize the active window	size	12-29

Performing DOS-like tasks

To do this	Use this command	See page
Clear all displayed information from the COMMAND window display area	cls	12-10
Change the current working directory from within the debugger environment	cd/chdir	12-10
List the contents of the current directory or any other directory	dir	12-13
Name additional directories that can be searched when you load source files	use	12-32
Execute commands from a batch file	take	12-32
Reset the SWDS	reset	12-25
Exit the debugger	quit	12-24

Managing and displaying data

To do this	Use this command	See page
Show the type of a data item	whatis	12-33
Evaluate and display the result of a C expression	?	12-7
Evaluate a C expression without displaying the results	eval	12-14
Display the values in an array or structure or display the value that a pointer is pointing to	disp	12-13
Display a different range of memory in the MEMORY window	mem	12-21
Continuously display the value of a variable, register, or memory location within the WATCH window	wa	12-33
Delete a data item from the WATCH window	wd	12-33
Delete all data items from the WATCH window and close the WATCH window	wr	12-34

Displaying files and loading programs

To do this	Use this command	See page
Display a text file in the FILE window	file	12-14
Display C and/or assembly language code at a specific point	addr	12-7
Display assembly language code at a specific address	dasm	12-13
Display a specific C function	func	12-15
Reopen the CALLS window	calls	12-10
Load an object file	load	12-16
Load only the object-code portion of an object file	reload	12-24
Load only the symbol-table portion of an object file	sload	12-30

Managing breakpoints

To do this	Use this command	See page
Add a breakpoint	ba	12-8
Delete a breakpoint	bd	12-8
Display a list of all the breakpoints that are set	bl	12-9
Reset (delete) all breakpoints	br	12-9

Customizing the screen

To do this	Use this command	See page
Change the screen colors and update the screen immediately	scolor	12-28
Change the screen colors, but don't update the screen immediately	color	12-11
Change the border style of any window	border	12-9
Change the command-line prompt	prompt	12-24
Save a custom screen configuration	ssave	12-31
Load and use a previously saved custom screen configuration	sconfig	12-29

Memory mapping

To do this	Use this command	See page
Initialize a block of memory	fill	12-15
Enable or disable memory mapping	map	12-18
Add an address range to the memory map	ma	12-17
Connect a simulated I/O port to an input or output file	mc	12-18
Delete an address range from the memory map	md	12-19
Disconnect a simulated I/O port	mi	12-20
Reset the memory map (delete all ranges)	mr	12-23
Save the values in a block of memory to a system file	ms	12-23
Display a list of the current memory map settings	ml	12-20

Running programs

To do this	Use this command	See page
Run a program	run	12-26
Run a program up to a certain point	go	12-16
Single-step through assembly language or C code	step	12-31
Single-step through assembly language or C code, one C statement at a time	cstep	12-12
Single-step through assembly language or C code; step over function calls	next	12-24
Single-step through assembly language or C code, one C statement at a time; step over function calls	cnext	12-11
Run a program with benchmarking—count the number of CPU clock cycles consumed by the executing portion of code (simulator only)	runb	12-27
Execute code in a function and return to the function's caller	return	12-26
Reset the program entry point	restart	12-25
Disconnect the SWDS from the target system and run free (SWDS only)	runf	12-27
Halt the target system after executing a RUNF command (SWDS only)	halt	12-16
Execute commands from a batch file	take	12-32
Reset the SWDS	reset	12-25

12.2 Alphabetical Summary of Debugger Commands

Commands are not case sensitive; to emphasize this, command names are shown in both uppercase and lowercase throughout this book.

?

Evaluate Expression

Syntax

? *expression*[@prog | @data]

Menu selection

none

Description

The ? (evaluate expression) command evaluates an expression and shows the result in the COMMAND window display area. 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 *expression* identifies an address, you can follow it with @prog to identify program memory or @data to identify data memory. Without the suffix, the debugger treats an address-expression as a program-memory location.

If the result of *expression* is not an array or structure, then the debugger displays the results in the COMMAND window. If *expression* is a structure or array, ? displays the entire contents of the structure or array; you can halt long listings by pressing **(ESC)**.

addr

Display Code at Selected Address

Syntax

addr *address*[@prog | @data]
addr *function name*

Menu selection

none

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* in the FILE window.
- In mixed mode, ADDR affects both the DISASSEMBLY and FILE windows.

The *address* parameter is treated as a program-memory address.

Note: Effects of ADDR on the FILE Window

ADDR affects the FILE window only if the specified *address* is in a C function.

asm

Enter Assembly-Only Debugging Mode

Syntax

asm

Menu selection

MoDe→Asm

Description

The ASM command changes from the current debugging mode to assembly mode. If you're already in assembly mode, the ASM command has no effect.

ba

Breakpoint Add

Syntax

ba *address*

Menu selection

Break→Add

Description

The BA command sets a breakpoint at a specific *address*. This command is useful because it doesn't require you to search through code to find the desired line. The *address* can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label.

Breakpoints can be set in program memory only; the *address* parameter is treated as a program-memory address.

bd

Breakpoint Delete

Syntax

bd *address*

Menu selection

Break→Delete

Description

The BD command clears a 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. The *address* is treated as a program-memory address.

bl*Breakpoint List***Syntax****bl****Menu selection****Break**→**List****Description**

The BL command provides an easy way to get a complete listing of all the breakpoints that are currently set in your program. It displays a table of breakpoints in the COMMAND window display area. BL lists all the breakpoints that are set, in the order in which you set them.

border*Change Style of Window Border***Syntax****border** [*active window style*] [[*,inactive window style*] [*,resize window style*]**Menu selection****Color**→**Border****Description**

The BORDER command changes the border style of the active window, the inactive windows, and any window that you're resizing. The debugger supports nine border styles. Each parameter for the BORDER command must be one of the numbers that identifies these styles:

Index	Style
0	Double-lined box
1	Single-lined box
2	Solid 1/2-tone top, double-lined sides/bottom
3	Solid 1/4-tone top, double-lined sides/bottom
4	Solid box, thin border
5	Solid box, heavy sides, thin top/bottom
6	Solid box, heavy borders
7	Solid 1/2-tone box
8	Solid 1/4-tone box

Note that you can execute the BORDER command as the Border selection on the Color pulldown menu. The debugger displays a dialog box so that you can enter the parameter values; in the dialog box, *active window style* is called *foreground*, and *inactive window style* is called *background*.

br*Breakpoint Reset***Syntax****br****Menu selection****Break**→**Reset****Description**

The BR command clears all breakpoints that are set.

c	<i>Enter Auto Debugging Mode</i>
Syntax	c
Menu selection	MoDe→ C (auto)
Description	The C command changes from the current debugging mode to auto mode. If you're already in auto mode, then the C command has no effect.

calls	<i>Open CALLS Window</i>
Syntax	calls
Menu selection	none
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 up again.

cd, chdir	<i>Change Directory</i>
Syntax	cd [<i>directory name</i>] chdir [<i>directory name</i>]
Menu selection	none
Description	The CD or CHDIR command changes the current working directory from within the debugger. You can use relative pathnames as part of the <i>directory name</i> . If you don't use a <i>pathname</i> , the CD command displays the name of the current directory. Note that this command can affect any other command whose parameter is a filename, such as the FILE, LOAD, and TAKE commands, when used with the USE command. You can also use the CD command to change the current drive. For example, <pre>cd c: cd d:\csource cd c:\c20h11</pre>

cls	<i>Clear Screen</i>
Syntax	cls
Menu selection	none
Description	The CLS command clears all displayed information from the COMMAND window display area.

cnext*Single-Step C, Next Statement*

Syntax**cnext** [*expression*]**Menu selection**Next=**F10** (in C code)**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're using CNEXT to step through assembly language code, the debugger won't 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 don't 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 also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page 8-16, discusses this in detail).

color*Change Screen Colors*

Syntax**color** *area name*, *attribute*₁ [,*attribute*₂ [,*attribute*₃ [,*attribute*₄]]]**Menu selection**

none

Description

The COLOR command changes the color of specified areas of the debugger display. COLOR doesn't update the display; the changes take effect when another command, such as SCOLOR, updates the display. The *area name* parameter identifies the areas of the display that are affected. The *attributes* identify how the areas are affected. The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Valid values for the *attribute* parameters include

black	blue	green	cyan
red	magenta	yellow	white
bright		blink	

Valid values for the *area name* parameters include

menu_bar	menu_border	menu_entry	menu_cmd
menu_hilite	menu_hicmd	win_border	win_hiborder
win_resize	field_text	field_hilite	field_edit
field_label	field_error	cmd_prompt	cmd_input
cmd_cursor	cmd_echo	asm_data	asm_cdata
asm_label	asm_clabel	background	blanks
error_msg	file_line	file_eof	file_text
file_brk	file_pc	file_pc_brk	

You don't have to type an entire *attribute* or *area name*; you need type only enough letters to uniquely identify the attribute. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they're listed above (left to right, top to bottom).

cstep

Single-Step C

Syntax

cstep [*expression*]

Menu selection

Step=**F8** (in C code)

Description

The CSTEP single-steps through a program one C statement at a time, updating the display after executing each statement. If you're using CSTEP to step through assembly language code, the debugger won't update the display until it has executed all assembly language statements associated with a single C statement.

If you're 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` debug option). When function execution completes, single-step execution returns to the caller. If the function wasn't compiled with the debug option, the debugger executes the function but doesn't show single-step execution of the function.

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page 8-16, discusses this in detail).

dasm*Display Selected Disassembly***Syntax****dasm** *address*[**@prog** | **@data**]**dasm** *function name***Menu selection**

none

Description

The DASM command displays code beginning at a specific point within the DISASSEMBLY window. By default, the *address* parameter is treated as a program-memory address. However, you can follow it with **@prog** to identify program memory or with **@data** to identify data memory.

dir*Show Directory Contents***Syntax****dir** [*directory name*]**Menu 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 don't use the parameter, the debugger lists the contents of the current directory.

disp*Open DISPLAY Window***Syntax****disp** *expression*[**@prog** | **@data**]**Menu selection**

none

Description

The DISP command opens a DISP window to display the contents of an array, structure, or 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. If the *expression* identifies an address, you can follow it with **@prog** to identify program memory or **@data** to identify data memory. Without the suffix, the debugger treats an address-expression as a program-memory location.

Once you open a DISP window, you may find that a displayed member is itself an array, structure, or pointer:

A member that is an array looks like this

[. . .]

A member that is a structure looks like this

{. . .}

A member that is a pointer looks like an address

0x00000000

You can display the additional data (the data pointed to or the members of the array or structure) in another DISP window by using the DISP command again,

using the arrow keys to select the field and then pressing **F9**, or pointing the mouse cursor to the field and pressing the left mouse button. You can have up to 120 DISP windows open at the same time.

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 DISP 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.

eval

Evaluate Expression

Syntax

```
eval expression[@prog | @data]
e expression[@prog | @data]
```

Menu selection

none

Description

The EVAL command evaluates an expression like the ? command does *but does not show the result* in the COMMAND window display area. EVAL is useful for assigning values to registers or memory locations in a batch file (where it's not necessary to display the result).

If the *expression* identifies an address, you can follow it with @prog to identify program memory or @data to identify data memory. Without the suffix, the debugger treats an address-expression as a program-memory location.

file

Display Text File

Syntax

```
file filename
```

Menu selection

Load→File

Description

The FILE command displays the contents of any text file in the FILE window. The debugger continues to display this file until you run a program and halt in a C function. This command is intended primarily for displaying C source code. You can view only one text file at a time.

You are restricted to displaying files that are 65,518 bytes long or less.

fill*Fill Memory***Syntax**

fill *address,page,length,data*

Menu selection

Memory→**Fill**

Description

The FILL command fills a block of memory 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 *page* is a 1-digit number that identifies the type of memory (program or data) to fill:

To fill this type of memory	Use this value as the <i>page</i> parameter
Program memory	0
Data memory	1

- The *data* parameter is the value that is placed in each word in the block.

func*Display Function***Syntax**

func *function name*
func *address*

Menu selection

none

Description

The FUNC command displays a specified C function in the FILE window. You can identify the function by its name or its address; an *address* parameter is treated as a program-memory address. Note that FUNC works similarly to FILE, but you don't need to identify the name of the file that contains the function.

go

Run to Specified Address

Syntax

go [*address*]

Menu selection

none

Description

The GO command executes code up to a specific point in your program. The *address* parameter is treated as a program-memory address. If you don't supply an *address*, then GO acts like a RUN command without an *expression* parameter.

halt

Halt Target System

Syntax

halt

Menu selection

none

Description

The HALT command halts the target system after you've entered a RUNF command. When you invoke the debugger, it automatically executes a HALT command. Thus, if you enter a RUNF, quit the debugger, and later reinvoke the debugger, you will effectively reconnect the SWDS to the target system and run the debugger in its normal mode of operation.

load

Load Object File

Syntax

load *object filename*

Menu selection

Load→ Load

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. Note that the LOAD command clears the old symbol table and closes the WATCH and DISP windows.

ma**Memory Map Add**

Syntax `ma address, page, length, type`

Menu selection **Memory**→**Add**

Description

The MA command identifies valid ranges of target memory. Note that a new memory map 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 data or program memory. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.
- The *page* parameter is a 1-digit number that identifies the type of memory (program, data, or I/O) that the range occupies:

To identify this page,	Use this value as the <i>page</i> parameter
Program memory	0
Data memory	1
I/O space	2

- 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, ROM, or READONLY
Write-only memory	W, WOM, or WRITEONLY
Read/write memory	WR or RAM
No-access memory	PROTECT
Input port	IPOINT
Output port	OPOINT
Input/output port	IOPORT

You can use the parameters (page 2 and type IPOINT, OPOINT, or IOPORT) in conjunction with the MC command to simulate I/O ports.

map

Enable Memory Mapping

Syntax

map {on | off}

Menu selection

Memory→Enable

Description

The MAP command enables or disables memory mapping. In some instances, you may want to explicitly enable or disable memory. Note that disabling memory mapping can cause bus fault problems in the target because the debugger may attempt to access nonexistent memory.

mc

Connect a Simulated I/O Port to a File

Syntax

mc *port address, page, filename, {READ | WRITE}*

Menu selection

Memory→Connect

Description

The MC command connects IPORT, OPORT, or IOPORT to an input or output file. Before you can connect the port, you must add it to the memory map with the MA command.

- The *port address* parameter defines the address of the I/O port. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.
- The *page* parameter is a 1-digit number that identifies the page that the port occupies.

To identify this page,	Use this value as the <i>page</i> parameter
Data memory	1
I/O space	2

Ports are usually on page 2 (the I/O space).

- The *filename* 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.
- The final parameter is specified as **READ** or **WRITE** and defines how the file will be used (for input or output, respectively).

The file is accessed during an IN or OUT instruction to the associated port address. 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 will read or write to the associated file.

This port-connect feature can also be used for some simulation of serial ports. The DXR and DRR registers can be connected to files.

If you're using the SWDS to simulate I/O space, you *must* set a breakpoint on the I/O instructions. For details about simulating I/O space with the SWDS, refer to Section 7.7 on page 7-10 .

md**Memory Map Delete****Syntax**

md *address, page*

Menu selection

Memory→Delete

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 program, data, or I/O memory. If you supply an *address* that is not the starting address of a range, the debugger displays this error message in the COMMAND window display area:

```
Specified map not found
```

- The *page* parameter is a 1-digit number that identifies the type of memory (program, data, or I/O) that the range occupies:

To identify this page,	Use this value as the <i>page</i> parameter
Program memory	0
Data memory	1
I/O space	2

Note: Deleting a Simulated I/O Port

If you are using the simulator and want to use the MD command to remove a simulated I/O port, you must first disconnect the port with the MI command.

mi

Disconnecting an I/O Port

Syntax

mi *port address, page, {READ | WRITE}*

Menu selection

Memory→DisConn

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 have been previously defined with the MC command.
- The *page* parameter is a 1-digit number that identifies the type of memory (program, data, or I/O) that the port occupies:

To identify this page,	Use this value as the <i>page</i> parameter
Data memory	1
I/O space	2

The page parameter for the MI command must match the page parameter that was used when the port was connected using the MC command.

ml

Memory Map List

Syntax

ml

Menu selection

Memory→List

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.

mem*Modify MEMORY Window Display*

Syntax**mem** *expression* [@**prog** | @**data**]**Menu selection**

none

Description

The MEM command identifies a new starting address for the block of memory displayed in the MEMORY window. 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.

You can display either program or data memory:

- By default, the MEMORY window displays data memory. Although it is not necessary, you can explicitly specify data memory by following the *expression* parameter with a suffix of @**data**.
- You can display the contents of program memory by following the *expression* parameter with a suffix of @**prog**. When you do this, the MEMORY window's label changes to MEMORY [PROG] so that there is no confusion about the type of memory being displayed.

mix*Enter Mixed Debugger Mode*

Syntax**mix****Menu selection**

MoDe→Mixed

Description

The MIX command changes from the current debugging mode to mixed mode. If you're already in mixed mode, the MIX command has no effect.

move

Move Window

Syntax **move** [*X position*, *Y position* [, *width*, *length*]]

Menu selection none

Description The MOVE command moves the active window to the specified XY position. If you choose, you can resize the window while you move it (see the SIZE command for valid *width* and *length* values). You can use the MOVE command in one of two ways:

- By supplying a specific *X position* and *Y position* or
- By omitting the *X position* and *Y position* parameters and using function keys to interactively move the window.

Valid X and Y positions depend on the screen size and the window size. These are the minimum and maximum XY positions. The maximum values assume that the window is as small as possible; for example, if a window was half as tall as the screen, you wouldn't be able to move its upper left corner to an X position on the bottom half of the screen.





Screen size	Debugger options	Valid X positions	Valid Y positions
80 characters by 25 lines	none	0 through 76	1 through 22
80 characters by 39 lines [†]	-b	0 through 76	1 through 36
80 characters by 43 lines [‡]			1 through 40
80 characters by 50 lines [§]			1 through 47
120 characters by 43 lines	-bb	0 through 116	1 through 40
132 characters by 43 lines	-bbb	0 through 128	1 through 40
80 characters by 60 lines	-bbbb	0 through 76	1 through 57
100 characters by 60 lines	-bbbbb	0 through 106	1 through 57



[†] PC version of simulator running under Microsoft Windows

[‡] PC with EGA card; Sun

[§] PC with VGA card

Note: To use larger screen sizes, you must invoke the debugger with the appropriate -b option. If you enter the MOVE command without *X position* and *Y position* parameters, you can use arrow keys to move the window.

-  Moves the active window down one line.
-  Moves the active window up one line.
-  Moves the active window left one character position.
-  Moves the active window right one character position.

When you're finished using the arrow keys, you *must* press  or .

mr*Memory Map Reset***Syntax****mr****Menu selection****Memory→Reset****Description**

The MR command resets the debugger's memory map by deleting all defined memory ranges from the map.

ms*Save a Block of Memory to a File***Syntax****ms** *address, page, length, filename***Menu selection****Memory→Save****Description**

The MS command saves the values in a block of memory to a system file; files are saved in COFF format.

- The *address* parameter identifies the first address in the block.
- The *page* is a 1-digit number that identifies the type of memory (program, data, or I/O) to save:

To save this type of memory	Use this value as the <i>page</i> parameter
Program memory	0
Data memory	1
I/O space	2

- The *length* parameter defines the length, in words, of the range. This parameter can be any C expression.
- The *filename* is a system file.

next

Single-Step, Next Statement

Syntax

next [*expression*]

Menu selection

Next=**F10** (in disassembly)

Description

The NEXT command is similar to the STEP command. If you're 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 don't 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 also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page 8-16, discusses this in detail).

prompt

Change Command-Line Prompt

Syntax

prompt *new prompt*

Menu selection

Color→**P**rompt

Description

The PROMPT command changes the command-line prompt. The *new prompt* can be any string of characters (note that a semicolon or comma ends the string).

quit

Exit Debugger

Syntax

quit

Menu selection

none

Description

The QUIT command exits the debugger and returns to the DOS environment.

reload

Reload Object Code

Syntax

reload *object filename*

Menu selection

Load→**R**eload

Description

The RELOAD command loads only an object file *without* loading its associated symbol table. This is useful for reloading a program when target memory has been corrupted.

reset*Reset Target System*

Syntax**reset****Menu selection**

Load→ReseT

Description

The RESET command resets the SWDS and reloads the monitor. (Use c2xreset for the simulator.) Note that this is a *software* reset.

restart*Reset PC to Program Entry Point*

Syntax**restart**
rest**Menu selection**

Load→REstart

Description

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.)

return

Return to Function's Caller

Syntax

return
ret

Menu selection

none

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 pressing the left mouse button or pressing (ESC).

run

Run Code

Syntax

run [*expression*]

Menu selection

Run=F5

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 don't supply an *expression*, the program executes until it encounters a breakpoint or until you press the left mouse button or press (ESC).
- If you supply a logical or relational *expression*, this becomes a conditional run (described in detail on page 8-16).
- 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>Run Benchmark</i>	Simulator Only
-------------	----------------------	-----------------------

Syntax **runb**

Menu selection none

Description The RUNB command executes a specific section of code and counts the number of CPU clock cycles consumed by the execution. In order to operate correctly, *execution must be halted by a breakpoint*. 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 8.7, *Benchmarking*, on page 8-18.

Note: Simulator Only

<p>This command is for the simulator only; it does not work with the SWDS. If you attempt to use the RUNB command with an SWDS system, the debugger displays this error message:</p>
--

<pre>--- Execution error</pre>

runf	<i>Run Free</i>	SWDS Only
-------------	-----------------	------------------

Syntax **runf**

Menu selection none

Description The RUNF command disconnects the SWDS from the target system while code is executing. When you enter RUNF, the debugger clears all breakpoints, disconnects the SWDS 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.

The HALT command stops a RUNF; note that the debugger automatically executes a HALT when the debugger is invoked.

scolor

Change Screen Colors

Syntax

scolor *area name, attribute₁ [, attribute₂ [, attribute₃ [, attribute₄]]]*

Menu selection

Color→Config

Description

The SCOLOR command changes the color of specified areas of the debugger display and updates the display immediately. The *area name* parameter identifies the areas of the display that are affected. The *attributes* identify how the area is affected. The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Valid values for the *attribute* parameters include

black	blue	green	cyan
red	magenta	yellow	white
bright		blink	

Valid values for the *area name* parameters include

menu_bar	menu_border	menu_entry	menu_cmd
menu_hilite	menu_hicmd	win_border	win_hiborder
win_resize	field_text	field_hilite	field_edit
field_label	field_error	cmd_prompt	cmd_input
cmd_cursor	cmd_echo	asm_data	asm_cdata
asm_label	asm_clabel	background	blanks
error_msg	file_line	file_eof	file_text
file_brk	file_pc	file_pc_brk	

You don't have to type an entire *attribute* or *area name*; you need type only enough letters to uniquely identify the attribute. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they're listed above (left to right, top to bottom).

sconfig*Load Screen Configuration***Syntax****sconfig** [*filename*]**Menu selection**

Color→Load

Description

The SCONFIG command restores the display to a specified configuration. This restores the screen colors, window positions, window sizes, and border styles that were saved with the SSAVE command into *filename*. If you don't supply a *filename*, the debugger looks for *init.clr*. The debugger searches for the specified file in the current directory and then in directories named with the D_DIR environment variable.

size*Size Window***Syntax****size** [*width, length*]**Menu selection**

none

Description

The SIZE command changes the size of the active window. You can use the SIZE command in one of two ways:

- By supplying a specific *width* and *length* or
- By omitting the *width* and *length* parameters and using function keys to interactively resize the window.

Valid values for the width and length depend on the screen size and the window position on the screen. These are the minimum and maximum window sizes.

Screen size	Debugger option	Valid widths	Valid lengths
80 characters by 25 lines	none	4 through 80	3 through 24
80 characters by 39 lines [†]	-b	4 through 80	3 through 38
80 characters by 43 lines [‡]			3 through 42
80 characters by 50 lines [§]			3 through 49
120 characters by 43 lines	-bb	4 through 120	3 through 42
132 characters by 43 lines	-bbb	4 through 132	3 through 42
80 characters by 60 lines	-bbbb	4 through 80	3 through 59
100 characters by 60 lines	-bbbbb	4 through 100	3 through 59

[†] PC version of simulator running under Microsoft Windows

[‡] PC with EGA card; Sun

[§] PC with VGA card

Note: To use larger screen sizes, you must invoke the debugger with the appropriate -b option.

The maximum sizes assume that the window is in the upper left corner (beneath the menu bar). If a window is in the middle of the display, for example, you can't size it to the maximum height and width; you can size it only to the right and bottom screen borders.

If you enter the SIZE command without *width* and *length* parameters, you can use arrow keys to size the window.

- ⏴ Makes the active window one line longer.
- ⏵ Makes the active window one line shorter.
- ⏪ Makes the active window one character narrower.
- ⏩ Makes the active window one character wider.

When you're finished using the arrow keys, you *must* press `ESC` or `↵`.

sload

Load Symbol Table

Syntax

sload *object filename*

Menu selection

Load→Symbols

Description

The SLOAD command loads the symbol table of the specified object file. SLOAD is useful in a debugging environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). SLOAD clears the existing symbol table before loading the new one but does not modify memory or set the program entry point. Note that SLOAD closes the WATCH and DISP windows.

ssave*Save Screen Configuration*

Syntax**ssave** [*filename*]**Menu selection**

Color→Save

Description

The SSAVE command saves the current screen configuration to a file. This saves the screen colors, window positions, window sizes, and border styles. The *filename* parameter names the new screen configuration file. You can include path information (including relative pathnames); if you don't supply path information, the debugger places the file in the current directory. The filename is required for the simulator but optional for the SWDS. If you don't supply a *filename*, then the debugger saves the current configuration into a file named *init.clr* (SWDS) and places the file in the current directory.

step*Single-Step*

Syntax**step** [*expression*]**Menu selection**

Step=F8 (in disassembly)

Description

The STEP command single-steps through assembly language or C code. If you're 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're 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` debug option). When function execution completes, single-step execution returns to the caller. If the function wasn't compiled with the debug option, the debugger executes the function but doesn't show single-step execution of the function.

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page 8-16, discusses this in detail).

take

Execute Batch File

Syntax

take *batch filename* [, *suppress echo flag*]

Menu selection

none

Description

The TAKE command tells the debugger to read and execute commands from a batch file. The *batch filename* parameter identifies the file that contains commands.

By default, the debugger echoes the commands to the output area of the COMMAND window and updates the display as it reads the commands from the batch file.

- If you don't use the *suppress echo flag* parameter, or if you use it but supply a nonzero value, then the debugger behaves in the default manner.
- If you would like to suppress the echoing and updating, use the value 0 for the *suppress echo flag* parameter.

use

Use Different Directory

Syntax

use *directory name*

Menu selection

none

Description

The USE command names an additional directory that the debugger can search when looking for source files. You can specify only one directory at a time.

wa*Watch Value Add*

Syntax**wa** *expression*[@prog | @data] [, *label*]**Menu selection**

Watch→Add

Description

The WA command displays the value of *expression* in the WATCH window. If the WATCH window isn't open, executing WA opens the WATCH window. The *expression* parameter can be any C expression, including an expression that has side effects. If the *expression* identifies an address, you can follow it with @prog to identify program memory or with @data to identify data memory. Without the suffix, the debugger treats an address-expression as a program-memory location.

WA is most useful for watching an expression whose value changes over time; constant expressions provide no useful function in the watch window. The *label* parameter is optional. When used, it provides a label for the watched entry. If you don't use a *label*, the debugger displays the *expression* in the label field.

wd*Watch Value Delete*

Syntax**wd** *index number***Menu selection**

Watch→Delete

Description

The WD command deletes a specific item from the WATCH window. The WD command's *index number* parameter must correspond to one of the watch indexes listed in the WATCH window.

whatis*What Is This Data?*

Syntax**whatis** *symbol***Menu selection**

none

Description

The WHATIS command shows the data type of *symbol* in the COMMAND window display area. The *symbol* can be any variable (local, global, or static), a function name, structure tag, typedef name, or enumeration constant.

win*Select Active Window*

Syntax**win** *WINDOW NAME***Menu selection**

none

Description

The WIN command allows you to select the active window by name. Note that the *WINDOW NAME* is in uppercase (matching the name exactly as displayed). You can spell out the entire window name, but you really need specify only enough letters to identify the window.

If several of the same types of window are visible on the screen, don't use the WIN command to select one of them. 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 doesn't find the window you asked for (because you closed the window or misspelled the name), then the WIN command has no effect.

wr

WATCH Window Reset

Syntax

wr

Menu selection

Watch→Reset

Description










The WR command deletes all items from the WATCH window and closes the window.

12.3 Summary of Special Keys





The debugger provides function key, cursor key, and command key sequences for performing a variety of actions:

- Editing text on the command line
- Using the command history
- Switching modes
- Halting or escaping from an action
- Displaying the pulldown menus
- Running code
- Selecting or closing a window
- Moving or sizing a window
- Scrolling through a window's contents
- Editing data or selecting the active field

Editing text on the command line

To do this	Use these function keys
Enter the current command (note that if you press the return key in the middle of text, the debugger truncates the input text at the point where you press this key)	
Move back over text without erasing characters	  OR 
Move forward through text without erasing characters	 
Move back over text while erasing characters	
Move forward through text while erasing characters	
Insert text into the characters that are already on the command line	

Using the command history

To do this	Use these function keys
Move backward, one command at a time, through the command history	
Move forward, one command at a time, through the command history	 
Execute the last command in the list	

Switching modes

To do this	Use this function key
Switch debugging modes in this order: 	F3

Halting or escaping from an action




The escape key acts as an end or undo key in several situations.

To do this	Use this function key
Halt program execution	ESC
Close a pulldown menu	
Undo an edit of the active field in a data-display window (pressing this key leaves the field unchanged)	
Halt the display of a long list of data in the COMMAND window display area	



Displaying pulldown menus

To do this	Use these function keys
Display the Load menu	ALT L
Display the Break menu	ALT B
Display the Watch menu	ALT W
Display the Memory menu	ALT M
Display the Color menu	ALT C
Display the MoDe menu	ALT D
Display an adjacent menu	← or →
Execute any of the choices from a displayed pulldown menu	Press the highlighted letter corresponding to your choice

Running code













To do this	Use these function keys
Run code from the current PC (equivalent to the RUN command without an <i>expression</i> parameter)	
Single-step code from the current PC (equivalent to the STEP command without an <i>expression</i> parameter)	
Single-step code from the current PC; step over function calls (equivalent to the NEXT command without an <i>expression</i> parameter)	

Selecting or closing a window

To do this	Use these function keys
Select the active window (pressing this key makes each window active in turn; stop pressing the key when the desired window becomes active)	
Close the CALLS or DISP window (the window must be active before you can close it)	

Moving or sizing a window

You can use the arrow keys to interactively move a window after entering the MOVE or SIZE command without parameters.

To do this	Use these function keys
 Move the window down one line	
 Make the window one line longer	
 Move the window up one line	
 Make the window one line shorter	
 Move the window left one character position	
 Make the window one character narrower	
 Move the window right one character position	
 Make the window one character wider	

Scrolling through a window's contents

These descriptions and instructions for scrolling apply to the active window. Some of these descriptions refer to specific windows; if no specific window is named, then the description/instructions refer to any window that is active.

To do this	Use these function keys
Scroll up through the window contents, one window length at a time	
Scroll down through the window contents, one window length at a time	
Move the field cursor up, one line at a time	
Move the field cursor down, one line at a time	
<i>FILE window only:</i> Scroll left 8 characters at a time	
<i>Other windows:</i> Move the field cursor left 1 field; at the first field on a line, wrap back to the last fully displayed field on the previous line	
<i>FILE window only:</i> Scroll right 8 characters at a time	
<i>Other windows:</i> Move the field cursor right 1 field; at the last field on a line, wrap around to the first field on the next line	
<i>FILE window only:</i> Adjust the window's contents so that the first line of the text file is at the top of the window	
<i>FILE window only:</i> Adjust the window's contents so that the last line of the text file is at the bottom of the window	
<i>DISP windows only:</i> Scroll up through an array of structures	
<i>DISP windows only:</i> Scroll down through an array of structures	

Editing data or selecting the active field

The F9 function key makes the current field (the field that the cursor is pointing to) active. This has various effects, depending on the field.

To do this	Use this function key
<i>FILE or DISASSEMBLY window:</i> Set or clear a breakpoint	
<i>CALLS window:</i> Display the source to a listed function	
<i>Any data-display window:</i> Edit the contents of the current field	
<i>DISP window:</i> Open an additional DISP window to display a member that is an array, structure, or pointer	

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 you'll need to know in order to use C expressions as debugger command parameters.

	Topic	Page
<i>If you're an experienced C programmer, skip this section.</i>	13.1 C Expressions for Assembly Language Programmers	13-2
<i>Because the C expressions you'll use are parameters to debugger commands, some language features may be inappropriate. This section covers specific implementation issues (including necessary limitations and additional features) related to using C expressions as command parameters.</i>	13.2 Restrictions and Features Associated With Expression Analysis in the Debugger	13-4
	Restrictions	13-4
	Additional features	13-4

13.1 C Expressions for Assembly Language Programmers

It's not necessary for you to be an experienced C programmer in order to use the debugger. However, in order to use the debugger's full capabilities, you should at least 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: Single Values as Expressions

A single value or symbol is a legal C expression.

K&R contains a complete description of C expressions; to get you started, here's a summary of the operators that you can use in expression parameters.

☐ Reference operators

→	indirect structure reference	.	direct structure reference
[]	array reference	*	indirection (unary)
&	address (unary)		

☐ Arithmetic operators

+	addition (binary)	-	subtraction (binary)
*	multiplication	/	division
%	modulo	-	negation (unary)
(type)	typecast		

☐ Relational and logical operators

>	greater than	>=	greater than or equal to
<	less than	<=	less than or equal to
==	is equal to	!=	is not equal to
&&	logical AND		logical OR
!	logical NOT (unary)		

13.2 Restrictions and Features Associated With 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, there are a few limitations, as well as a few additional features 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* * ...)

Note that you can use up to six *s 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 TMS320C2x's 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. Note that 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.

Note that in this expression, *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*.

Note that these expression forms can be combined into an expression of the form:

filename.function name.variable name

- Any integral or void expression may 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
*AR5
*(AR2 + 123)
*(I*J)
```

By default, the values are treated as integers (that is, these expressions point to integer values).

- Any expression can be typecast 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 typecast 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 when you invoke it.

- 1) Reads options from the 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) *SWDS*: Looks for the `init.clr` screen configuration file.

VAX/VMS version of the simulator: Looks for the `clrs.dat` screen configuration file.

(The debugger searches for the screen configuration file in directories named with `D_DIR`.)

- 5) Initializes the debugger screen and windows but initially displays only the `COMMAND` window.
- 6) *SWDS*: Looks for the `dbinit.cmd` batch file.

Simulator: Looks for the `siminit.cmd` batch file.

The debugger searches for `dbinit.cmd` or `siminit.cmd` in directories named with `D_DIR`. If the debugger finds the file, it opens the file and reads and executes the commands it finds inside. The debugger expects this file to set up the memory map.

- 7) Loads any object filenames 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.

Debugger Messages

This appendix contains an alphabetical listing of the progress and error messages that the debugger might display in the COMMAND window display area. Each message contains both a description of the situation that causes the message and an action to take if the message indicates a problem or error.

Synopsis Page		Topic
<i>The main portion of this appendix is the alphabetical message reference.</i>	B.1 Alphabetical Reference of Debugger Messages	O-2
<i>These sections supplement the actions provided with error messages.</i>	B.2 Additional Instructions for Expression Errors	O-18
	B.3 Additional Instructions for Hardware Errors	O-18

B.1 Alphabetical Summary of Debugger Messages

Symbols

']' expected

Description This is an expression error—it means that the parameter contained an opening [symbol but didn't contain a closing] symbol.

Action See Section B.2 (page O-18).

(') expected

Description This is an expression error—it means that the parameter contained an opening (symbol but didn't contain a closing) symbol.

Action See Section B.2 (page O-18).

A

Aborted by user

Description The debugger halted a long COMMAND display listing (from WHATIS, DIR, ML, or BL) because you pressed the `(ESC)` key.

Action None required; this is normal debugger behavior.

B

Breakpoint already exists at address

Description During single-step execution, the debugger attempted to set a breakpoint where one already existed. (This isn't necessarily a breakpoint that you set—it may have been an internal breakpoint that was used for single-stepping).

Action None should be required; you may want to reset the program entry point (RESTART) and re-enter 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>	Enter a BL command to see where you have breakpoints set in your program. Use the BR command to delete all breakpoints, or use the BD command to delete individual unnecessary breakpoints.

C

Cannot allocate host memory

<i>Description</i>	This is a fatal error—it means that the debugger is running out of memory to run in.
<i>Action</i>	You might try invoking the debugger with the <code>-v</code> option so that fewer symbols may be loaded. Or you might want to relink your program and link in fewer modules at a time.

Corrupt call stack

<i>Description</i>	The debugger tried to update the CALLS window and couldn't. This may be because a function was called that didn't return. Or it could be that the call stack was overwritten in memory.
<i>Action</i>	If your program called a function that didn't return, then this is normal behavior (as long as you intended for the function not to return). Otherwise, you may be overwriting program memory.

Cannot change directory

<i>Description</i>	The directory name specified with the CD command either doesn't exist or is not in the current or auxiliary directories.
<i>Action</i>	Check the directory name that you specified. If this is really the directory that you want, re-enter the CD command and specify the entire pathname for that directory (for example, specify <code>C:\c2xh11</code> , not just <code>c2xh11</code>).

Cannot edit field

Description Expressions that are displayed in the WATCH window cannot be edited.

Action 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 ? or EVAL command to edit the actual symbol or register. The expression value will automatically be updated.

Cannot find/open initialization file

Description The debugger can't find the dbinit.cmd or siminit.cmd file.

Action Be sure that dbinit.cmd or siminit.cmd is in the appropriate directory. If it isn't, 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 *Setting Up the Debugger Environment* in the appropriate installation chapter.

Cannot halt the processor

Description This is a fatal error—for some reason, pressing **ESC** didn't halt program execution.

Action Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger again.

Cannot map port address

Description Attempt to do a connect/disconnect on an illegal port address.

Cannot open config file

Description The SCONFIG command can't find the screen-customization file that you specified.

Action Be sure that the filename was typed correctly. If it wasn't, re-enter the command with the correct name. If it was, re-enter the command and specify full path information with the filename.

Cannot open "filename"

Description The debugger attempted to show *filename* in the FILE window but could not find the file.

Action Be sure that the file exists as named. If it does, enter the USE command to identify the file's directory.

Cannot open object file: “filename”

Description The file specified with the LOAD, SLOAD, or RELOAD command is not an object file that the debugger can load.

Action Be sure that you're loading an actual object file. Be sure that the file was linked (you may want to run dspcl again to create an executable object file).

Cannot open new window

Description A maximum of 127 windows can be open at once. The last request to open a window would have made 128, which isn't possible.

Action Close any unnecessary windows. Windows that can be closed include WATCH, CALLS, and DISP. To close the WATCH window, enter WD. To close the CALLS window or a DISP window, make the desired window active and press **F4**.

Cannot read processor status

Description This is a fatal error—for some reason, pressing **ESC** didn't halt program execution.

Action Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger again.

Cannot reset the processor

Description This is a fatal error—for some reason, pressing **ESC** didn't halt program execution.

Action Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger again.

Cannot restart processor

Description If a program doesn't have an entry point, then RESTART won't reset the PC to the program entry point.

Action Don't use RESTART if your program doesn't have an explicit entry point.

Cannot set/verify breakpoint at *address*

Description Either you attempted to set a breakpoint in read-only or protected memory, or there are hardware problems with the target system or SWDS.

Action Check your memory map. If the address that you wanted to breakpoint wasn't in ROM, see Section B.3 (page O-18).

Cannot step (SWDS only)

Description The monitor software has been overwritten or damaged by program execution.

Action Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger.

Cannot take address of register

Description This is an expression error. C does not allow you to take the address of a register.

Action See Section B.2 (page O-18).

Command “*cmd*” not found

Description The debugger didn't recognize the command that you typed.

Action Re-enter the correct command. Refer to Chapter 12 or the Quick Reference Card for a list of valid debugger commands.

Command timed out (SWDS only)

Description The monitor software has been overwritten or damaged by program execution.

Action Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger.

Conflicting map range

Description A block of memory specified with the MA command overlaps an existing memory map entry. Blocks cannot overlap.

Action Use the ML command to list the existing memory map; this will help you find that existing block that the new block would overlap. If the existing block is not necessary, delete it with the MD command and re-enter the MA command. If the existing block is necessary, re-enter the MA command with parameters that will not overlap the existing block.

E**Error in expression**

Description This is an expression error.

Action See Section B.2 (page O-18).

Execution error (SWDS only)

Description The monitor software has been overwritten or damaged by program execution.

Action Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger.

F**File already tied to port**

Description Attempt to connect on an address that already has a file connected to it.

File does not exist

Description Port file could not be opened for reading.

Files must be disconnected from ports

Description Attempt to delete a memory map that has files connected to it.

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 wasn't, re-enter the FILE command with the correct name. If it was, re-enter the FILE command and specify full path information with the filename.

File not found : “filename”

Description 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.

Action Be sure that the filename was typed correctly. If it wasn't, re-enter the command with the correct name. If it was, re-enter the command and specify full path information with the filename.

File too large (filename)

Description You attempted to load a file that was more than 65,518 bytes long.

Action Try loading the file without the symbol table (SLOAD), or use gspl to relink the program with fewer modules.

Float not allowed

Description This is an expression error—a floating-point value was used invalidly.

Action See Section B.2 (page O-18).

Function required

Description The parameter for the FUNC command must be the name of a function in the program that is loaded.

Action Re-enter the FUNC command with a valid function name.



Illegal addressing mode

Description An illegal C2x addressing mode was encountered.

Illegal cast

Description This is an expression error—the expression parameter uses a cast that doesn't meet the C language rules for casts.

Action See Section B.2 (page O-18).

Illegal control transfer instruction

Description The instruction following a delayed branch/call instruction was modifying the program counter.

Illegal left hand side of assignment

Description This is an expression error—the left hand side of an assignment expression doesn't meet C language assignment rules.

Action See Section B.2 (page O-18).

Illegal memory access

Description Access to unconfigured/reserved/nonexistent memory.

Illegal opcode

Description An invalid C2x instruction was encountered.

Illegal operand of &

Description This is an expression error—the expression attempts to take the address of an item that doesn't have an address.

Action See Section B.2 (page O-18).

Illegal pointer math

Description This is an expression error—some types of pointer math are not valid in C expressions.

Action See Section B.2 (page O-18).

Illegal pointer subtraction

Description This is an expression error—the expression attempts to use pointers in a way that is not valid.

Action See Section B.2 (page O-18).

Illegal structure reference

Description 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.

Action See Section B.2 (page O-18).

Illegal use of structures

Description This is an expression error—the expression parameter is not using structures according to the C language rules.

Action See Section B.2 (page O-18).

Illegal use of void expression

Description This is an expression error—the expression parameter does not meet the C language rules.

Action See Section B.2 (page O-18).

Integer not allowed

Description This is an expression error—the command did not accept an integer as a parameter.

Action See Section B.2 (page O-18).

Invalid address

— Memory access outside valid range: *address*

Description The debugger attempted to access memory at *address*, which is outside the memory map.

Action Check your memory map to be sure that you access valid memory.

Invalid argument

Description One of the command parameters does not meet the requirements for the command.

Action Re-enter the command with valid parameters. Refer to the appropriate command description in Chapter 12.

Invalid attribute name

Description The COLOR and SCOLOR commands accept a specific set of area names for their first parameter. The parameter entered did not match one of the valid attributes.

Action Re-enter the COLOR or SCOLOR command with a valid area name parameter. Valid area names are listed in Table 11-2 (page 11-3).

Invalid color name

Description The COLOR and SCOLOR commands accept a specific set of color attributes as parameters. The parameter entered did not match one of the valid attributes.

Action Re-enter the COLOR or SCOLOR command with a valid color parameter. Valid color attributes are listed in Table 11–1 (page 11-2).

Invalid memory attribute

Description The third parameter of the MA command specifies the type, or attribute, of the block of memory that MA adds to the memory map. The parameter entered did not match one of the valid attributes.

Action Re-enter the MA command. Use one of the following valid parameters to identify the memory type:

R, ROM, READONLY	(read-only memory)
W, WOM, WRITEONLY	(write-only memory)
RW, RAM	(read/write memory)
PROTECT	(no-access memory)
OPORT	(I/O memory, simulator only)
IPOINT	(I/O memory, simulator only)
IOPORT	(I/O memory, simulator only)

Invalid object file

Description Either the file specified with the LOAD, SLOAD, or RELOAD command is not an object file that the debugger can load, or it has been corrupted.

Action Be sure that you're loading an actual object file. Be sure that the file was linked (you may want to run dspcl 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 dspcl.

Invalid watch delete


Description The debugger can't delete the parameter supplied with the WD command. Usually, this is because the watch index doesn't exist or because a symbol name was typed in instead of a watch index.

Action Re-enter the WD command. Be sure to specify the watch index that matches the item you'd like to delete (this is the number in the left column of the WATCH window). Remember, you can't delete items symbolically—you must delete them by number.

Invalid window position

Description The debugger can't move the active 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.

Action You can use the mouse to move the window.


If you don't have a mouse, enter the MOVE command without parameters; then use the arrow keys to move the window. When you're finished, you *must* press **ESC** or .

If you prefer to use the MOVE command with parameters, refer to Table 5-2 (page 5-23) for a list of the XY limits. The minimum XY position is 0,1; the maximum position depends on which screen size you're using.

Invalid window size

Description 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.

Action You can use the mouse to size the window.

If you don't have a mouse, enter the SIZE command without parameters; then use the arrow keys to move the window. When you're finished, you *must* press **ESC** or .

If you prefer to use the SIZE command with parameters, refer to Table 5-1 (page 5-21) for a list of valid sizes. The minimum size is 4 by 3; the maximum size depends on which screen size you're using.

L

Load aborted

Description This message always follows another message.

Action Refer to the message that preceded *Load aborted*.

Lost power (or cable disconnected)

Description Either the target cable is disconnected, or the target system is faulty.

Action Check the target cable connections. If the target seems to be connected correctly, see Section B.3 (page O-18).

Lost processor clock

Description Either the target cable is disconnected, or the target system is faulty.

Action Check the target cable connections. If the target seems to be connected correctly, see Section B.3 (page O-18).

Lval required

Description This is an expression error—an assignment expression was entered that requires a legal lefthand side.

Action See Section B.2 (page O-18).

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 wasn't, re-enter the command with the correct name. If it was, then be sure that the associated object file is loaded.

If *name* was some other type of parameter, refer to the command's description for a list of valid parameters.

Nonrepeatable instruction

Description The instruction following the RPT instruction is not a repeatable instruction.

M

Memory access error at *address*

Description Either the processor is receiving a bus fault or there are problems with target system memory.

Action See Section B.3 (page O-18).

Memory map table full

Description Too many blocks have been added to the memory map. This will rarely happen unless someone is adding blocks word by word (which is inadvisable).

Action Stop adding blocks to the memory map. Consolidate any adjacent blocks that have the same memory attributes.

P

Pointer not allowed

Description This is an expression error.

Action See Section B.2 (page O-18).

Processor is already running

Description One of the RUN commands was entered while the debugger was running free from the target system.

Action Enter the HALT command to stop the free run, then re-enter the desired RUN command.

R**Read not allowed for port**

Description There was an attempt to connect a file for input operation to an address that is not configured for read.

Register access error

Description Either the processor is receiving a bus fault, or there are problems with target-system memory.

Action See Section B.3 (page O-18).

S**Specified map not found**

Description The MD command was entered with an address or block that is not in the memory map.

Action Use the ML command to verify the current memory map. When using MD, it is possible to specify only the first address of a defined block.

Structure member not found

Description This is an expression error—an expression references a non-existent structure member.

Action See Section B.2 (page O-18).

Structure member name required

Description This is an expression error—a symbol name followed by a period but no member name.

Action See Section B.2 (page O-18).

Structure not allowed

Description This is an expression error—the expression is attempting an operation that cannot be performed on a structure.

Action See Section B.2 (page O-18).

T

Take file stack too deep

Description Batch files can be nested up to 10 levels deep. Batch files can call other batch files, which can call other batch files, and so on. Apparently, the batch file that you are TAKEing calls batch files that are nested more than 10 levels.

Action Edit the batch file that caused the error. Instead of calling another batch file from within the offending file, you may want to copy the contents of the second file into the first. This will remove a level of nesting.

Too many breakpoints

Description 200 breakpoints are already set, and there was an attempt to set another. Note that 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.

Action Enter a BL command to see where you have breakpoints set in your program. Use the BR command to delete all breakpoints or use the BD command to delete individual unnecessary breakpoints.

Too many paths

Description More than 20 paths have been specified cumulatively with the USE command, D_SRC environment variable, and -i debugger option.

Action If you are entering the USE command before entering another command that has a *filename* parameter, don't enter the USE command. Instead, enter the second command and specify full path information for the *filename*.

W**Window not found**

Description The parameter supplied for the WIN command is not a valid window name.

Action Re-enter the WIN command. Remember that window names must be typed in uppercase letters. Here are the valid window names; the bold letters show the smallest acceptable abbreviations:

CALLS	CPU	DISP	MEMORY
COMMAND	DISASSEMBLY	FILE	WATCH

Write not allowed for port

Description There was an attempt to connect a file for output operation to an address that is not configured for write.

U**Undeclared port address**

Description There was an attempt to do a connect/disconnect on an address that isn't declared as a port.

User halt

Description The debugger halted program execution because you pressed the **ESC** key.

Action None required; this is normal debugger behavior.

B.2 Additional Instructions for Expression Errors

Whenever you receive an expression error, you should re-enter 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.

B.3 Additional Instructions for Hardware Errors

If you continue to receive the messages that send you to this section, this indicates persistent hardware problems.

- SWDS:** Your program may be overwriting the monitor software installed on the board.

Registers and Pseudoregisters

While using the 'C2x debugger, you can display the registers and pseudoregisters listed in this appendix. For more information about these registers, refer to the *TMS320C2x User's Guide* (literature number SPRU014B).

Register Acronym	Size (in bits)	Description
PC	16	Program counter
ST0	16	Status register 0
ST1	16	Status register 1
ACC	32	Accumulator
ACCL	16	Accumulator low word
ACCH	16	Accumulator high word
PREG	32	Product register
PLR	16	Product register low word
PHR	16	Product register high word
TREG	16	Temporary register
AR0	16	Auxiliary register 0
AR1	16	Auxiliary register 1
AR2	16	Auxiliary register 2
AR3	16	Auxiliary register 3
AR4	16	Auxiliary register 4
AR5	16	Auxiliary register 5
AR6	16	Auxiliary register 6
AR7	16	Auxiliary register 7
DRR	16	Serial port data receive register
DXR	16	Serial port data transmit register

Register Acronym	Size (in bits)	Description
TIM	16	Time register
PRD	16	Period register
IMR	16	Interrupt mask register
GREG	16	Global memory allocation register
IFR	16	Interrupt flag register
RPTC	16	Repeat counter
TOS	16	Top of stack
MPMC	16	pseudoregister for determining whether the simulator is in the microprocessor or microcomputer mode.
CLK	32	Clock pseudoregister for benchmarking (simulator only)
BIO	16	Branch control input pin (simulator only)
RIRT	16	Receive interrupt timer register
RIRP	16	Receive interrupt period register
XIRT	16	Transmit interrupt timer register
XIRP	16	Transmit interrupt period register
STK(0-7)	16	'C2x hardware stack pseudoregisters

Glossary

A

active window: 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, where a variable is composed of multiple variables, called members.

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 doesn't 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 shown 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: Either of two different types of files. One type of batch file contains DOS commands for the PC to execute. A second type of batch file contains debugger commands for the debugger to execute. The PC doesn't execute debugger batch files, and the debugger doesn't 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.

breakpoint: A point within your program where execution will halt because of a previous request from you.

C

c2xreset: A utility that resets the SWDS.

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.

children: Additional windows opened for aggregate types that are members of a parent aggregate type displayed in the original DISP window.

click: To press and release a mouse button without moving the whole mouse.

CLK: A pseudoregister that shows the number of CPU cycles consumed during benchmarking. The value in CLK is valid only after entering a RUNB command but before entering another RUN command.

code-display windows: Windows that show code, text files, or code-specific information. This category includes the DISASSEMBLY, FILES, and CALLS windows.

COFF: *Common Object File Format*. An implementation of the object file format of the same name developed by AT&T. The TMS320 fixed-point DSP compiler, assembler, and linker use and generate COFF files.

command line: The portion of the COMMAND window where you can enter commands.

command-line cursor: Block-shaped cursor that identifies the current character position on the command line.

COMMAND window: A window for you to enter commands and for the debugger to echo command entry, show command output, and list progress or error messages.

CPU window: Displays the contents of 'C2x on-chip registers, including the program counter, status register, and auxiliary registers.

current-field cursor: A screen icon that identifies the current field in the active window.

cursor: An icon on the screen (such as a rectangle or a horizontal line) that is used as a pointing device. The cursor is usually under mouse or keyboard control.

D

data-display windows: Windows for observing and modifying various types of data. This category includes the MEMORY, CPU, DISP, and WATCH windows.

D_DIR: An environment variable that identifies the directory containing the commands and files necessary for running the debugger.

debugger: A window-oriented software interface that helps you to debug 'C2x programs running on the SWDS, emulator, or simulator.

disassembly: The reverse-assembly of the contents of memory to form assembly language code.

DISASSEMBLY window: A window that displays the disassembly of memory contents.

DISP window: A window that displays the members of an aggregate data type.

display area: The portion of the COMMAND window where the debugger echoes command entry, shows command output, and lists progress or error messages.

D_OPTIONS: An environment variable that you can use for identifying often-used debugger options.

drag: To move the mouse while pressing one of the mouse buttons.

dspcl: A shell utility that invokes the TMS320 fixed-point DSP compiler, assembler, and linker to create an executable object file version of your program.

D_SRC: An environment variable that identifies directories containing program source files.

E

EGA: *Enhanced Graphics Adaptor*. An industry standard for video cards.

EISA: *Extended Industry Standard Architecture*. A standard for PC buses.

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 primarily intended for displaying C code but can be used to display any text file.

I

initdb.bat: A batch file created to contain DOS commands for setting up the debugger environment.

I/O switches: Hardware switches on the SWDS board that identify the PC I/O memory space used for SWDS–debugger communications.

ISA: *Industry Standard Architecture*. A subset of the EISA standard.

M

memory map: A map of memory space that tells the debugger which areas of memory can and can't 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.

mouse cursor: Block-shaped cursor that tracks mouse movements over the entire display.

P

PC: Personal computer or program counter, depending on the context and where it's used in this book: 1) In installation instructions or information relating to hardware and boards, *PC* means *Personal Computer* (as in IBM PC). 2) In general debugger and program-related information, *PC* means *Program Counter*, which is the register that identifies the current statement in your program.

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.

S

scalar type: A C type in which the variable is a single variable itself, not composed of other variables.

scrolling: A method of moving the contents of a window up, down, left, or right to view contents that weren't shown.

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 software program that simulates 'C2x operation, providing a low-cost method of testing 'C2x applications without target hardware.

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.

SWDS: *Software Development System.* A PC-compatible plug-in board that provides a low-cost method of program evaluation and development.

symbol table: A file that contains the names of all variables and functions in your 'C2x program.

T

TMS320C25: A 100-ns, fixed-point, CMOS digital signal processor, capable of executing 10 million instructions per second. It is pin-for-pin and object-code upward compatible with the TMS32020.

TMS320C26: A 100-ns, fixed-point, CMOS digital signal processor, capable of executing 10 million instructions per second. It is pin-for-pin and object-code upward compatible with the TMS320C25, except for the RAM configuration instructions.

V

VGA: *Video Graphics Array.* An industry standard for video cards.

W

WATCH window: A window that displays the values of selected expressions, symbols, addresses, and registers.

window: A defined rectangular area of virtual space on the display.



TMS320C2x C Source Debugger Reference Card

Phone Numbers

TI Customer Response Center
(CRC) Hotline: (800) 232-3200

DSP Hotline: (713) 274-2320

Invoking the Debugger

SWDS:	db2x	[filename]	[-options]
Simulator:	sim2x	[filename]	[-options]

Debugger Options

Option	Description																		
-b[bbbb]	Screen size options (PCs only). <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Option</th> <th>Chars./Lines</th> <th>Notes</th> </tr> </thead> <tbody> <tr> <td>none</td> <td>80 × 25</td> <td>Default display</td> </tr> <tr> <td>-b</td> <td>80 × 39[†] 80 × 43[‡] 80 × 50[§]</td> <td></td> </tr> <tr> <td>-bb</td> <td>120 × 43</td> <td rowspan="4">Supported on PCs with a Video Seven VEGA Deluxe™ card</td> </tr> <tr> <td>-bbb</td> <td>132 × 43</td> </tr> <tr> <td>-bbbb</td> <td>80 × 60</td> </tr> <tr> <td>-bbbbb</td> <td>100 × 60</td> </tr> </tbody> </table>	Option	Chars./Lines	Notes	none	80 × 25	Default display	-b	80 × 39 [†] 80 × 43 [‡] 80 × 50 [§]		-bb	120 × 43	Supported on PCs with a Video Seven VEGA Deluxe™ card	-bbb	132 × 43	-bbbb	80 × 60	-bbbbb	100 × 60
Option	Chars./Lines	Notes																	
none	80 × 25	Default display																	
-b	80 × 39 [†] 80 × 43 [‡] 80 × 50 [§]																		
-bb	120 × 43	Supported on PCs with a Video Seven VEGA Deluxe™ card																	
-bbb	132 × 43																		
-bbbb	80 × 60																		
-bbbbb	100 × 60																		
-c	Sets memory reserved for uninitialized data to all zeros.																		
-i <i>pathname</i>	Identifies additional directories that contain source files.																		
-mv25 -mv26	Simulator only. Identifies '25, or '26 memory map ('25 is the default).																		
-p <i>memory segment</i>	SWDS only. You must use -p to identify the correct jumper settings (P1-P4).																		

[†] PC version of simulator running under Microsoft Windows

[‡] PC with EGA card; Sun

[§] PC with VGA card

Debugger Options (continued)

Option	Description
-s	Tells the debugger to load <i>filename's</i> symbol table only.
-t <i>filename</i>	Allows you to specify an initialization file other than <i>dbinit.cmd</i> or <i>siminit.cmd</i> .
-v	Loads only global symbols; later, local symbols are loaded as needed. Affects all loads.
-x	Ignores options supplied with <code>D_OPTIONS</code> .

Summary of Debugger Commands

? <i>expression</i> [@prog @data]
addr <i>address</i> [@prog @data] addr <i>function name</i>
asm
ba <i>address</i>
bd <i>address</i>
bl
border [<i>active</i>] [[, <i>inactive</i>] [, <i>resize</i>]]
br
c
calls
cd <i>directory name</i> chdir <i>directory name</i>
cls
cnext [<i>expression</i>]
color <i>area</i> , <i>attr₁</i> [, <i>attr₂</i> [, <i>attr₃</i> [, <i>attr₄</i>]]]
cstep [<i>expression</i>]
dasm <i>address</i> [@prog @data] dasm <i>function name</i>
dir [<i>directory</i>]
disp <i>expression</i> [@prog @data]
eval <i>expression</i> [@prog @data] e <i>expression</i> [@prog @data]
file <i>filename</i>
fill <i>address</i> , <i>page</i> , <i>length</i> , <i>data</i>
func <i>function name</i> func <i>address</i>
go [<i>address</i>]

Summary of Debugger Commands (continued)

load <i>object filename</i>
ma <i>address, page, length, type</i>
map { <i>on</i> <i>off</i> }
mc <i>port address, page, filename, {READ WRITE} †</i>
md <i>address, page</i>
mi <i>port address, page, {READ WRITE} †</i>
ml
mem <i>expression</i> [@prog @data]
mix
move [<i>X, Y</i> [, <i>width, length</i>]]
mr
ms <i>address, page, length, filename</i>
next [<i>expression</i>]
prompt <i>new prompt</i>
quit
reload <i>object filename</i>
reset
restart
rest
return
ret
run [<i>expression</i>]
runb [†]
runf [†]
scolor <i>area, attr₁ [, attr₂ [, attr₃ [, attr₄]]]</i>
sconfig [<i>filename</i>]
size [<i>width, length</i>]
sload <i>object filename</i>
ssave [<i>filename</i>]
step [<i>expression</i>]
take <i>filename</i> [, <i>flag</i>]
use <i>directory name</i>
wa <i>expression</i> [@prog @data] [, <i>label</i>]
wd <i>index number</i>
whatis <i>symbol</i>
win <i>WINDOW NAME</i>
wr

[†] SWDS only

[‡] Simulator only

**Border Styles
(BORDER Command)**

Index	Style
0	Double-lined box
1	Single-lined box
2	Solid 1/2-tone top, double-lined sides/bottom
3	Solid 1/4-tone top, double-lined sides/bottom
4	Solid box, thin border
5	Solid box, heavy sides, thin top/bottom
6	Solid box, heavy borders
7	Solid 1/2-tone box
8	Solid 1/4-tone box

**Colors and Attributes
(COLOR/SCOLOR Commands)**

black	blue	green	cyan
red	magenta	yellow	white
bright		blink	

**Area Names
(COLOR/SCOLOR Commands)**

menu_bar	menu_border	menu_entry	menu_cmd
menu_hilite	menu_hicmd	win_border	win_hiborder
win_resize	field_text	field_hilite	field_edit
field_label	field_error	cmd_prompt	cmd_input
cmd_cursor	cmd_echo	asm_data	asm_cdata
asm_label	asm_clabel	background	blanks
error_msg	file_line	file_eof	file_text
file_brk	file_pc	file_pc_brk	

**Window Size and Position Limits
(SIZE and MOVE Commands)**

Screen size	Option	Valid widths	Valid lengths	Valid X pos.	Valid Y pos.
80x25	none	4-80	3-24	0-76	1-22
80x39 [†]	-b	4-80	3-38	0-76	1-36
80x43 [‡]			3-42		1-40
80x50 [§]			3-49		1-47
120x43	-bb	4-120	3-42	0-116	1-40
132x43	-bbb	4-132	3-42	0-128	1-40
80x60	-bbbb	4-80	3-59	0-76	1-57
100x60	-bbbbb	4-100	3-59	0-106	1-57

[†] PC version of simulator running under Microsoft Windows

[‡] PC with EGA card; Sun

[§] PC with VGA card

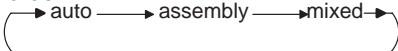
Memory Types

To identify this kind of memory	Use this keyword as the <i>type</i> parameter
read-only memory	R , ROM , or READONLY
write-only memory	W , WOM , or WRITEONLY
read/write memory	RW or RAM
no-access memory	PROTECT
input port	IPOINT or IN PORT ‡
output port	OPOINT or OUT PORT ‡
input/output port	IOPORT ‡

Page Types

To identify this page	Use this 1-digit <i>page</i> parameter
program memory	0
data memory	1
I/O space	2

Switching Modes

To do this	Use this function key
Switch debugging modes in this order: 	F3

Running Code










To do this	Use these function keys
Run code from the current PC	F5
Single-step from the current PC	F8
Single-step code from the current PC; step over function calls	F10

Selecting or Closing a Window




To do this	Use these function keys
Select the active window	F6
Close the CALLS or DISP window	F4

‡ Simulator only


Editing Text on the Command Line

To do this	Use these function keys
Enter the current command	
Move back over text without erasing characters	  or 
Move forward through text without erasing characters	 
Move back over text while erasing characters	
Move forward through text while erasing characters	
Insert text into the characters that are already on the command line	


Using the Command History

To do this	Use these function keys
Move backward, one command at a time, through the command history	
Move forward, one command at a time, through the command history	 















Editing Data or Selecting the Active Field

To do this	Use this function key
<input type="checkbox"/> <i>FILE or DISASSEMBLY window:</i> Set or clear a breakpoint	
<input type="checkbox"/> <i>CALLS window:</i> Display the source to a listed function	
<input type="checkbox"/> <i>Any data-display window:</i> Edit the contents of the current field	
<input type="checkbox"/> <i>DISP window:</i> Open an additional DISP window	

Halting or Escaping From an Action





To do this	Use this function key
<input type="checkbox"/> Halt program execution	
<input type="checkbox"/> Close a pulldown menu	
<input type="checkbox"/> Undo an edit of the active field in a data-display window	
<input type="checkbox"/> Halt the display of a long list of data	

Displaying Pulldown Menus













To do this	Use these function keys
Display the Load menu	 
Display the Break menu	 
Display the Watch menu	 
Display the Memory menu	 
Display the Color menu	 
Display the MoDe menu	 
Display an adjacent menu	 or 
Execute any of the choices from a displayed pulldown menu	Press the high-lighted letter corresponding to your choice

Moving or Sizing a Window

Enter the MOVE or SIZE command without parameters, then use the arrow keys:

To do this	Use these function keys
<input type="checkbox"/> Move the window down one line	
<input type="checkbox"/> Make the window one line longer	
<input type="checkbox"/> Move the window up one line	
<input type="checkbox"/> Make the window one line shorter	
<input type="checkbox"/> Move the window left one character position	
<input type="checkbox"/> Make the window one character narrower	
<input type="checkbox"/> Move the window right one character position	
<input type="checkbox"/> Make the window one character wider	

Scrolling the Active Window's Contents

To do this	Use these function keys
Scroll up through the window contents, one window length at a time	
Scroll down through the window contents, one window length at a time	
Move the field cursor up one line at a time	
Move the field cursor down one line at a time	
<input type="checkbox"/> <i>FILE window only:</i> Scroll left 8 characters at a time	
<input type="checkbox"/> <i>Other windows:</i> Move the field cursor left 1 field; at the first field on a line, wrap back to the last fully displayed field on the previous line	
<input type="checkbox"/> <i>FILE window only:</i> Scroll right 8 characters at a time	
<input type="checkbox"/> <i>Other windows:</i> Move the field cursor right 1 field; at the last field on a line, wrap around to the first field on the next line	
<i>FILE window only:</i> Adjust the window's contents so that the first line of the text file is at the top of the window	
<i>FILE window only:</i> Adjust the window's contents so that the last line of the text file is at the bottom of the window	
<i>DISP windows only:</i> Scroll up through an array of structures	 
<i>DISP windows only:</i> Scroll down through an array of structures	 

? command, 3-17, 9-3, 12-7, 12-34
 modifying PC, 8-10
 side effects, 9-5

A

absolute addresses, 9-7, 10-3
active window, 5-17—5-19
 breakpoints, 10-3
 current field, 3-7, 5-16
 customizing its appearance, 11-4
 default appearance, 5-17
 effects on command entry, 6-3
 identifying, 3-7, 5-17
 selecting, 5-18, 12-35
 function key method, 3-7, 5-18, 12-38
 mouse method, 3-7, 5-18
 WIN command, 3-6, 5-19, 12-35
ADDR command, 5-7, 5-8, 8-5, 8-7, 12-7
addresses
 absolute addresses, 9-7, 10-3
 accessible locations, 7-1
 contents of (indirection), 9-8
 data memory notation, 3-6
 hexadecimal notation, 9-7
 I/O address space, simulator, 7-10—7-14
 in MEMORY window, 3-6, 9-7
 nonexistent locations, 7-2
 pointers in DISP window, 3-23
 program memory notation, 3-6
 symbolic addresses, 9-7
aggregate types, displaying, 3-22, 5-14, 9-12—9-14
ANSI C, 4-6
archiver, 4-7
area names (for customizing the display)
 code-display windows, 11-5
 COMMAND window, 11-4
 common display areas, 11-3
 data-display windows, 11-6
 menus, 11-7
 summary of valid names, 11-3
 window borders, 11-4

arithmetic operators, 13-2
arrays
 displaying/modifying contents, 9-12
 format in DISP window, 3-23, 9-13, 12-14
 member operators, 13-2
arrow keys
 editing, 9-4
 moving a window, 3-9, 5-24, 12-38
 scrolling, 5-26, 12-39
 sizing a window, 3-8, 5-22, 12-38
ASM command, 3-13, 8-3, 12-8
 pulldown selection, 6-12, 8-3
assembler, 1-3, 2-2, 2-8, 2-11, 4-7, 4-8
assembly language code, displaying, 5-2, 5-3, 8-4
assembly mode, 3-12, 5-3
 ASM command, 8-3, 12-8
 selection, 8-3
assignment operators, 9-5, 13-3
attributes, 11-2
auto mode, 3-12, 5-2—5-3
 C command, 8-3, 12-10
 selection, 8-3
autoexec.bat
 invoking, 1-10, 2-5
 sample
 simulator, 2-4
 SWDS, 1-10
 simulator, 2-4—2-14
 SWDS, 1-9—1-16
auxiliary registers, 9-11

B

-b debugger option
 effect on window positions, 5-23
 effect on window sizes, 5-21
 simulator, 2-13
 SWDS, 1-13
 with D_OPTIONS environment variable
 simulator, 2-6
 SWDS, 1-12
BA command, 10-3, 12-8

- pull-down selection, 6-11
- background, 11-3
- batch files, 6-13
 - autoexec.bat
 - simulator*, 2-4—2-14
 - SWDS*, 1-9—1-16
 - clrs.dat, 2-8, 11-9
 - dbinit.cmd, 1-9, A-1
 - SWDS*, 1-3
 - displaying, 8-7
 - execution, 12-33
 - halting execution, 6-13
 - init.clr, 11-9
 - SWDS*, 1-3
 - initdb.bat
 - simulator*, 2-4—2-14
 - SWDS*, 1-9—1-16
 - invoking
 - autoexec.bat*
 - simulator*, 2-5
 - SWDS*, 1-10
 - initdb.bat*
 - simulator*, 2-5
 - SWDS*, 1-10
 - mem.map, 7-9
 - memory maps, 7-9
 - mono.clr, 11-9
 - SWDS*, 1-3
 - siminit.cmd, A-1
 - simulator*
 - PC systems, 2-2
 - SUN systems, 2-11
 - VMS version, 2-8
 - TAKE command, 6-13, 7-9, 12-33
- BD command, 10-4, 12-8
 - pull-down selection, 6-11
- benchmarking, 3-17, 8-17
 - simulator* constraints, 8-17
 - SWDS*, 8-18
- BIO pseudoregister, 9-17
- bitwise operators, 13-3
- BL command, 10-5, 12-9
 - pull-down selection, 6-11
- blanks, 11-3
- BORDER command, 11-8, 12-9
 - pull-down selection, 6-12
- borders
 - colors, 11-4
 - styles, 11-8

- BR command, 3-17, 10-4, 12-10
 - pull-down selection, 6-11
- breakpoints, 10-1—10-6
 - active window, 3-7
 - adding, 12-8
 - function key method*, 10-3, 12-39
 - mouse method*, 10-3
 - with commands*, 10-3
 - benchmarking with RUNB, 3-17, 8-17
 - clearing, 3-17, 10-4, 12-8, 12-10
 - function key method*, 10-4, 12-39
 - mouse method*, 10-4
 - with commands*, 10-4
 - commands
 - BA* command, 10-3, 12-8
 - BD* command, 10-4, 12-8
 - BL* command, 10-5, 12-9
 - BR* command, 3-17, 10-4, 12-10
 - listing set breakpoints, 10-5, 12-9
 - pull-down menu, 6-11
 - setting, 3-15, 3-17, 10-2
 - function key method*, 10-3, 12-39
 - mouse method*, 10-3
 - with commands*, 10-3

C

- C command, 3-13, 8-3, 12-10
 - pull-down selection, 6-12, 8-3
- C source
 - displaying, 3-11, 8-4, 12-15
 - managing memory data, 9-8
- c2xhll directory, 2-3, 2-5
- c2xhll directory, 1-9, 1-11
- c2xreset, 1-3
- CALLS command, 5-9, 5-10, 12-10
- CALLS window, 3-11, 5-9, 8-7
 - closing, 5-10, 5-28, 12-38
 - opening, 5-10, 12-10
- casting, 3-25, 9-8, 13-4
- CHDIR (CD) command, 3-22, 6-14, 8-9, 12-10
- clearing the display area, 3-22, 6-5, 12-11
- “click and type” editing, 5-27, 9-4
- CLK pseudoregister, 3-17, 8-17
- closing
 - a window, 5-28
 - CALLS window, 5-10, 5-28, 12-38
 - debugger, 1-15, 2-14, 12-25
 - DISP window, 3-24, 5-28, 9-14, 12-38

- WATCH window, 5-28, 9-16, 12-35
- clrs.dat, 2-8, 11-9
- CLS command, 3-22, 6-5, 12-11
- CNEXT command, 8-13, 12-11
- code-display windows, 5-5, 8-2
 - CALLS window, 5-9, 8-2, 8-7
 - DISASSEMBLY window, 3-6, 5-7, 8-2
 - effect of debugging modes, 8-2
 - FILE window, 5-8, 8-2
- COLOR command, 11-2, 12-12
- color.clr, 11-9
- colors, 11-2
 - area names, 11-3—11-7
- comma operator, 13-4
- command history, 6-4
 - function key summary, 12-36
- command line, 5-6, 6-2
 - changing the prompt, 11-12, 12-25
 - cursor, 5-16
 - customizing its appearance*, 11-4, 11-12
 - editing, 6-3
 - function key summary*, 12-36
- COMMAND window, 5-5, 5-6, 6-2
 - colors, 11-4
 - command line, 3-5, 6-2
 - editing keys*, 12-36
 - customizing, 11-4
 - display area, 3-5, 6-2
 - clearing*, 12-11
- commands
 - alphabetical summary, 12-7—12-35
 - batch files, 6-13
 - breakpoint commands, 10-1—10-6, 12-5
 - code-execution (run) commands, 8-10, 12-6
 - command line, 6-2
 - data-management commands, 9-2—9-18, 12-4
 - entering and using, 6-1—6-14
 - file-display commands, 8-4, 12-4
 - load commands, 8-8, 12-4
 - memory commands, 7-6—7-14
 - memory-map commands, 12-5
 - mode commands, 8-2, 12-3
 - pull-down menus, 6-6, 6-11
 - screen-customization commands, 11-1, 12-5
 - system commands, 6-14, 12-3
 - window commands, 5-19, 12-3
- compiler, 1-3, 2-2, 2-8, 2-11, 4-6, 4-8
- constraints
 - CLK, 8-17
 - simulator, benchmarking, 8-17
 - control bits
 - FO, 7-11, 7-13, 12-20
 - SPC, 7-11, 7-13, 12-20
 - TDM, 7-11, 7-13, 12-20
 - TSPC, 7-11, 7-13, 12-20
 - CPU window, 5-13, 9-2, 9-11
 - colors, 11-6
 - customizing, 11-6
 - CSTEP command, 3-19, 8-13, 12-13
 - current directory, changing, 6-14, 8-9, 12-10
 - current field
 - cursor, 5-16
 - dialog box, 6-4
 - editing, 9-4—9-5
 - current PC, 3-5, 5-7
 - finding, 8-10
 - selecting, 8-10
 - cursors, 5-16
 - command-line cursor, 5-16
 - current-field cursor, 5-16
 - mouse cursor, 5-16
 - customizing the display, 11-1—11-12
 - changing the prompt, 11-12
 - clrs.dat, 2-8
 - colors, 11-2—11-7
 - init.clr, SWDS, 1-3
 - loading a custom display, 11-10, 12-30
 - mono.clr, SWDS, 1-3
 - saving a custom display, 11-10, 12-32
 - window border styles, 11-8

D

- DASM command, 5-7, 8-5, 12-13
- data-display windows, 3-22, 5-5, 9-2
 - colors, 11-6
 - CPU window, 5-13, 9-2, 9-11
 - DISP window, 5-14, 9-2, 9-12
 - MEMORY window, 3-6, 5-11, 9-2, 9-6
 - WATCH window, 3-18, 5-15, 9-2, 9-15
- data-management commands, 3-20, 3-22, 9-2
 - ? command, 3-17, 9-3, 12-7, 12-34
 - controlling data format, 3-25, 9-8
 - DISP command, 9-12, 12-14
 - EVAL command, 9-3, 12-15
 - FILL, 12-16
 - FILL command, 9-10

- MEM command, 3-6, 9-6, 12-22
- MS command, 9-9, 12-24
- side effects, 9-5
- WA command, 3-18, 9-15, 12-34
- WD command, 9-16, 12-34
- WHATIS command, 3-21, 9-2, 12-34
- WR command, 3-20, 9-16, 12-35
- data memory
 - adding to memory map, 7-6, 12-18
 - deleting from memory map, 7-8, 12-20
 - filling, 9-10, 12-16
 - saving, 9-9, 12-24
- db2x command, 3-4
 - options
 - b*, 1-13
 - c*, 1-13
 - D_OPTIONS* environment variable, 1-12
 - i*, 1-13
 - p*, 1-14
 - s*, 1-14
 - t*, 1-14
 - v*, 1-14
 - z*, 1-14
- db2x command, 8-8, 8-9
- dbinit.cmd, 1-9, A-1
 - SWDS, 1-3
- D_DIR environment variable, 6-13, 11-10, 12-30
 - effects on debugger invocation, A-1
 - simulator, 2-5
 - SWDS, 1-11
- debugger
 - description, 4-2
 - environment setup
 - simulator (PC systems)*, 2-4—2-7
 - SWDS*, 1-9—1-12
 - installation
 - simulator*, 2-1—2-14
 - PC systems, 2-2—2-7
 - SUN systems, 2-11—2-12
 - VMS systems, 2-8—2-10
 - SWDS*, 1-4—1-8
 - invocation, 3-4
 - simulator*, 2-13—2-14
 - SWDS*, 1-13—1-14
 - task ordering*, A-1
 - key features, 4-3—4-4
 - messages, B-1—B-18
 - SWDS version, 1-1—1-16
 - environment setup*, 1-9
 - installation*, 1-4
 - invocation*, 1-13
- debugging modes, 3-12—3-13, 8-3
 - assembly mode, 5-3
 - auto mode, 5-2
 - default mode, 5-2, 8-2
 - mixed mode, 5-4
 - pulldown menu, 3-13, 8-3
 - restrictions, 5-4
 - selection, 3-12
 - commands*, 8-3
 - function key method*, 8-3, 12-37
 - mouse method*, 8-3
- decrement operator, 13-3
- default
 - debugging mode, 5-2, 8-2
 - display, 3-5, 5-2, 8-2, 11-11
 - memory map, 7-3
 - simulator*
 - PC systems, 2-2
 - SUN systems, 2-11
 - VMS version, 2-8
 - SWDS*, 1-3
 - screen configuration file, 11-9
 - monochrome displays*, 1-3, 11-9
 - simulator, VAX version*, 2-8
 - SWDS*, 1-3
- dialog boxes, 6-9
- DIR command, 3-22, 6-14, 12-14
- directories
 - c2xhll directory, 2-3, 2-5
 - c2xhll directory, 1-9, 1-11
 - changing current directory, 6-14, 12-10
 - for auxiliary files
 - simulator*, 2-5
 - SWDS*, 1-11
 - for debugger software, 1-9, 2-3
 - simulator*, 2-5
 - SWDS*, 1-11
 - identifying additional source directories, 12-33
 - simulator*, 2-6
 - SWDS*, 1-11
 - USE command*, 12-33
 - identifying current directory, 8-9
 - listing contents of current directory, 6-14, 12-14
 - relative pathnames, 6-14, 12-10
 - search algorithm, 6-13, 8-9, A-1
- DISASSEMBLY window, 3-6, 5-7
 - colors, 11-5
 - customizing, 11-5
 - modifying display, 12-13

DISP command, 3-22, 5-14, 9-12, 12-14

DISP window, 3-22, 5-14, 9-2, 9-12

- closing, 3-24, 5-28, 9-14
- colors, 11-6
- customizing, 11-6
- identifying arrays, structures, pointers, 12-14
- opening, 9-12
- opening another DISP window, 9-13
 - function key method, 3-24, 9-13, 12-39*
 - mouse method, 3-23, 9-13*
 - with DISP command, 9-13*

display area, 5-6

- clearing, 3-22, 6-5, 12-11

display format

- enumerated types, 5-14
- floating-point values, 5-14
- integers, 5-14
- pointers, 5-14

display requirements

- simulator
 - PC version, 2-2*
 - SUN version, 2-11*
 - VAX version, 2-8*
- SWDS, 1-2

displaying

- assembly language code, 8-4
- batch files, 8-7
- C code, 8-6
- source programs, 8-4
- text files, 8-7

D_OPTIONS environment variable

- effects on debugger invocation, A-1
- simulator, 2-6
- SWDS, 1-12

DOS, setting up debugger environment

- simulator, 2-4
- SWDS, 1-10

DRR register, 7-11, 7-13, 12-20

dspcl shell, 4-9

D_SRC environment variable, 8-9

- effects on debugger invocation, A-1
- simulator, 2-6
- SWDS, 1-11

DXR register, 7-11, 7-13, 12-20

E

E command, 12-15

“edit” key (F9), 5-27, 9-4, 12-39

editing

- “click and type” method, 3-26, 9-4
- command line, 6-3, 12-36
- data values, 9-4, 12-39
- dialog boxes, 6-9
- expression side effects, 9-5
- FILE, DISASSEMBLY, CALLS, 5-27
- function key method, 9-4, 12-39
- MEMORY, CPU, DISP, WATCH, 5-27
- mouse method, 9-4
- overwrite method, 9-4
- window contents, 5-27

end key, scrolling, 5-26, 12-39

entering commands

- from pulldown menus, 6-6—6-12
- on the command line, 6-2—6-5

entry point, 8-10

enumerated types, display format, 5-14

environment variables

- D_OPTIONS
 - simulator, 2-6, 2-13*
 - SWDS, 1-12, 1-13*
- D_DIR, 6-13, 11-10
 - simulator, 2-5*
 - SWDS, 1-11*
- D_SRC, 8-9
 - simulator, 2-6*
 - SWDS, 1-11*
- for debugger options
 - simulator, 2-6, 2-13*
 - SWDS, 1-12, 1-13*
- identifying auxiliary directories
 - simulator, 2-5*
 - SWDS, 1-11*
- identifying source directories
 - simulator, 2-6*
 - SWDS, 1-11*

errors, SWDS, system configuration, 1-15

EVAL command, 9-3, 12-15

- modifying PC, 8-10
- side effects, 9-5

executing code, 3-11, 8-10—8-15

- See also* run commands
- benchmarking, 3-17, 8-12
- conditionally, 3-20, 8-15
- function key method, 12-38
- halting execution, 3-15, 8-16
- program entry point, 3-15, 3-17, 8-10—8-15

- single stepping, 3-19, 12-11, 12-13, 12-25, 12-32
- while disconnected from the target system, 8-14, 12-28

executing commands, 6-3

exiting the debugger, 1-15, 2-14, 3-28, 12-25

expressions, 13-1—13-6

- addresses, 9-7
- evaluation
 - with *? command*, 9-3, 12-7, 12-34
 - with *EVAL command*, 9-3, 12-15
- expression analysis, 13-4
- operators, 13-2—13-3
- restrictions, 13-4
- side effects, 9-5
- void expressions, 13-4

extensions, 4-9

F

F4 key, 5-28, 9-14, 12-38

FILE command, 3-11, 3-14, 5-8, 8-6, 12-15

- changing the current directory, 6-14, 12-10
- pull-down selection, 6-11

FILE window, 3-11, 3-14, 5-8, 8-6

- colors, 11-5
- customizing, 11-5

file/load commands

- ADDR command, 8-5, 8-7, 12-7
- CALLS command, 8-7, 12-10
- DASM command, 8-5, 12-13
- FILE command, 3-11, 3-14, 8-6, 12-15
- FUNC command, 3-14, 8-6, 12-16
- LOAD command, 3-5, 8-8, 12-17
- pull-down menu, 6-11
- RELOAD command, 8-8, 12-26
- RESTART command, 12-26
- SLOAD command, 8-8, 12-31

files

- connecting to I/O ports, 7-10, 12-19
- disconnecting from I/O ports, 7-13, 12-21
- saving memory to a file, 9-9, 12-24

FILL command, 9-10, 12-16

floating point

- display format, 3-25, 5-14
- operations, 13-4

FO control bit, 7-11, 7-13, 12-20

FUNC command, 3-14, 5-8, 8-6, 12-16

function calls

- displaying functions, 12-16
 - keyboard method*, 5-10
 - mouse method*, 5-10
- executing function only, 12-27
- in expressions, 9-5, 13-4
- stepping over, 12-11, 12-25
- tracking in CALLS window, 5-9—5-10, 8-7, 12-10

G

–g shell option, 4-8, 4-9

GO command, 3-11, 8-11, 12-17

graphics card requirements, 1-2, 2-2

grouping/reference operators, 13-2

H

HALT command, 8-14, 12-17

halting

- batch file execution, 6-13
- debugger, 1-15, 2-14, 12-25
- program execution, 1-15, 2-14, 3-15, 8-10, 8-16
 - function key method*, 8-16, 12-37
 - mouse method*, 8-16
- target system, 12-17

hardware checklist

- simulator
 - PC systems*, 2-2
 - SUN systems*, 2-11
 - VMS systems*, 2-8
- SWDS, 1-2

hexadecimal notation, addresses, 9-7

history, of commands, 6-4

home key, scrolling, 5-26, 12-39

host system

- simulator
 - PC systems*, 2-2
 - SUN systems*, 2-11
 - VMS systems*, 2-8
- SWDS, 1-2

I

–i debugger option

- simulator, 2-13, 8-9
- SWDS, 1-13, 8-9
- with *D_OPTIONS* environment variable
 - simulator*, 2-6

SWDS, 1-12

I/O memory

- adding to memory map, 7-6, 12-18
- deleting from memory map, 7-8, 12-20
- simulating, 7-10—7-14, 12-19, 12-21

increment operator, 13-3

index numbers, for data in WATCH window, 5-15, 9-16

indirection operator (*), 9-8

init.clr, 11-9, 11-10, 12-30, A-1 SWDS, 1-3

initdb.bat

- invoking, 1-10, 2-5
- sample
 - simulator*, 2-4
 - SWDS*, 1-10
- simulator, 2-4—2-14
- SWDS, 1-9—1-16

installation

- debugger software
 - simulator*
 - PC systems, 2-3
 - VMS systems, 2-9
 - SWDS version*, 1-9
- simulator, Sun version, 2-12
- SWDS
 - board*, 1-4—1-8
 - debugger software*, 1-9

integer, display format, 5-14

interrupts

- receive, 7-11, 7-13, 12-20
- transmit, 7-11, 7-13, 12-20

invoking

- autoexec.bat
 - simulator*, 2-5
 - SWDS*, 1-10
- custom displays, 11-11
- debugger, 3-4
 - simulator*, 2-13
 - SWDS*, 1-13
- initdb.bat
 - simulator*, 2-5
 - SWDS*, 1-10
- shell program, 4-9

J

jumper settings

I/O address space, SWDS, 1-12

wait state, 1-7

K

key sequences

- displaying functions, 12-39
- displaying previous commands (command history), 12-36
- editing
 - command line*, 6-3, 12-36
 - data values*, 5-27, 12-39
- halting actions, 12-37
- moving a window, 5-24, 12-38
- opening additional DISP windows, 12-39
- pulldown selections, 12-37
- restrictions
 - SUN version of simulator*, 2-12
 - VMS version of simulator*, 2-10
- running code, 12-38
- scrolling, 5-26, 12-39
- selecting the active window, 5-18, 12-38
- setting/clearing breakpoints, 12-39
- single stepping, 8-13
- sizing a window, 5-22, 12-38
- switching debugging modes, 12-37

L

labels, for data in WATCH window, 3-18, 5-15, 9-16

limits

- breakpoints, 10-2
- file size, 8-7
- open DISP windows, 5-14
- paths, 8-9
- window positions, 5-23
- window sizes, 5-21

linker, 1-3, 2-2, 2-8, 2-11, 4-7, 4-8
command files, MEMORY definition, 7-2

LOAD command, 3-5, 8-8, 12-17

load/file commands, 12-16

- ADDR command, 8-5, 12-7
- CALLS command, 8-7, 12-10
- DASM command, 8-5, 12-13
- FILE command, 3-11, 8-6, 12-15
- FUNC command, 3-14, 8-6, 12-16
- LOAD command, 3-5, 8-8, 12-17
- pulldown menu, 6-11
- RELOAD command, 8-8, 12-26
- RESTART command, 12-26

- SLOAD command, 8-8, 12-31
 - loading
 - batch files, 6-13
 - custom displays, 11-10
 - object code, 3-4, 8-8
 - after invoking the debugger, 8-8
 - symbol table only, 8-8, 12-31
 - while invoking the debugger, 8-8
 - simulator, 2-13
 - SWDS, 1-13
 - without symbol table, 8-8, 12-26
 - logical operators, 13-2
 - conditional execution, 8-15
- M**
- MA command, 7-6, 7-9, 12-18—12-19
 - pulldown selection, 6-12
 - managing data, 9-1—9-18
 - basic commands, 9-2—9-3
 - MAP command, 7-7, 12-19
 - pulldown selection, 6-12
 - MC command, 7-10, 12-19
 - pulldown selection, 6-12
 - MD command, 7-8, 12-20
 - pulldown selection, 6-12
 - MEM command, 3-6, 5-11, 9-6, 12-22
 - memory
 - commands
 - FILL* command, 9-10, 12-16
 - MA* command, 7-6, 7-9, 12-18—12-19
 - MAP* command, 7-7, 12-19
 - MC*, 7-10, 12-19
 - MD* command, 7-8, 12-20
 - MI*, 7-13, 12-21
 - ML* command, 7-7, 12-21
 - MR* command, 7-8, 12-24
 - MS* command, 9-9, 12-24
 - pulldown menu*, 6-12
 - default map, 7-3
 - simulator*, 2-11
 - PC systems, 2-2
 - VMS version, 2-8
 - SWDS, 1-3
 - displaying in different numeric format, 3-25, 9-8
 - filling, 9-10, 12-16
 - invalid locations, 7-7
 - map
 - adding ranges, 12-18
 - deleting ranges, 12-20
 - resetting, 12-24
 - mapping, 7-1—7-14
 - adding ranges, 7-6
 - dbinit.cmd*, SWDS, 1-3
 - defining a memory map, 7-2
 - deleting ranges, 7-8
 - enabling/disabling, 7-7
 - listing current map, 7-7
 - modifying, 7-8
 - resetting, 7-8
 - returning to default, 7-9
 - siminit.cmd*, *simulator*
 - PC systems, 2-2
 - SUN systems, 2-11
 - VMS version, 2-8
 - simulating I/O ports, 7-10, 7-13, 12-19, 12-21
 - nonexistent locations, 7-2
 - pseudoregisters, 7-12
 - requirements
 - simulator*, PC systems, 2-2
 - SWDS, 1-2
 - saving, 9-9, 12-24
 - serial port, 7-11
 - simulating I/O memory, 7-10—7-14, 12-19, 12-21
 - valid types, 7-6
 - MEMORY window, 3-6, 5-11, 9-2, 9-6, 12-22
 - colors, 11-6
 - customizing, 11-6
 - modifying display, 12-22
 - menu bar, 3-5, 6-6
 - customizing its appearance, 11-7
 - items without menus, 6-10
 - using menus, 6-6—6-12
 - messages, B-1—B-18
 - MI command, 7-13, 12-21
 - pulldown selection, 6-12
 - MIX command, 3-13, 8-3, 12-22
 - pulldown selection, 6-12, 8-3
 - mixed mode, 3-12, 5-4
 - selection, 8-3
 - ML command, 7-7, 12-21
 - pulldown selection, 6-12
 - modes, 5-2—5-4
 - assembly mode, 5-3
 - auto mode, 5-2
 - commands
 - ASM* command, 3-13
 - C* command, 3-13, 12-10
 - MIX* command, 3-13, 12-22

- mixed mode, 5-4
- pull-down menu, 3-12, 3-13, 6-12, 8-3
- restrictions, 5-4
- selection, 3-12, 8-3
 - commands*, 8-3
 - function key method*, 8-3, 12-37
 - mouse method*, 8-3
- modifying
 - colors, 11-2—11-7
 - command line, 6-3
 - command-line prompt, 11-12
 - current directory, 6-14, 12-10
 - data values, 9-4
 - memory map, 7-8
 - window borders, 11-8
- mono.clr, 11-9
 - SWDS, 1-3
- monochrome monitors, 11-9
- mouse
 - cursor, 5-16
 - requirements, 2-2, 2-11
 - SWDS, 1-2
 - restrictions, VMS version of simulator, 2-10
- MOVE command, 3-9, 5-23, 12-23
 - effect on entering other commands, 6-4
- moving a window, 5-22, 12-23
 - function key method, 3-9, 5-24, 12-38
 - mouse method, 3-9, 5-22
 - MOVE command, 3-9, 5-23
 - XY screen limits, 5-23
- MR command, 7-8, 12-24
 - pull-down selection, 6-12
- MS command, 9-9, 12-24
 - pull-down selection, 6-12
- mv debugger option, 2-6, 2-13

N

- natural format, 3-25, 13-5
- NEXT command, 3-19, 8-13, 12-25
 - from the menu bar, 6-10
 - function key entry, 6-10, 12-38
- nonexistent locations, 7-2

O

- object files

- creating, 8-8
- loading, 12-17
 - after invoking the debugger*, 8-8
 - simulator*, 2-13
 - SWDS, 1-13
 - symbol table only*, 1-14, 2-14, 12-31
 - while invoking the debugger*, 3-4, 8-8
 - simulator, 2-13
 - SWDS, 1-13
 - without symbol table*, 8-8, 12-26
- object format converter, 4-7
- operators, 13-2—13-3
 - & operator, 9-7
 - * operator (indirection), 9-8
 - side effects, 9-5
- overwrite editing, 9-4

P

- p debugger option
 - SWDS, 1-14
 - with D_OPTIONS environment variable, SWDS, 1-12
- page-up/page-down keys, scrolling, 5-26, 12-39
- parameters
 - db2x command, 1-13—1-16
 - dspcl shell, 4-9
 - entering in a dialog box, 6-9
 - sim2x command, 2-13—2-14
- PATH statement, simulator, 1-11, 2-5
- PC, 8-10
 - displaying contents of, 3-6
 - finding the current PC, 5-7
- pointers
 - displaying/modifying contents, 3-23, 9-12
 - format in a DISP window, 5-14
 - format in DISP window, 3-23, 9-13, 12-14
 - natural format, 13-5
 - typecasting, 13-5
- port address
 - simulator, 7-10—7-14
 - SWDS, 1-12, 1-14
- ports, simulating, 7-10—7-11, 7-13—7-14, 12-19—12-36
- power requirements, SWDS, 1-2
- program
 - entry point, 8-10
 - resetting*, 12-26

- execution, halting, 1-15, 2-14, 3-15, 8-10, 8-16, 12-37
- preparation for debugging, 4-8
- program counter (PC), 9-11
- program memory
 - adding to memory map, 7-6, 12-18
 - deleting from memory map, 7-8, 12-20
 - filling, 9-10, 12-16
 - saving, 9-9, 12-24
- PROMPT command, 11-12, 12-25
 - pulldown selection, 6-12
- pseudoregisters, BIO, 9-17
- pulldown menus, 6-6
 - colors, 11-7
 - correspondence to commands, 6-11
 - customizing their appearance, 11-7
 - entering parameter values, 6-9
 - escaping, 6-8
 - function key methods, 6-8, 12-37
 - list of menus, 6-6
 - mouse methods, 6-7
 - moving to another menu, 6-8
 - usage, 6-7

Q

QUIT command, 1-15, 2-14, 3-28, 12-25

R

re-entering commands, 6-4, 12-36

registers

- BIO pseudoregister, 9-17
- CLK pseudoregister, 3-17, 8-17
- displaying/modifying, 9-11
- DRR, 7-11, 7-13, 12-20
- DXR, 7-11, 7-13, 12-20
- program counter (PC), 9-11
- referencing by name, 13-4
- stack pointer (SP), 9-11
- status register (ST), 9-11
- TDXR, 7-11, 7-13, 12-20
- TRCV, 7-11, 7-13, 12-20

relational operators, 13-2

- conditional execution, 8-15

relative pathnames, 6-14, 8-9, 12-10

RELOAD command, 12-26

- pulldown selection, 6-11

required tools, 1-3, 2-2, 2-8, 2-11

RESET command, 3-5, 8-14, 12-26

- pulldown selection, 6-11

resetting

- memory map, 12-24
- program entry point, 12-26
- SWDS, 1-3, 1-12
- target system, 3-5, 8-14, 12-26

RESTART (REST) command, 3-15, 3-17, 8-10, 12-26

- pulldown selection, 6-11

restrictions

- See also* limits and complaints
- C expressions, 13-4
- simulator
 - PC systems*, 2-6
 - Sun version*, 2-12
 - VAX version*, 2-10

RETURN (RET) command, 8-11, 12-27

RUN command, 3-15, 8-11, 12-27

- from the menu bar, 6-10
- function key entry, 6-10, 8-11, 12-38
- menu bar selections, 6-10
- with conditional expression, 3-20

run commands, 3-11

- CNEXT command, 8-13, 12-11
- conditional parameters, 3-20
- CSTEP command, 3-19, 8-13, 12-13
- GO command, 8-11, 12-17
- HALT command, 8-14, 12-17
- menu bar selections, 6-10, 12-38
- NEXT command, 3-19, 8-13, 12-25
- RESET command, 8-14
- RESTART command, 3-15, 3-17, 8-10
- RETURN command, 8-11, 12-27
- RUN command, 3-15, 8-11, 12-27
- RUNB command, 3-17, 8-12, 8-17, 12-28
- RUNF command, 8-14, 12-28
- STEP command, 3-19, 8-12, 12-32

RUNB command, 3-17, 8-12, 8-17, 12-28

RUNF command, 8-14, 12-28

running programs, 8-10—8-15

- conditionally, 8-15
- halting execution, 8-16
- program entry point, 8-10—8-15
- while disconnected from the target system, 8-14

S

- s debugger option
 - simulator, 2-14, 8-8
 - SWDS, 1-14, 8-8
 - with D_OPTIONS environment variable
 - simulator*, 2-6
 - SWDS*, 1-12
- saving custom displays, 11-10
- SCOLOR command, 11-2, 12-29
 - pull-down selection, 6-12
- SCONFIG command, 11-10, 12-30
 - pull-down selection, 6-12
- screen-customization commands
 - BORDER command, 11-8, 12-9
 - COLOR command, 11-2, 12-12
 - PROMPT command, 11-12, 12-25
 - pull-down menu, 6-12
 - SCOLOR command, 11-2, 12-29
 - SCONFIG command, 11-10, 12-30
 - SSAVE command, 11-10, 12-32
- scrolling, 3-10, 5-25
 - function key method, 3-10, 5-26, 12-39
 - mouse method, 3-10, 5-26, 9-7
- serial port, pseudoregister
 - RIRP, 7-12
 - RIRT, 7-12
 - XIRP, 7-12
 - XIRT, 7-12
- serial ports
 - simulation, 7-11, 7-13, 12-20
 - TDM mode, 7-11, 7-13, 12-20
- shell program, 4-9
- side effects, 9-5, 13-3
 - valid operators, 9-5
- sim2x command, 3-4
 - options
 - mv*, 2-13
 - b*, 2-13
 - D_OPTIONS* environment variable, 2-6
 - i*, 2-13
 - s*, 2-14
 - t*, 2-14
 - v*, 2-14
 - z*, 2-14
- sim2x command, 8-8, 8-9
 - options, 2-13—2-14
- siminit.cmd, A-1
- simulator, 2-11
 - PC systems*, 2-2
 - VMS version*, 2-8
- simulator, 2-1—2-14
 - BIO simulation, 9-17
 - I/O memory, 7-10—7-14, 12-19, 12-21
 - PC systems, 2-2—2-7
 - debugger environment*, 2-4—2-7
 - hardware requirements*, 2-2
 - restrictions*, 2-6
 - software installation*, 2-3—2-7
 - software requirements*, 2-2
 - restrictions
 - benchmarking*, 8-17
 - CLK*, 8-17
 - color displays*, 2-10, 2-12
 - keyboard mapping*, 2-10, 2-12
 - memory map size*, 2-6
 - mouse use*, 2-10
 - PC version*, 2-6
 - Sun version*, 2-12
 - VMS version*, 2-10
- SUN systems, 2-11—2-12
 - debugger restrictions*, 2-12
 - hardware requirements*, 2-11
 - installation*, 2-12
 - software requirements*, 2-11
- VMS systems, 2-8—2-10
 - debugger restrictions*, 2-10
 - hardware requirements*, 2-8
 - installation*, 2-9
 - software requirements*, 2-8
- single step
 - commands
 - CNEXT* command, 8-13, 12-11
 - CSTEP* command, 3-19, 8-13, 12-13
 - menu bar selections*, 6-10
 - NEXT* command, 3-19, 8-13, 12-25
 - STEP* command, 3-19, 8-12, 12-32
 - execution, 8-12
 - assembly language code*, 8-12, 12-32
 - C code*, 8-13, 12-13
 - function key method*, 8-13, 12-38
 - mouse methods*, 8-13
 - over function calls*, 8-13, 12-11, 12-25
- SIZE command, 3-8, 5-21, 12-30
 - effect on entering other commands, 6-4
- sizeof operator, 13-4
- sizes
 - display, 5-23

- displayable files, 8-7
- windows, 5-21
- sizing a window, 5-20
 - function key method, 3-8, 5-22, 12-38
 - mouse method, 3-8, 5-20
 - SIZE command, 3-8, 5-21
 - size limits, 5-21
 - while moving it, 5-23, 12-23
- SLOAD command, 8-8, 12-31
 - pulldown selection, 6-11
 - s debugger option
 - simulator*, 2-14
 - SWDS*, 1-14
- software checklist, simulator
 - PC systems, 2-2
 - SUN systems, 2-11
 - VMS systems, 2-8
- SPC control bit, 7-11, 7-13, 12-20
- SSAVE command, 11-10, 12-32
 - pulldown selection, 6-12
- stack pointer (SP), 9-11
- status register (ST), 9-11
- STEP command, 3-19, 8-12, 12-32
 - from the menu bar, 6-10
 - function key entry, 6-10, 12-38
- structures
 - direct reference operator, 13-2
 - displaying/modifying contents, 9-12
 - format in DISP window, 3-24, 9-13, 12-14
 - indirect reference operator, 13-2
- SWDS
 - benchmarking, 8-18
 - debugger installation, 1-1—1-16
 - hardware requirements, 1-2
 - installation
 - board*, 1-4—1-8
 - preparation*, 1-4
 - jumper settings
 - PC memory segment selections*, 1-6
 - wait state*, 1-7
 - power requirements, 1-2
 - resetting, 1-3, 1-12
- switch settings, I/O address space, SWDS, 1-14
- symbol table, loading without object code, 1-14, 2-14, 8-8, 12-31
- symbolic addresses, 9-7
- system commands, 6-14
 - CD command, 3-22, 6-14, 8-9, 12-10

- CLS command, 3-22, 6-5, 12-11
- DIR command, 3-22, 6-14, 12-14
- QUIT command, 3-28, 12-25
- RESET command, 3-5, 12-26
- TAKE command, 6-13, 7-9, 12-33
- USE command, 8-9, 12-33
- system configuration errors, 1-15

T

- t debugger option, 2-6
 - simulator, 2-14
 - SWDS, 1-14
 - with D_OPTIONS environment variable
 - simulator*, 2-6
 - SWDS*, 1-12
- TAKE command, 6-13, 7-9, 12-33
- target system
 - memory definition for debugger, 7-1—7-14
 - resetting, 3-5, 12-26
- TDM serial-port mode, 7-11, 7-13, 12-20
- terminating the debugger, 12-25
- text files, displaying, 3-14, 8-7
- troubleshooting, SWDS, 1-15
- type casting, 3-25, 13-4
- type checking, 3-21, 9-2

U

- USE command, 8-9, 12-33

V

- v debugger option
 - simulator, 2-14
 - SWDS, 1-14
 - with D_OPTIONS environment variable
 - simulator*, 2-6
 - SWDS*, 1-12
- variables
 - aggregate values in DISP window, 3-22, 5-14, 9-12, 12-14
 - determining type, 9-2
 - displaying in different numeric format, 3-25, 13-5
 - displaying/modifying, 9-15
 - scalar values in WATCH window, 5-15, 9-15—9-16
- versions, simulator memory map, 2-13

void expressions, 13-4

W

WA command, 3-18, 5-15, 9-15, 12-34
 pulldown selection, 6-11

watch commands

pull-down menu, 6-11, 9-15
 WA command, 3-18, 9-15, 12-34
 WD command, 3-20, 9-16, 12-34
 WR command, 3-20, 9-16, 12-35

WATCH window, 3-18, 5-15, 9-2, 9-15, 12-34, 12-35
 adding items, 9-15
 closing, 5-28, 9-16, 12-35
 colors, 11-6
 customizing, 11-6
 deleting items, 9-16, 12-34
 labeling watched data, 9-16
 opening, 9-15

WD command, 3-20, 5-15, 9-16, 12-34
 pulldown selection, 6-11

WHATIS command, 3-21, 9-2, 12-34

WIN command, 3-6, 5-19, 12-35

windows, 5-5—5-15

active window, 5-17—5-19
 border styles, 11-8, 12-9
 closing, 5-28
 commands

ADDR command, 5-7, 5-8
CALLS command, 5-9
DASM command, 5-7
DISP command, 5-14
FILE command, 5-8

FUNC command, 5-8

MEM command, 5-11

MOVE command, 3-9

SIZE command, 3-8, 12-30

WA command, 5-15

WD command, 5-15

WIN command, 3-6, 5-19, 12-23, 12-35

WR command, 5-15

editing, 5-27

moving, 3-9, 5-22, 12-23

function keys, 5-24, 12-38

mouse method, 5-22

MOVE command, 5-23

XY positions, 5-23

resizing, 3-8, 5-20

function keys, 5-22, 12-38

mouse method, 5-20

SIZE command, 5-21

while moving, 5-23, 12-23

scrolling, 3-10, 5-25

size limits, 5-21

WR command, 3-20, 5-15, 9-16, 12-35
 pulldown selection, 6-11

X

-x debugger option
 simulator, 2-14
 SWDS, 1-14

Z

-z shell option, 4-9