

TMS320C16

Source Debugger

User's Guide

PRELIMINARY

PRELIMINARY



**TEXAS
INSTRUMENTS**

TMS320C16

Source Debugger

User's Guide

SPRU077
MARCH 1992



IMPORTANT NOTICE

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

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

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

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

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

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

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

IMPORTANT NOTICE

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

TI warrants performance of its semiconductor products to current specifications 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. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

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

Texas Instruments products are not intended for use in life-support appliances, devices, or systems. Use of a TI product in such applications without the written consent of the appropriate TI officer is prohibited.

WARNING

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

Read This First

What Is This Book About?

This book tells you how to install and use the TMS320C16 source debugger as a programmer interface to the 'C16 evaluation module (EVM).

The TMS320C16 device is a memory and speed superset of both the TMS320C10 and TMS320C15. The TMS320C16 EVM can be used to evaluate both the 'C10 and 'C15 in addition to the 'C16. For more information about the differences in speed, memory maps, and other characteristics among these devices, refer to the *TMS320C1x User's Guide*.

How to Use This Manual

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.
 - Chapter 1 provides installation instructions for the 'C16 EVM and debugger.
 - Chapter 2 is a tutorial that introduces you to many of the debugger features.
- ☐ **Part II: Debugger Description** contains detailed information about using the debugger.
 - Chapter 3 is analogous to a traditional manual introduction. It lists the key features of the debugger, describes additional 'C16 software tools, and tells you how to prepare a 'C16 program for debugging.
 - The remaining chapters in Part II detail the individual topics that are introduced in the tutorial. For example, Chapter 4 describes all of the debugger's windows and tells you how to move and size them; Chapter 5 describes everything you need to know about entering commands.
- ☐ **Part III: Reference Material** provides supplementary information.
 - Chapter 11 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 12 provides information about C expressions. The debugger commands are powerful because they accept C expressions as parameters. 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 installation chapter.
 - Complete the tutorial in Chapter 2.
 - Read the alphabetical command reference in Chapter 11.
- ☐ If this is the first time that you have used a debugger or similar tool, then you may want to:
 - Use the installation chapter.
 - Complete the tutorial in Chapter 2.
 - Read all of the chapters in Part II.

Notational Conventions


This document uses the following conventions.


- ☐ The TMS320C16 processor is referred to as the 'C16.
- ☐ The 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
? pc,x	0x00c2
? acc,x	0xd6bff77e
? p,x	0x000bec3d
? tos,x	0x5555

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 TMS320C16 device and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Customer Resource Center (CRC) at (800) 336-5236. When ordering, please identify the book by its title and literature number.

TMS320C1x User's Guide (literature number SPRU013) discusses the hardware aspects of the 'C1x generation of CMOS fixed-point digital signal processors. It describes pin assignments, architecture, instruction set, and software and hardware applications. This book also features a section with analog interface peripherals and applications for the 'C1x DSPs, and includes a consolidated data sheet with electrical specifications and package information for all 'C1x devices.

TMS320 Fixed-Point DSP Assembly Language Tools User's Guide (literature number SPRU018) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C1x, 'C2x, and 'C5x generations of devices.

If you 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†: (800) 336-5236 Or write to: Texas Instruments Incorporated Market Communications Manager, MS 736 P.O. Box 1443 Houston, Texas 77251-1443
Order Texas Instruments documentation	Call the CRC†: (800) 336-5236
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: Texas Instruments Incorporated Technical Publications Manager, MS 702 P.O. Box 1443 Houston, Texas 77251-1443

† Texas Instruments Customer Response Center

Trademarks

MS-DOS is a registered trademark of Microsoft Corp.

MS-Windows is a registered trademark of Microsoft Corp.

PC-DOS is a trademark of International Business Machines Corp.

VEGA Deluxe is a trademark of Video Seven Incorporated.

Contents

Part I: Hands-On Information

1	Installing the Evaluation Module and the Debugger	1-1
	<i>Lists the hardware and software you'll need to install and run the 'C16 EVM with the debugger, guides you through the installation process, and tells you how to invoke the EVM version of the debugger.</i>	
1.1	What You'll Need	1-2
	Hardware checklist	1-2
	Software checklist	1-3
1.2	Step 1: Installing the EVM Board in Your PC	1-4
	Preparing the EVM board for installation	1-4
	Setting the EVM board into your PC	1-6
1.3	Step 2: Installing the Debugger Software	1-7
1.4	Step 3: Setting Up the Debugger Environment	1-8
	Invoking the new or modified batch file	1-9
	Modifying the PATH statement	1-9
	Setting up the environment variables	1-9
	Identifying the correct I/O switches	1-10
1.5	Using the Debugger With Microsoft Windows	1-11
1.6	Invoking the Debugger	1-12
1.7	Exiting the Debugger	1-13
2	An Introductory Tutorial to the Debugger	2-1
	<i>This chapter provides a step-by-step introduction to the debugger and its features.</i>	
	How to use this tutorial	2-2
	A note about entering commands	2-2
	An escape route (just in case)	2-3
	Invoke the debugger and load the sample program's object code	2-3
	Take a look at the display	2-4
	What's in the DISASSEMBLY window?	2-5
	Select the active window	2-5
	Size the active window	2-7
	Zoom the active window	2-8
	Move the active window	2-9
	Scroll through a window's contents	2-10
	Execute some code	2-11
	Open a text file	2-11
	Become familiar with the two debugging modes	2-12

Use the basic RUN command	2-14
Set some breakpoints	2-14
Watch some values and single-step through code	2-16
Run code conditionally	2-17
Clear the COMMAND window display area	2-18
Change some values	2-19
Define a memory map	2-20
Define your own command string	2-21
Close the debugger	2-21

Part II: Debugger Description

3 Overview of a Code Development and Debugging System	3-1
<i>Discusses features of the debugger and additional software tools.</i>	
3.1 Description of the 'C16 Debugger	3-2
Key features of the debugger	3-3
3.2 Developing Code for the 'C16	3-5
3.3 Preparing Your Program for Debugging	3-7
3.4 Debugging 'C16 Programs	3-8
4 The Debugger Display	4-1
<i>Describes the default displays, describes the various types of windows on the display and tells you how to move and size the windows.</i>	
4.1 The Debugger Modes	4-2
4.2 Descriptions of the Different Kinds of Windows and Their Contents	4-3
COMMAND window	4-4
DISASSEMBLY window	4-5
FILE window	4-5
MEMORY window	4-6
CPU window	4-8
WATCH window	4-9
4.3 Cursors	4-10
4.4 The Active Window	4-10
Identifying the active window	4-11
Selecting the active window	4-11
4.5 Manipulating Windows	4-13
Resizing a window	4-13
Zooming a window	4-15
Moving a window	4-16
4.6 Manipulating a Window's Contents	4-19
Scrolling through a window's contents	4-19
Editing the data displayed in windows	4-21
4.7 Closing a Window	4-21

5	Entering and Using Commands	5-1
	<i>Describes the rules for entering commands from the command line, tells you how to use the menu selections 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.</i>	
5.1	Entering Commands From the Command Line	5-2
	How to type in and enter commands	5-3
	Sometimes, you can't type a command	5-4
	Using the command history	5-5
	Clearing the display area	5-5
	Recording information from the display area	5-6
5.2	Using the Menu Bar and the Pulldown Menus	5-7
	Using the pulldown menus	5-8
	Escaping from the pulldown menus	5-9
	Entering parameters in a dialog box	5-10
	Using menu bar selections that don't have pulldown menus	5-11
5.3	Entering Commands From a Batch File	5-12
	Echoing strings in a batch file	5-13
	Controlling command execution in a batch file	5-13
5.4	Defining Your Own Command Strings	5-15
5.5	Entering Operating-System Commands	5-18
	Entering a single command from the debugger command line	5-18
	Entering several commands from a system shell	5-19
	Additional system commands	5-19
6	Defining a Memory Map	6-1
	<i>Contains instructions for setting up a memory map that will enable the debugger to correctly access target memory and includes hints about using batch files.</i>	
6.1	The Memory Map: What It Is and Why You Must Define It	6-2
	Defining the memory map in a batch file	6-2
	Potential memory map problems	6-3
6.2	A Sample Memory Map	6-3
6.3	Identifying Usable Memory Ranges	6-4
6.4	Enabling Memory Mapping	6-5
6.5	Checking the Memory Map	6-5
6.6	Modifying the Memory Map During a Debugging Session	6-6
	Returning to the original memory map	6-7
7	Loading, Displaying, and Running Code	7-1
	<i>Tells you how to load object files, how to run your programs, and how to halt program execution.</i>	
7.1	Displaying Your Source Programs (or Other Text Files)	7-2
	Displaying assembly language code	7-2
	Modifying assembly language code	7-3
	Additional information about modifying assembly language code	7-5
	Displaying other text files	7-5
7.2	Loading Object Code	7-6
	Loading code while invoking the debugger	7-6
	Loading code after invoking the debugger	7-6

7.3	Where the Debugger Looks for Source Files	7-7
7.4	Running Your Programs	7-8
	Defining the starting point for program execution	7-8
	Running code	7-9
	Single-stepping through code	7-9
	Running code while emulation is disabled	7-11
	Running code conditionally	7-12
7.5	Halting Program Execution	7-12
8	Managing Data	8-1
	<i>Describes the data-display windows and tells you how to edit data (memory contents and register contents).</i>	
8.1	Where Data Is Displayed	8-2
8.2	Basic Commands for Managing Data	8-2
8.3	Basic Methods for Changing Data Values	8-4
	Editing data displayed in a window	8-4
	Advanced “editing”—using expressions with side effects	8-5
8.4	Managing Data in Memory	8-6
	Displaying memory contents	8-6
	Displaying program memory	8-7
	Saving memory values to a file	8-8
	Filling a block of memory	8-9
8.5	Managing Register Data	8-10
	Displaying register contents	8-11
8.6	Managing Data in a WATCH Window	8-12
	Displaying data in the WATCH window	8-12
	Deleting watched values and closing the WATCH window	8-13
8.7	Displaying Data in Alternative Formats	8-14
	Changing the default format for specific data types	8-14
	Changing the default format with ?, MEM, and WA	8-16
9	Using Breakpoints	9-1
	<i>Describes the use of hardware breakpoints to halt code execution.</i>	
9.1	Setting a Breakpoint	9-2
9.2	Clearing a Breakpoint	9-3
9.3	Finding the Breakpoints That Are Set	9-4
10	Customizing the Debugger Display	10-1
	<i>Contains information about the commands that you can use for customizing the display and identifies the display areas that you can modify.</i>	
10.1	Changing the Colors of the Debugger Display	10-2
	Area names: common display areas	10-3
	Area names: window borders	10-4
	Area names: COMMAND window	10-4
	Area names: DISASSEMBLY window	10-5
	Area names: data-display windows	10-6
	Area names: menu bar and pulldown menus	10-7

10.2	Changing the Border Styles of the Windows	10-8
10.3	Saving and Using Custom Displays	10-9
	Changing the default display for monochrome monitors	10-9
	Saving a custom display	10-10
	Loading a custom display	10-10
	Invoking the debugger with a custom display	10-11
	Returning to the default display	10-11
10.4	Changing the Prompt	10-12

Part III: Reference Material

11 Summary of Commands and Special Keys 11-1

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

11.1	Functional Summary of Debugger Commands	11-2
	Changing modes	11-3
	Managing windows	11-3
	Displaying and changing data	11-3
	Performing system tasks	11-4
	Displaying files and loading programs	11-5
	Managing breakpoints	11-5
	Customizing the screen	11-5
	Memory mapping	11-6
	Running programs	11-6
11.2	How the Menu Selections Correspond to Commands	11-7
	Program-execution commands	11-7
	File/load commands	11-7
	Breakpoint commands	11-7
	Watch commands	11-8
	Memory commands	11-8
	Screen-configuration commands	11-8
	Mode commands	11-8
11.3	Alphabetical Summary of Debugger Commands	11-9
11.4	Summary of Special Keys	11-34
	Editing text on the command line	11-34
	Using the command history	11-34
	Halting or escaping from an action	11-35
	Displaying menu selections	11-35
	Running code	11-35
	Selecting a window	11-36
	Moving or sizing a window	11-36
	Editing data or selecting the active field	11-36
	Scrolling through a window's contents	11-37

12 Basic Information About C Expressions	12-1
<i>Many of the debugger commands accept C expressions as parameters. This chapter provides general information about the rules governing C expressions and describes specific implementation features related to using C expressions as command parameters.</i>	
12.1 C Expressions for Assembly Language Programmers	12-2
12.2 Restrictions and Features Associated With Expression Analysis in the Debugger ..	12-4
Restrictions	12-4
Additional features	12-4
A What the Debugger Does During Invocation	A-1
<i>In some circumstances, you may find it helpful to know the steps that the debugger goes through during the invocation process; this appendix lists these steps.</i>	
B Debugger Messages	B-1
<i>Describes progress and error messages that the debugger may display.</i>	
B.1 Associating Sound With Error Messages	B-2
B.2 Alphabetical Summary of Debugger Messages	B-2
B.3 Additional Instructions for Expression Errors	B-14
B.4 Additional Instructions for Hardware Errors	B-14
C Glossary	C-1
<i>Defines acronyms and key terms used in this book.</i>	

Figures

1-1	EVM Board I/O Switches	1-4
1-2	EVM Installation	1-6
1-3	DOS-Command Setup for the Debugger	1-8
3-1	The Debugger Display	3-2
3-2	'C16 Software Development Flow	3-5
3-3	Steps You Go Through to Prepare a 'C16 Program	3-7
4-1	Typical Debugger Display	4-2
4-2	Default Appearance of an Active and an Inactive Window	4-11
5-1	The COMMAND Window	5-2
5-2	The Menu Bar in the Debugger Display	5-7
5-3	All of the Pulldown Menus	5-7
6-1	Sample Memory Map for Use With a 'C16 EVM	6-3

Tables

1-1	EVM Board Switch Settings	1-5
1-2	Your Switch Settings	1-5
1-3	Nondefault I/O Address Space	1-10
1-4	Debugger Options	1-12
4-1	Width and Length Limits for Window Sizes	4-14
4-2	Minimum and Maximum Limits for Window Positions	4-17
5-1	Predefined Constants for Use With Conditional Commands	5-14
8-1	Registers and Pseudoregisters for Use With the 'C16	8-10
8-2	Display Formats for Debugger Data	8-14
8-3	Data Types for Displaying Debugger Data	8-15
10-1	Colors and Other Attributes for the COLOR and SCOLOR Commands	10-2
10-2	Summary of Area Names for the COLOR and SCOLOR Commands	10-3

Chapter 1

Installing the Evaluation Module and the Debugger

This chapter contains information that will help you prepare to use the evaluation module (EVM) and debugger. When you complete the installation, turn to Chapter 2, *An Introductory Tutorial to the Debugger*.

Topic	Page
1.1 What You'll Need	1-2
Hardware checklist	1-2
Software checklist	1-3
1.2 Step 1: Installing the EVM Board in Your PC	1-4
Preparing the EVM board for installation	1-4
Setting the EVM board into your PC	1-6
1.3 Step 2: Installing the Debugger Software	1-7
1.4 Step 3: Setting Up the Debugger Environment	1-8
Invoking the new or modified batch file	1-9
Modifying the PATH statement	1-9
Setting up the environment variables	1-9
Identifying the correct I/O switches	1-10
1.5 Using the Debugger With Microsoft Windows	1-11
1.6 Invoking the Debugger	1-12
1.7 Exiting the Debugger	1-13

1.1 What You'll Need

In addition to the materials that are shipped with the debugger and EVM, you'll need the following.

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
<input type="checkbox"/>	display	Monochrome or color (color recommended)
<input type="checkbox"/>	slot	One 16-bit slot
<input type="checkbox"/>	EVM board power requirements	Approximately 1.5 amps @ 5 volts (15 watts)
<input type="checkbox"/>	optional hardware	A Microsoft-compatible mouse
<input type="checkbox"/>		An EGA-compatible graphics display card
<input type="checkbox"/>		A 17" or 19" monitor. The 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 1-4, page 1-12, explains this in detail.)
<input type="checkbox"/>	miscellaneous materials	Blank, formatted disks

Software checklist

- | | | |
|--------------------------|-------------------------|--|
| <input type="checkbox"/> | operating system | MS-DOS or PC-DOS (version 3.0 or later)
Optional: Microsoft Windows (version 3.0 or later) |
| <input type="checkbox"/> | software tools | TMS320 fixed-point family DSP ('C1x'/'C2x'/'C5x) assembler and linker |
| <input type="checkbox"/> | required files † | <i>c16reset</i> resets the EVM |
| <input type="checkbox"/> | optional files † | <i>init.cmd</i> is a file that contains debugger commands. The version of this file that's shipped with the debugger defines a 'C16 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 EVM, 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 6, <i>Defining a Memory Map</i> . |
| <input type="checkbox"/> | † | <i>init.clr</i> is a general-purpose screen configuration file. If this file isn't present when you invoke the debugger, the debugger uses the default screen configuration. For information about this file and about setting up your own screen configuration, refer to Chapter 10, <i>Customizing the Debugger Display</i> . |
| | † | 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 10, <i>Customizing the Debugger Display</i> . |

† Included as part of the debugger package

1.2 Step 1: Installing the EVM Board in Your PC

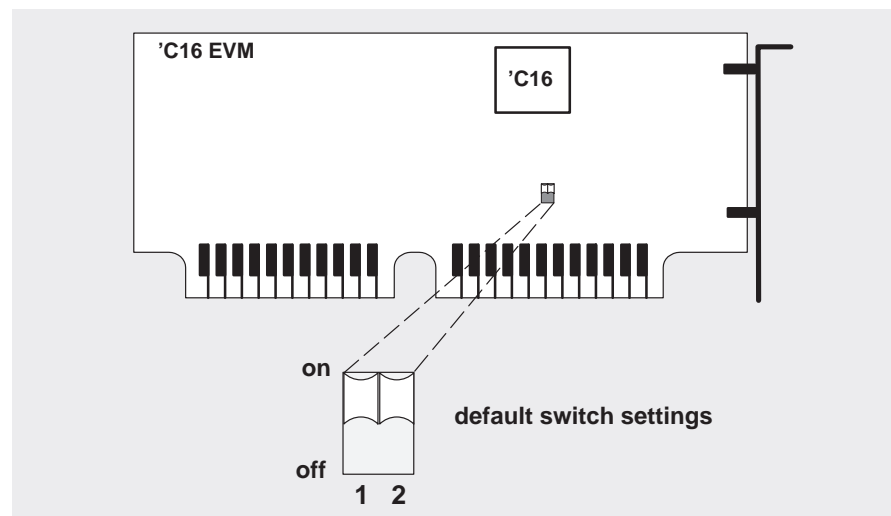
This section contains the hardware installation information for the EVM.

Preparing the EVM board for installation

The EVM board has two switches that identify your system's I/O address space. You can change these switch settings to identify the I/O address space that the EVM uses in your system.

Figure 1–1 shows where these switches are on the EVM board and identifies the switch numbers.

Figure 1–1. EVM Board I/O Switches



Switches are shipped in the default settings shown here and described in Table 1–1. If you use an I/O space that differs from the default, change the switch settings. Table 1–1 shows you how to do this.


In most cases, you can leave the switch settings in the default position. However, you must ensure that the EVM I/O address space does not conflict with other bus settings. For example, if you've installed a bus mouse in your system, you may not be able to use the default switch settings for the I/O address space—the mouse might use this space. Refer to your PC technical reference manual and your other hardware-board manuals to see if there are any I/O space conflicts. If you find a conflict, use one of the settings in Table 1–1.

Table 1–1. EVM Board Switch Settings

		switch #	
		1	2
default	0x0240-0x025F	on	on
	0x0280-0x029F	on	off
	0x0320-0x033F	off	on
	0x0340-0x035F	off	off

Some of the other installation steps require you to know which switch settings you used. If you reset the I/O switches, note the modified settings here for later reference.

Table 1–2. Your Switch Settings

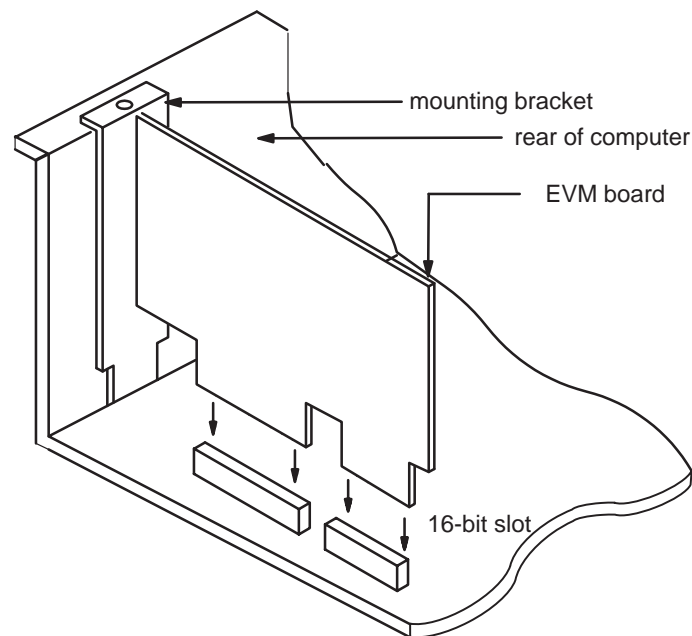
I/O switches		
Switch #	1	2
		

Setting the EVM board into your PC

After you've prepared the EVM board for installation, follow these steps.

- Step 1:** Turn off your PC's power and unplug the power cord.
- Step 2:** Remove the cover of your PC.
- Step 3:** Remove the mounting bracket from an unused 16-bit slot (see Figure 1–2).
- Step 4:** Install the EVM board in a 16-bit slot (see Figure 1–2).

Figure 1–2. EVM Installation



- Step 5:** Tighten down the mounting bracket.
- Step 6:** Replace the PC cover.
- Step 7:** Plug in the power cord and turn on the PC's power.

1.3 Step 2: Installing the Debugger Software

This section explains the simple process of installing the debugger software on a hard-disk system.

- ☐ Make a backup copy of each disk. (If necessary, refer to the DOS manual that came with your computer.)

- ☐ On your hard disk or system disk, create a directory named *c16dbgr*. This directory will contain the debugger software.

```
MD C:\c16dbgr
```

- ☐ Insert a product disk into drive A. Copy the debugger software onto the hard disk or system disk.

```
COPY A:\*.* C:\c16dbgr\*.* /V
```

Repeat this step for each product disk.

- ☐ 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 *evm16.exe* from your disk and change the name of *evm16w.exe* to *evm16.exe*:

```
del evm16.exe  
ren evm16w.exe evm16.exe
```

If you do not plan to use Microsoft Windows, delete *evm16w.exe* from your disk:

```
del evm16w.exe
```

1.4 Step 3: Setting Up the Debugger Environment

To ensure that your debugger works correctly, you must:

- ☐ Modify the PATH statement to identify the c16dbgr directory.
- ☐ Define environment variables so that the debugger can find the files it needs.
- ☐ Identify any nondefault I/O space used by the EVM.



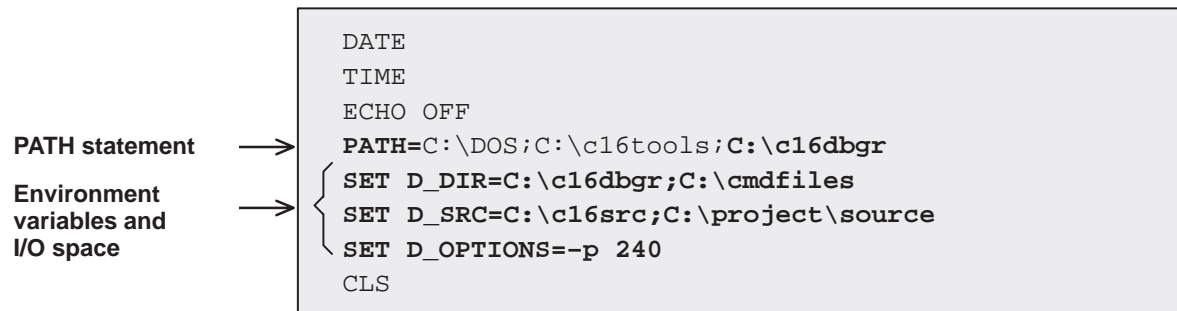
Not only must you do these things before you invoke the debugger for the first time, *you must do them any time you power up or reboot your PC.*

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.

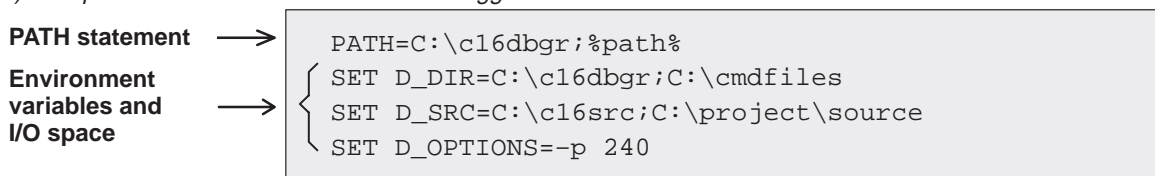
Figure 1–3 (a) shows an example of an autoexec.bat file that contains the suggested modifications (highlighted in bold type). Figure 1–3 (b) shows a sample batch file that you could create instead of editing the autoexec.bat file. (For the purpose of discussion, assume that this sample file is named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

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

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



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



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

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

PATH=C:\c16dbgr

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:\c16dbgr 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:\c16dbgr;%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 following tells 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.

- ☐ Set up the D_DIR environment variable to identify the c16dbgr directory:

SET D_DIR=C:\c16dbgr

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

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

- 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...

For example, if your 'C16 programs were in a directory named *c16src*, the D_SRC setup would be:

SET D_SRC=C:\c16src

- 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]

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 pathname	-p port address
-s	-t filename	-v

For more information about options, refer to Section 1.6 (page 1-12). Note that you can override D_OPTIONS by invoking the debugger with the **-x** option.

Identifying the correct I/O switches

Refer to your entries in Table 1-2 (page 1-5). If you didn't modify the I/O switches, skip this step.

If you modified the I/O switch settings, you must use the debugger's **-p** option to identify the I/O space that the EVM is using. You can do this each time you invoke the debugger, or you can specify this information by using the D_OPTIONS environment variable. Table 1-3 lists the nondefault I/O switch setting and the appropriate line that you can add to the autoexec.bat or init.bat file.

Table 1-3. Nondefault I/O Address Space

	switch #		Add this line to the batch file
	1	2	
0x0280—0x029F	on	off	SET D_OPTIONS=-p 280
0x0320—0x033F	off	on	SET D_OPTIONS=-p 320
0x0340—0x035F	off	off	SET D_OPTIONS=-p 340

Note: I/O Address Space

- 1) The 'C16 EVM uses 10 bytes of the PC I/O space.
- 2) If you didn't note the I/O switch settings, you may use a trial-and-error approach to find the correct -p setting. **If you use the wrong setting, you'll see this error message when you try to invoke the debugger:**

```
CANNOT INITIALIZE TARGET SYSTEM ! !  
- Check I/O configuration  
- Check cabling and target power
```

Note: Resetting the EVM

Never reset the 'C16 EVM with *c16reset* unless you have first loaded a valid object file to the EVM.

1.5 Using the Debugger 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.

1.6 Invoking the Debugger

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



```
evm16 [filename] [-options]
```

evm16 is the command that invokes the EVM version of the debugger.

filename is an optional parameter that names an object file. If you want the debugger to load a certain program during invocation, use *filename* to identify the name of the object file. The debugger looks for the file in the current directory; if the file isn't in the current directory, you must supply the entire pathname.

-options supply the debugger with additional information (see Table 1-4).

You can also specify filename and option information with the D_OPTIONS environment variable (described on page 1-10).

Table 1-4. Debugger Options


Option	Description																								
-b[bbbb]	<p>Screen-size options. By default, the debugger uses an 80-character-by-25-line screen. If you have a special graphics card, however, you can choose one of several larger screen sizes.</p> <table><tr><th>Option</th><th>Characters/Lines</th><th>Notes</th></tr><tr><td><i>none</i></td><td>80 by 25</td><td>This is the default display</td></tr><tr><td>-b</td><td>80 by 39</td><td>PC under Microsoft Windows</td></tr><tr><td></td><td>80 by 43 (EGA)</td><td>Use any EGA or VGA card</td></tr><tr><td></td><td>80 by 50 (VGA)</td><td></td></tr><tr><td>-bb</td><td>120 by 43</td><td rowspan="4">} Currently, the debugger supports these modes on a Video Seven VEGA Deluxe card.</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></table>	Option	Characters/Lines	Notes	<i>none</i>	80 by 25	This is the default display	-b	80 by 39	PC under Microsoft Windows		80 by 43 (EGA)	Use any EGA or VGA card		80 by 50 (VGA)		-bb	120 by 43	} Currently, the debugger supports these modes on a Video Seven VEGA Deluxe card.	-bbb	132 by 43	-bbbb	80 by 60	-bbbbb	100 by 60
Option	Characters/Lines	Notes																							
<i>none</i>	80 by 25	This is the default display																							
-b	80 by 39	PC under Microsoft Windows																							
	80 by 43 (EGA)	Use any EGA or VGA card																							
	80 by 50 (VGA)																								
-bb	120 by 43	} Currently, the debugger supports these modes on a Video Seven VEGA Deluxe card.																							
-bbb	132 by 43																								
-bbbb	80 by 60																								
-bbbbb	100 by 60																								
-i <i>pathname</i>	<p>Additional directories. -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>evm16 -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 1-10). 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 1–4. Debugger Options (Continued)

Option	Description															
-p <i>port address</i>	<p>Port address. -p identifies the I/O port address that the debugger uses for communicating with the EVM. If you used the EVM's default switch settings, you don't need to use the -p option. If you used nondefault switch settings, you must use -p. Refer to your entries in Table 1-2, page 1-5; depending on your switch settings, replace <i>port address</i> with one of these values:</p> <table><tr><th>Switch 1</th><th>Switch 2</th><th>Option</th></tr><tr><td>on</td><td>on</td><td>none needed</td></tr><tr><td>on</td><td>off</td><td>-p 280</td></tr><tr><td>off</td><td>on</td><td>-p 320</td></tr><tr><td>off</td><td>off</td><td>-p 340</td></tr></table>	Switch 1	Switch 2	Option	on	on	none needed	on	off	-p 280	off	on	-p 320	off	off	-p 340
Switch 1	Switch 2	Option														
on	on	none needed														
on	off	-p 280														
off	on	-p 320														
off	off	-p 340														
-s	<p>Load symbol table only. 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 (not the file's object code). This is similar to the debugger's SLOAD command.</p>															
-t <i>filename</i>	<p>New Initialization file. The -t option allows you to specify an initialization command file that will be used instead of init.cmd.</p>															
-v	<p>Load without symbol table. This option prevents the debugger from loading the entire symbol table when you load an object file. The debugger loads only the global symbols and later loads local symbols as it needs them. This speeds up the loading time and consumes less memory space.</p> <p>The -v option affects all loads, including loading when you invoke the debugger and loading with the LOAD command from within the debugger environment.</p>															
-x	<p>Ignore D_OPTIONS. -x tells the debugger to ignore any information supplied with D_OPTIONS.</p>															

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.

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.

Chapter 2

An Introductory Tutorial to the Debugger

This chapter provides a step-by-step, hands-on demonstration of the debugger's basic features. This is not the kind of tutorial that you can take home to read—it is effective only if you're sitting at your terminal, performing the lessons in the order that they're presented. This tutorial contains two sets of lessons (11 in the first, 9 in the second) and takes about one hour to complete.

Topic	Page
How to use this tutorial	2-2
A note about entering commands	2-2
An escape route (just in case)	2-3
Invoke the debugger and load the sample program's object code	2-3
Take a look at the display...	2-4
What's in the DISASSEMBLY window?	2-5
Select the active window	2-5
Size the active window	2-7
Zoom the active window	2-8
Move the active window	2-9
Scroll through a window's contents	2-10
Execute some code	2-11
Open a text file	2-11
Become familiar with the two debugging modes	2-12
Use the basic RUN command	2-14
Set some breakpoints	2-14
Watch some values and single-step through code	2-16
Run code conditionally	2-17
Clear the COMMAND window display area	2-18
Change some values	2-19
Define a memory map	2-20
Define your own command string	2-21
Close the debugger	2-21

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. . .

Important! This tutorial assumes that you have correctly and completely installed your EVM (including invoking any files or DOS commands as instructed in Chapter 1).


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** . 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.

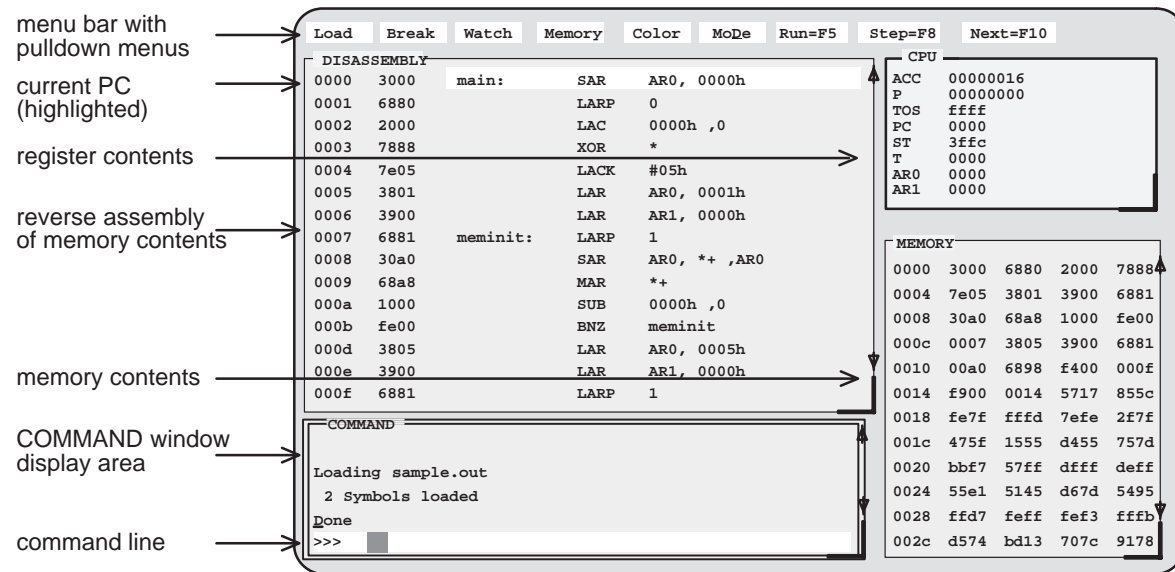
Important! This step assumes that you are using the default I/O address or that you have identified the I/O address with the **D_OPTIONS** environment variable (as described in the installation instructions in Chapter 1).

Invoke the debugger and load the sample program:

```
evm16 -b c:\c16dbgr\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).



- ☐ If you **don't** see a display, then your debugger or EVM 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 'C16 processor:


```
reset
```
 - 2) Load the sample program again:


```
load c:\c16dbgr\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 also starts at address 0.

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

mem 0@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 'C16 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 0 in data memory, enter:

mem 0@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.

lesson continues on the next page →



Make the CPU window the active window:

`win CPU` 

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.

Important! If you don't see a change in the appearance of the CPU window, 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.



Try This: Press the `F6` key to cycle 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 COMMAND window's border.
- 2) Click the left mouse button.

Be careful! If you point *inside* a 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 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.

Size the active window

This lesson shows you how to change the size of the active window.

Important! Be sure the CPU window is active.



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 4-1 on page 4-17.)



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.

Zoom the active window

Another way to resize the active window is to zoom it. Zooming the window makes it as large as possible.

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



Make the active window as large as possible:

zoom 

The window should now be as large as possible, taking up the entire display (except for the menu bar) and hiding all the other windows.

“Unzoom” or return the window to its previous size by entering the ZOOM command again. (Even though the COMMAND window is hidden by the CPU window, the ZOOM command will be recognized.)



zoom 

The window should now be back to the size it was before zooming.



Try This: You can also use the mouse to zoom the window.

Zoom the active window:

-  1) Point to the upper left corner of the active window.
-  2) Click the left mouse button.

Return the window to its previous size by repeating these steps.

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 4-2 on page 4-20.)



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 11.4 (page 11-34) summarizes the functions of all the special keys, key sequences, and how their effects vary for the different windows.

Execute some code

Now that you can find your way around the debugger interface, let's run some code—not the whole program, just a portion of it.

Execute a portion of the sample program:

`go 0x0006` 

You've just executed your program up to the address where meminit is declared. Notice how the display has changed:

- ☐ The current PC is highlighted in the DISASSEMBLY window.
- ☐ The values of the PC (and possibly some additional registers) are highlighted in the CPU window because they were changed by program execution.

Open a text file

You can display any text file in the FILE window.

Display a text file in the FILE window:

`file ..\autoexec.bat` 

This opens your autoexec file in the FILE window.

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

Become familiar with the two debugging modes

The debugger has two basic debugging modes:




- ☐ **Assembly mode** shows the DISASSEMBLY, CPU, MEMORY, and COMMAND windows.
- ☐ **Mixed mode** shows the windows listed above plus the FILE window.

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 mixed mode:

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

You should be in mixed mode now, and you should see the FILE and DISASSEMBLY windows.

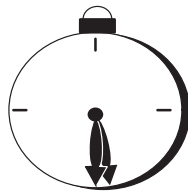


Try This: You can also switch modes by typing one of these commands:

asm switches to assembly mode
mix switches to mixed mode


Switch back to assembly mode before continuing:

asm 



Halfway Point

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

If you want to close the debugger, just type QUIT . When you come back, reinvoke the debugger and load the sample program (page 2-3). Then turn to page 2-14 and continue with the second set of lessons.

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 0x0006` 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.out` in the DISASSEMBLY window. If you aren't, enter:

`load sample.out` 

Set a breakpoint and run your program:


- 1) Scroll to the line with address 0x0006 in the DISASSEMBLY window and set a breakpoint on that line.

- a)  Point the mouse cursor at the statement at address 0x0006.
- 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 after executing the breakpointed line

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 through the statement at address 0x0006 in the DISASSEMBLY window. The current PC should be at the line marked `meminit`.

Clear the breakpoint:

-  1) Point the mouse cursor at the statement with breakpoint. (It should still be highlighted from setting the breakpoint.)
-  2) Click the left mouse button. *The line is no longer highlighted.*

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 the single-step example:

```
restart 
```

The debugger has another type of window called a WATCH window that's very useful in combination with single-step execution. 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. You can observe these values in a WATCH window.

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


Open a WATCH window:

```
wa pc,,x   
wa st, Status Register,x   
wa ar1, Aux. Register 1,x 
```

You may have noticed that the WA (watch add) command can have one or more parameters. The first parameter is the item that you're watching. The second parameter is an optional label. The third parameter defines the display format of the watched item. In this case, the **x** parameter tells the debugger to display the register contents in hexadecimal notation.

Now try out the single-step commands.

Single-step through the sample program:

```
step 50 
```



Observe the DISASSEMBLY and WATCH windows.

Run code conditionally

In the loop beginning at `meminit`, the program is doing a lot of work with AR1 (auxiliary register 1). You may want to check the value of AR1 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 two data items from the WATCH window (don't watch them anymore):

```
wd 2   
wd 1 
```


AR1 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 a breakpoint at the statement at address 0x0009.
- 2) Reset the program entry point:

```
reset   
restart 
```

- 3) Reset the value of AR1:

```
?ar1=0 
```

Now initiate the conditional run:


```
run ar1 < 0xa 
```

This causes the debugger to run through the loop as long as the value of AR1 is less than 0x000a. Each time the debugger encounters the breakpoint in the loop, it updates the value of AR1 in the WATCH window.

lesson continues on the next page →

When the conditional run completes, close the WATCH window.


Close the WATCH window:

WT 

Clear the COMMAND window display area

After entering several commands, 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 c16dbg
```

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

Change some values

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



Change a value in memory:

- 1) Display memory beginning with address 0x0001:
`mem 0x1`
- 2) Point to the contents of memory location 0x0001.
- 3) Click the left mouse button. This highlights the field to identify it as the field that will be edited.
- 4) Type 5754.
- 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 000b.
- 5) Press 3 times. You should now be pointing at the contents of register AR0.
- 6) Type 0.
- 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. When you invoked the debugger for this tutorial, the debugger automatically read a default memory map from the `init.cmd` file included in the `c16dbg` 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:


```
ml 
```

Look in the COMMAND window display area—you'll see a listing of the areas that are currently mapped. The 'C16 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 data memory:

```
ma 0x2000,1,0xff,RAM 
```


Define your own command string

If you find that you often enter a command with the same parameters, or often enter the same commands in sequence, you will find it helpful to have a short-hand method for entering these commands. The debugger provides an *aliasing* feature that allows you to do this.

This lesson shows you how you can define an alias to set up a memory map.

Define an alias for setting up the memory map:

- 1) Use the ALIAS command to associate a nickname with the commands used for defining a memory map:

```
alias mymap,"mr;ma 0x0000,0,0xffff,ROM;  
ma 0x0000,1,0xff,RAM;ml" 
```

(Note: Because of space constraints, the command is shown on two lines.)

- 2) Now, to use this memory map, just enter the alias name:

```
mymap 
```


This is equivalent to entering the following four commands:

```
mr  
ma 0x0000,0,0xffff,ROM  
ma 0x0000,1,0xff,RAM  
ml
```

Close the debugger

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

Close the debugger and return to the operating system (or Microsoft Windows):

```
quit 
```


Chapter 3

Overview of a Code Development and Debugging System

The 'C16 debugger has an advanced software interface that helps you to develop, test, and refine assembly language programs. The debugger serves as the programmer interface to the TI 'C16 EVM.

This chapter provides an overview of the debugger and describes the 'C16 code development environment.

Topic	Page
3.1 Description of the 'C16 Debugger	3-2
Key features of the debugger	3-3
3.2 Developing Code for the 'C16	3-5
3.3 Preparing Your Program for Debugging	3-7
3.4 Debugging 'C16 Programs	3-8

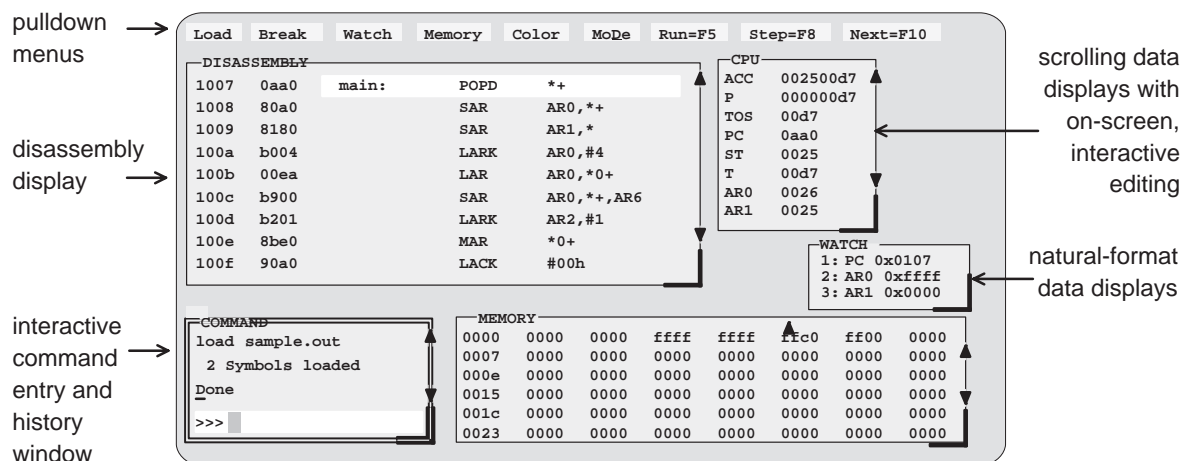
3.1 Description of the 'C16 Debugger

The 'C16 debugger improves productivity by allowing you to debug your assembly language programs through its high-level features.

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 3–1 identifies several features of the debugger display.

Figure 3–1. The Debugger Display



Key features of the debugger

- ☐ **Fully configurable, state-of-the-art, window-oriented interface.** The 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 memory contents and pointers.
- ☐ **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 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 other systems.
- ☐ **Flexible command entry.** There are a variety of ways to enter commands. You can type commands or use a mouse, function keys, or the menu selections; 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 either 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.

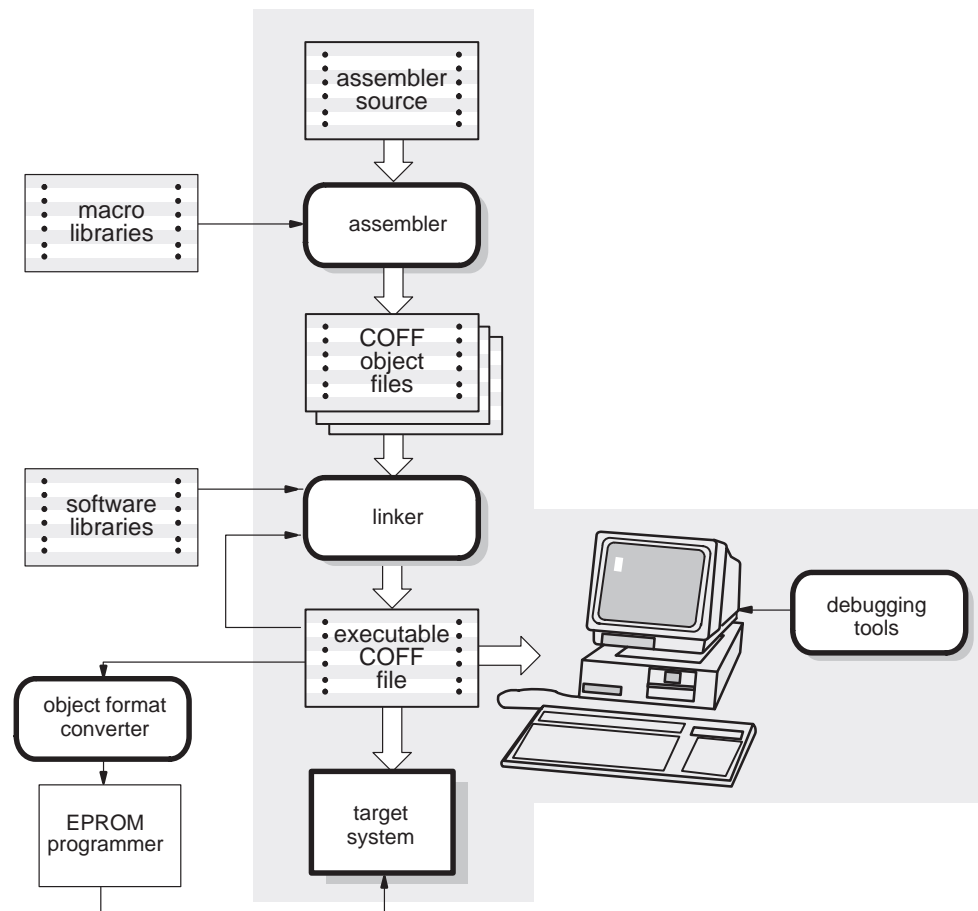
- ❑ **Variety of screen sizes.** The debugger has a default configuration set up for a typical PC display, with 25 lines by 80 characters. If you use a sophisticated graphics card, you can take advantage of the debugger's additional screen sizes. A larger screen size allows you to display more information and provides you with more screen space for organizing the display—bringing the benefits of workstation displays to your PC.

- ❑ **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. 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.

3.2 Developing Code for the 'C16

The 'C16 is supported by a complete set of hardware and software development tools, including an assembler and a linker. Figure 3–2 illustrates the development flow for the 'C16. The figure highlights the most common paths of software development; the other portions are optional.

Figure 3–2. 'C16 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 symbolic debugging.

The following list describes the tools shown in Figure 3–2.

assembler

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

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 your target system. You can use the debugger as a software interface to your EVM to refine and correct your code.

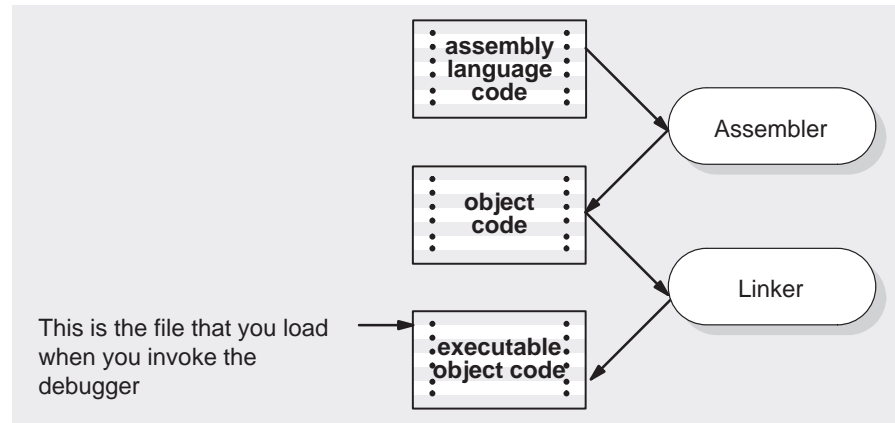
object
format
converter

The **object format converter** converts a COFF object file into an Intel, Tektronix, or TI-tagged object-format file that can be downloaded to an EPROM programmer.

3.3 Preparing Your Program for Debugging

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

Figure 3–3. Steps You Go Through to Prepare a 'C16 Program



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

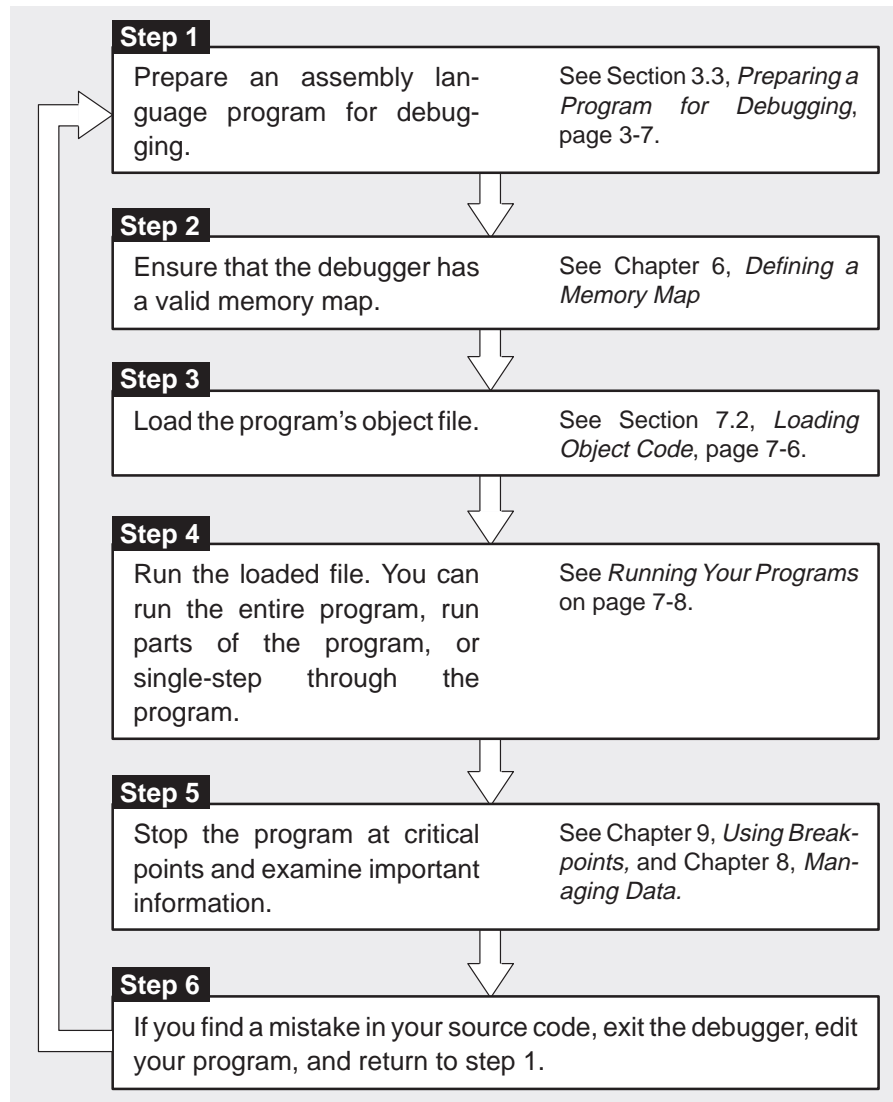
This produces an object file that you can load into the debugger.

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

Extension	File Type	File Description
.asm	assembly language source	assembled and linked
.s* (any extension that begins with s)	assembly language source	assembled and linked
.o* (any extension that begins with o)	object file	linked

3.4 Debugging 'C16 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.



Chapter 4

The Debugger Display

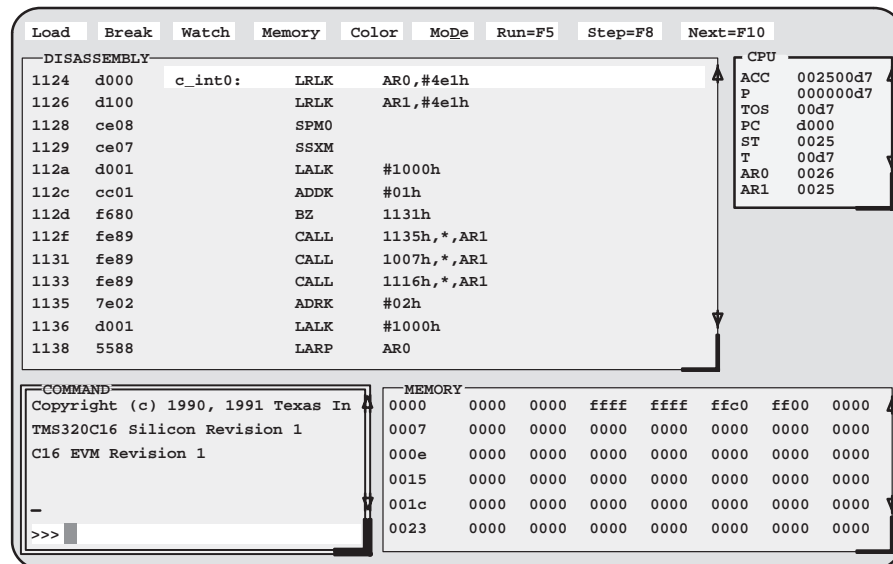
The debugger has a window-oriented display. This chapter shows what windows can look like and describes the basic types of windows that you'll use.

Topic	Page
4.1 The Debugger Modes	4-2
4.2 Descriptions of the Different Kinds of Windows and Their Contents	4-3
COMMAND window	4-4
DISASSEMBLY window	4-5
FILE window	4-5
MEMORY window	4-6
CPU window	4-11
WATCH window	4-12
4.3 Cursors	4-13
4.4 The Active Window	4-13
Identifying the active window	4-14
Selecting the active window	4-14
4.5 Manipulating Windows	4-16
Resizing a window	4-16
Zooming the active window	4-18
Moving a window	4-19
4.6 Manipulating a Window's Contents	4-22
Scrolling through a window's contents	4-22
Editing the data displayed in windows	4-24
4.7 Closing a Window	4-24

4.1 The Debugger Modes

The 'C16 version of the debugger is for viewing assembly language programs. When you invoke the debugger, you'll see a display similar to the one shown in Figure 4–1.

Figure 4–1. Typical Debugger Display



This is the **assembly** mode of the debugger. Windows that are automatically displayed while using the assembly mode include the MEMORY window, the DISASSEMBLY window, the CPU window, and the COMMAND window. If you choose, you can also open a WATCH window.

If you open a text file using the FILE command, the debugger automatically switches to **mixed** mode. Mixed mode displays all of the windows shown in assembly mode, plus the FILE window.

4.2 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.

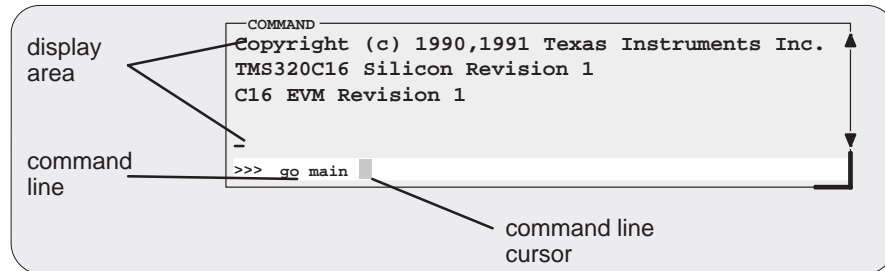
Every window is identified by a name in its upper left corner. Each type of window serves a specific purpose and has unique characteristics. There are six different windows, divided into four general categories:

- ☐ 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.
- ☐ The **DISASSEMBLY window** displays the disassembly (assembly language version) of program-memory contents.
- ☐ The **FILE window** displays any text file that you want to display.
- ☐ **Data-display windows** are for observing and modifying various types of data. There are three data-display windows:
 - The **MEMORY** window displays the contents of a range of memory.
 - The **CPU** window displays the contents of 'C16 registers.
 - 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 4.4, *The Active Window*, on page 4-13.

The remainder of this section describes the individual windows.

COMMAND window



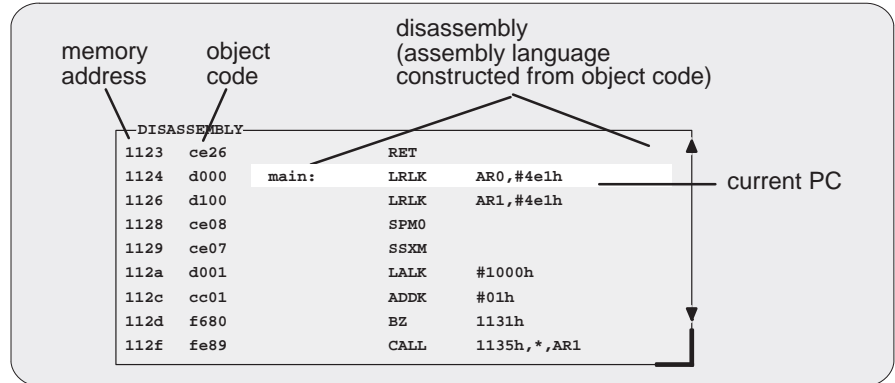
- Purpose**
- ☐ Provides an area for entering commands
 - ☐ Provides an area for echoing commands and displaying command output, errors, and messages
- Editable?** Command line is editable; the display area isn't
- Created** Automatically
- Affected by**
- ☐ All commands entered on the command line
 - ☐ 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 an internal list (a command history) of the last 50 commands that you entered. You can select and re-enter commands from the list without retyping them. (For more information on using the command history, see *Using the command history*, page 5-5).
- ☐ **Display area.** This area 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 5, *Entering and Using Commands*.

DISASSEMBLY window



Purpose Displays the disassembly (or reverse assembly) of program-memory contents

Editable? No; pressing the edit key (**F9**) or the left mouse button sets a breakpoint on an assembly language statement

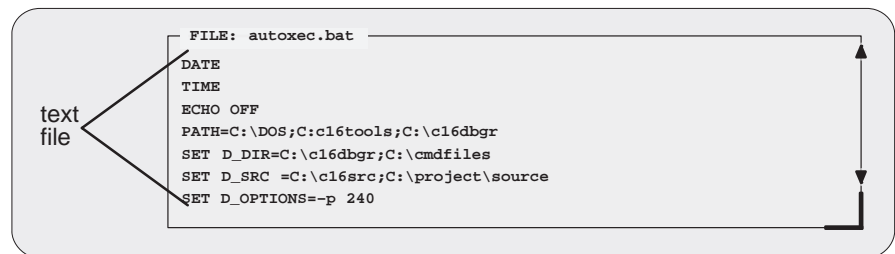
Created Automatically

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

FILE window



Purpose Shows any text file you want to display

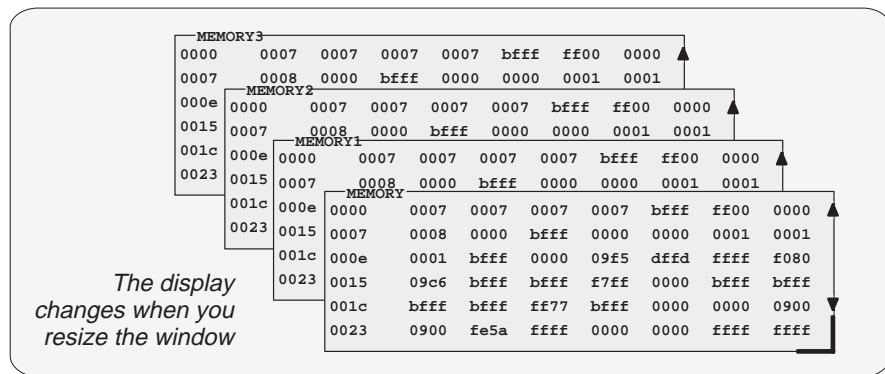
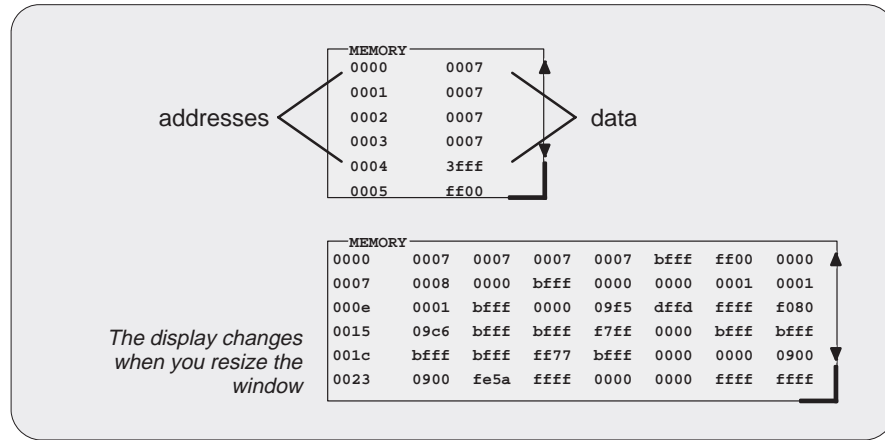
Editable? No

Created With the FILE command

Affected by FILE command

You can use the FILE command to display the contents of any file within the FILE window.

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 eight 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 eighth value (first value in the second row) is at address 0x0007, the ninth 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).

Created ☐ Automatically (the default MEMORY window only)
☐ With the MEM command

Affected by MEM commands:
☐ MEM
☐ MEM0
☐ MEM1
☐ MEM2
☐ MEM3

There are four MEMORY windows available: the default MEMORY window, MEMORY1, MEMORY2, and MEMORY3. Notice the default window does not have an extension number in its name; this is because MEMORY1, MEMORY2, and MEMORY3 are pop-up windows and can be opened and

closed throughout your debugging session. Having four windows allows you to view four different memory ranges.

Figure 4–2. Address and Data Columns in a MEMORY Window

MEMORY							
0000	0007	0007	0007	0007	bfff	fff0	0000
0007	0008	0000	bfff	0000	0000	0001	0001
000e	0001	bfff	0000	09f5	dfdf	ffff	f080
0015	09c6	bfff	bfff	f7ff	0000	bfff	bfff
001c	bfff	bfff	ff77	bfff	0000	0000	0900
0023	0900	ff5a	ffff	0000	0000	ffff	ffff

A MEMORY window has two parts:

- ❑ **Addresses.** The first column of numbers identifies the addresses of the first column of displayed data. Resizing the MEMORY window to change the number of data columns will not effect the address column; the debugger always displays 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 MEMORY window above has four columns of data, so each new address is incremented by four. Although the window shows seven columns of data, there is still only one column of addresses; the first value is at address 0x000000, the second at address 0x0001, etc.; the eighth value (first value in the second row) is at address 0x0007, the ninth at address 0x0008, etc.

As you run programs, some memory values change as the result of program execution. The debugger highlights the 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).

To create a pop-up MEMORY window or modify the memory block displayed in the current window, use the MEM (*Modify MEMORY Window Display*) command.

❑ **Creating a new MEMORY window.**

If the default MEMORY window is the only window open and you want to open another MEMORY window, enter the MEM command with the appropriate extension number.

mem# *address*

For example, if you want to create a new memory window starting at the address 0x8000, you would enter:

mem1 0x8000

This displays a new window, MEMORY1, showing the contents of memory starting at the address 0x8000.

The 'C16 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]

MEMORY [PROG]								
0000	ff80	1000	0000	0000	0000	0000	0000	0000
0007	0000	0000	0000	0000	0000	0000	0000	0000
000e	0000	0000	0000	0000	0000	0000	0000	0000
0015	0000	0000	0000	0000	0000	0000	0000	0000
001c	fefa	fdcf	7175	1454	57d3	5555	ffff	

□ Modifying the memory block displayed in the current MEMORY window.

This identifies a new starting address for the block of memory displayed in the current MEMORY window. The debugger displays the contents of memory at *address* in the first data position in your MEMORY window. The end of the range is defined by the size of the window.

- If the only memory window open is the default MEMORY window, you can view different memory locations by entering the following:

mem *address*

- If more than one MEMORY window is open and you want to view a different block of memory in the default MEMORY window, you can distinguish the default window from the pop-up windows by entering:

mem0 *address*

To view different memory locations in the alternate, or pop-up MEMORY windows, use the MEM command with the appropriate extension number

on the end. For example, **mem1** 0x8000, will show you the block of memory starting at address 0x8000 in the MEMORY1 window; **mem2** 0x8000 will show you the same block of memory (starting at address 0x8000) but in the MEMORY2 window, etc.

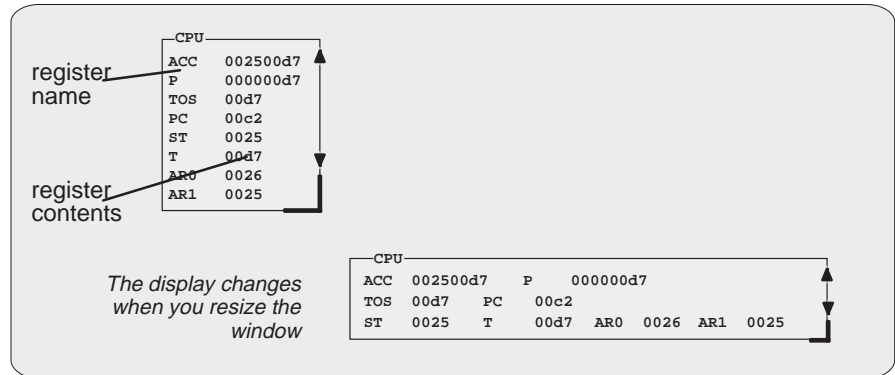
You can close and reopen the alternate MEMORY windows as often as you like.

- ☐ Closing the window is a two-step process:
 - 1) Make the appropriate MEMORY window the active window (see Section 4.4, *The Active Window*, on page 4-13).
 - 2) Press **F4**.

Remember, you cannot close the default MEMORY window.

- ☐ To reopen a pop-up MEMORY window after you've closed it, enter the MEM command with its appropriate extension number (see the discussion above).

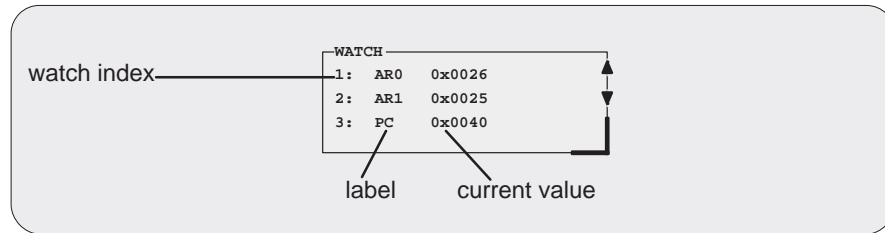
CPU window



<i>Purpose</i>	Shows the contents of the 'C16 registers
<i>Editable?</i>	Yes—you can edit the value of any displayed register
<i>Created</i>	Automatically
<i>Affected by</i>	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.

WATCH window



<i>Purpose</i>	Displays the values of selected expressions
<i>Editable?</i>	Yes—you can edit the value of any expression whose value specifies a storage location (in registers or memory)
<i>Created</i>	With the WA command
<i>Affected by</i>	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).

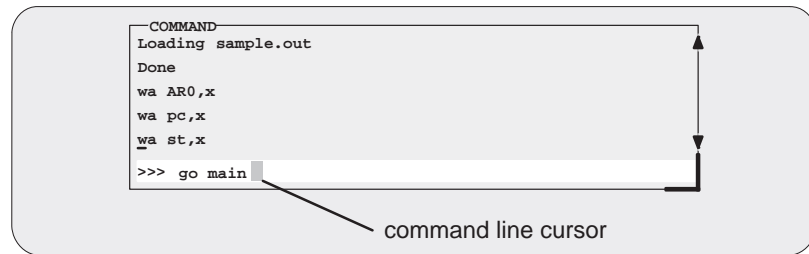
To delete individual entries from the WATCH window, use the WD command. To delete all entries at once and close the WATCH window, use the WR command.

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.

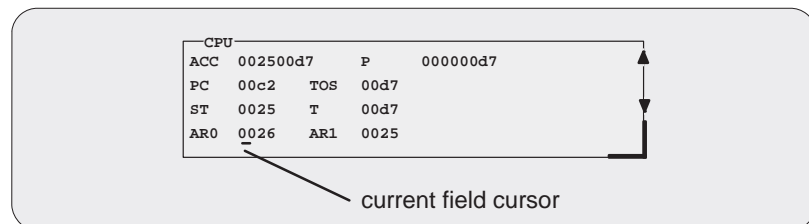
4.3 Cursors

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.



4.4 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**.

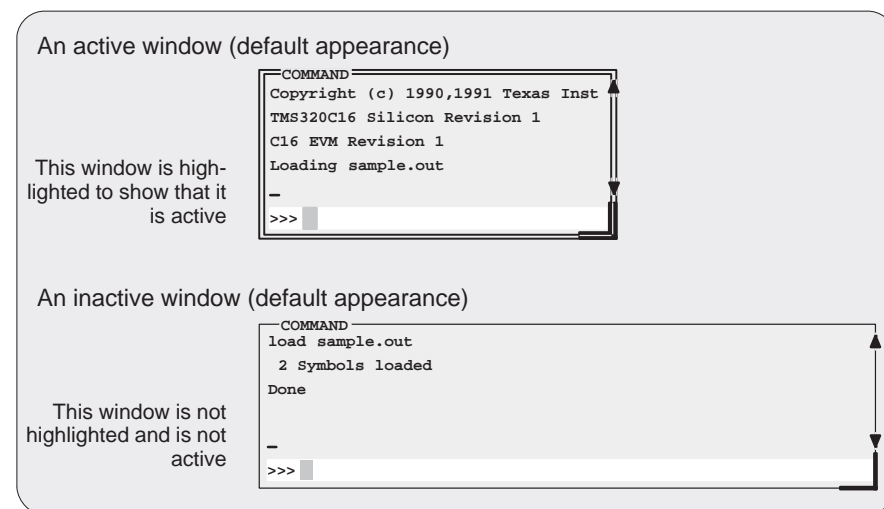
You can move, resize, or close *only one window at a time*; thus, only one window at a time can be the **active window**. Whether or not a window is active doesn't affect the debugger's ability to update information in a window—it affects only your ability to manipulate a window.

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 4–3 illustrates the default appearance of an active window and an inactive window.

Figure 4–3. 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 **SC** to get out of this.*

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

Press the button again to clear the breakpoint.



- F6** This key cycles 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 COMMAND window as the active window, you could enter either of these two commands:

win COMMAND 
or **win** CO 

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, 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.

4.5 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 4.4 (page 4-13).

Resizing a window

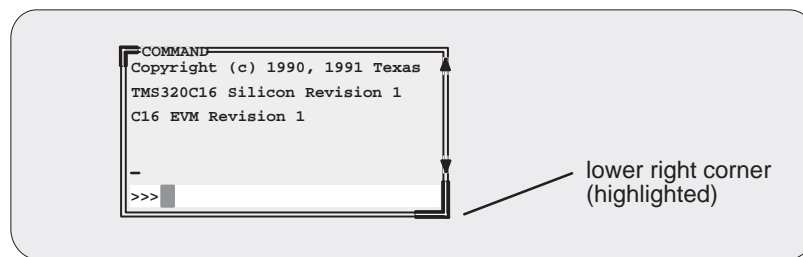
The minimum window size is three lines by four characters. The maximum window size varies, depending on which screen size you're using, but you can't make a window larger than the screen.

There are two basic ways to resize a window:

- ☐ By using the mouse
- ☐ 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 4–1 lists the minimum and maximum window sizes.

Table 4–1. 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 running under Microsoft Windows



[‡] PC with EGA card

[§] 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. The easiest way to make a window as large as possible is to zoom it, as described on page 4-18.

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

```
win CPU 
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  or .

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

```
win CPU   
size   
     SC
```

Zooming a window

Another way to resize the active window is to zoom it. Zooming a window makes it as large as possible, so that it takes up the entire display (except for the menu bar) and hides all the other windows. Unlike the SIZE command, zooming is not affected by the window's position in the display.

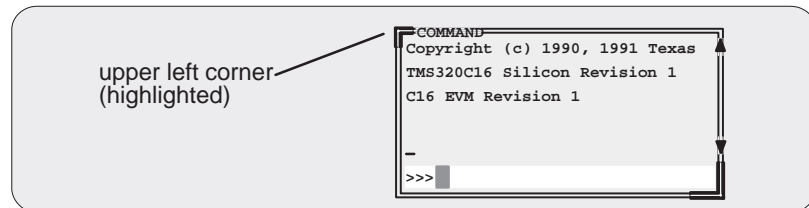
To “unzoom” a window, repeat the same steps you used to zoom it. This will return the window to its prezoom size and position.

There are two basic ways to zoom a window:

- ☐ By using the mouse
- ☐ By using the ZOOM command



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



- 2) Click the left mouse button.



zoom You can also use the ZOOM command to zoom/unzoom the window. The format for this command is:

zoom

Moving a window

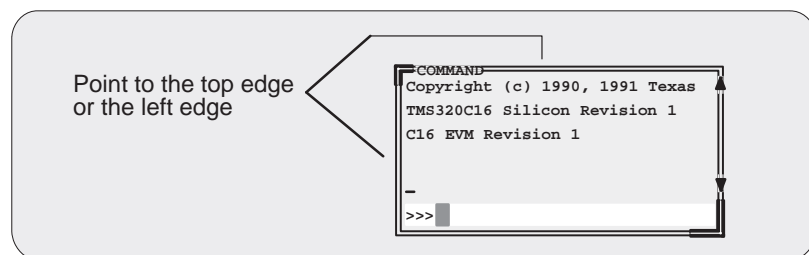
The windows in the debugger display don't have fixed positions—you can move them around.

There are two ways to move a window:

- ☐ By using the mouse
- ☐ 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 4–2 lists the minimum and maximum XY positions.

Table 4–2. 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 running under Microsoft Windows

[‡] PC with EGA card

[§] 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  or  .

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

```
win COM   
move   
       SC
```

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.

4.6 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 4.4 (page 4-13).

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:




- ☐ By using the mouse to scroll the contents of the window.
- ☐ By using the 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.



The page-up key scrolls up through the window contents, one window length at a time.



The page-down key scrolls down through the window contents, one window length at a time.



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.



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, 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 8.3, *Basic Methods for Changing Data Values*, page 8-4.)

Note: “Editing” the DISASSEMBLY Window

In the DISASSEMBLY window, 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. 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 DISASSEMBLY window.

4.7 Closing a Window

The debugger opens various windows on the display when you invoke the debugger. Most of the windows remain open—you can't close them. However, you can close the WATCH by entering:

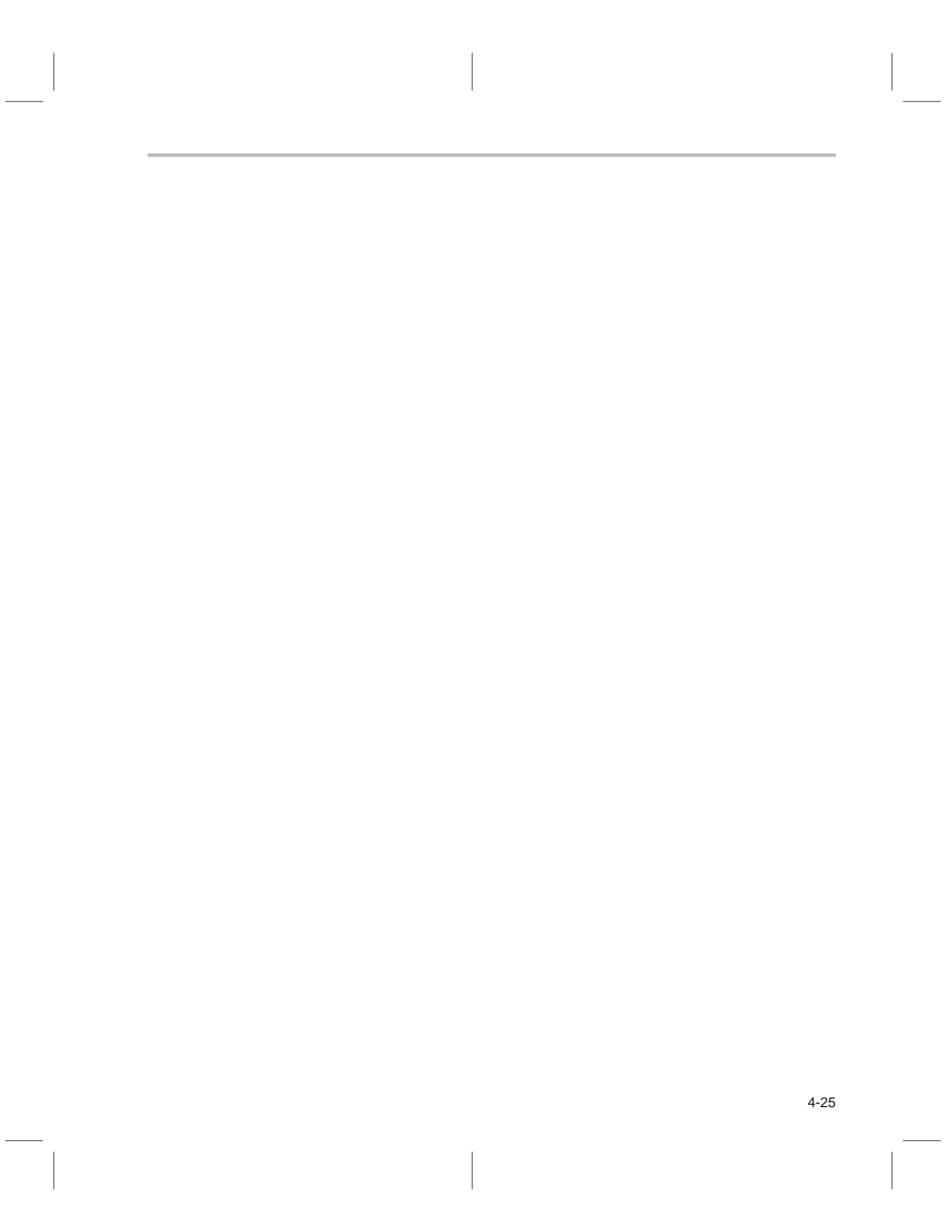
WT 

The debugger opens various windows on the display when you invoke the debugger. Most of the windows remain open—you can't close them. However, you can close the WATCH and the alternate MEMORY windows.

- ☐ To close the WATCH window, enter:

WT 

- ☐ To close a MEMORY window,
 - 1) Make the appropriate MEMORY window the active window.
 - 2) Press **[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 menu selections, with a mouse, and with function keys. Using the mouse and function keys differs 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.

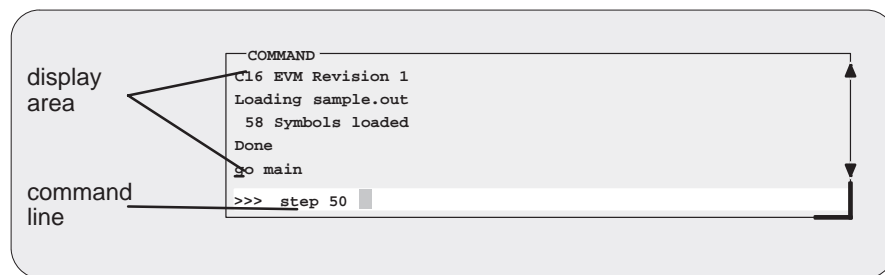
Topic	Page
5.1 Entering Commands From the Command Line	5-2
How to type in and enter commands	5-3
Sometimes, you can't type a command	5-4
Using the command history	5-5
Clearing the display area	5-5
Recording information from the display area	5-6
5.2 Using the Menu Bar and the Pulldown Menus	5-7
Using the pulldown menus	5-8
Escaping from the pulldown menus	5-9
Entering parameters in a dialog box	5-10
Using menu bar selections that don't have pulldown menus	5-11
5.3 Entering Commands From a Batch File	5-12
Echoing strings in a batch file	5-13
Controlling command execution in a batch file	5-13
5.4 Defining Your Own Command Strings	5-15
5.5 Entering Operating-System Commands	5-18
Entering a single command from the debugger command line	5-18
Entering several command from a system shell	5-19
Additional system commands	5-19

5.1 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 11 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 5–1 shows the COMMAND window.

Figure 5–1. 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 5–1 shows that a STEP 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 5–1 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 **CTRL** 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 50 commands 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...
Repeat the last command that you entered	F2
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

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

Recording information from the display area

The information shown in the display area of the COMMAND window can be written to a log file. The log file is a system file that contains commands you've entered, their results, and error or progress messages. To record this information in a log file, use the DLOG command.

Log files can be executed by using the TAKE command. When you use DLOG to record the information from the display area of the COMMAND window, the debugger automatically precedes all error or progress messages and command results with a semicolon to turn them into comments. This way, you can easily re-execute the commands in your log file by using the TAKE command.

- ☐ To begin recording the information shown in the display area of the COMMAND window, use:

dlog *filename*

This command opens a log file called *filename* that the information is recorded into.

- ☐ To end the recording session, enter:

dlog close 

If necessary, you can write over existing log files or append additional information to existing files. The extended format for the DLOG command is:

dlog *filename* [{**a** | **w**}]

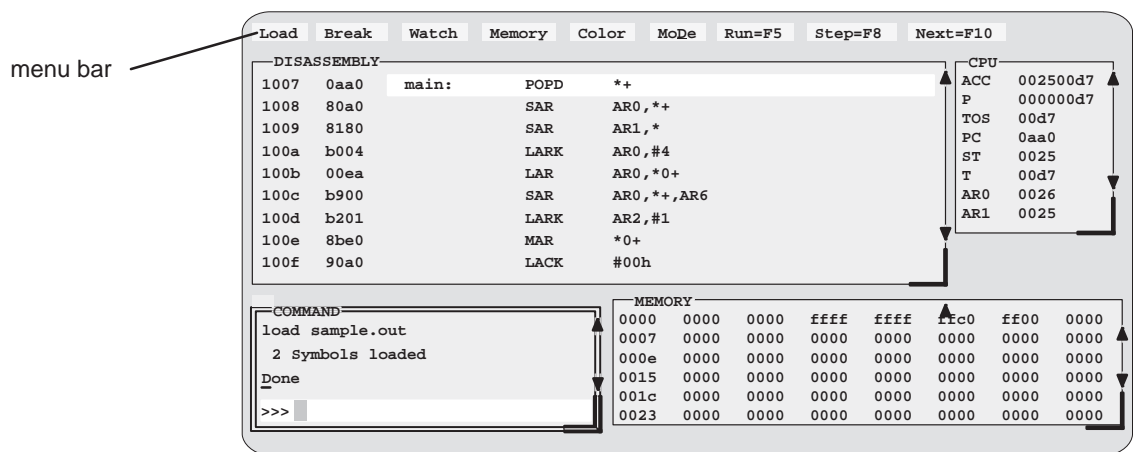
The optional parameters of the DLOG command control how the log file is created and/or used:

- ☐ **Creating a new log file.** If you use the DLOG command without one of the optional parameters, the debugger creates a new file that it records the information into. If you are recording to a log file already, entering a new DLOG command and filename closes the previous log file and opens a new one.
- ☐ **Appending to an existing file.** Use the **a** parameter to open an existing file to which to append the information in the display area.
- ☐ **Writing over an existing file.** Use the **w** parameter to open an existing file to write over the current contents of the file. Note that this is the default action if you specify an existing filename without using either the **a** or **w** options; you will lose the contents of an existing file if you don't use the append (a) option.

5.2 Using the Menu Bar and the Pulldown Menus

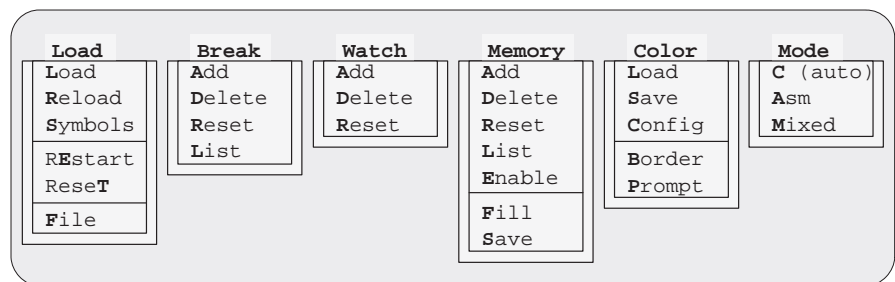
In both of the debugger displays, you'll see a menu bar at the top of the screen. The pulldown menus offer you an alternative method for entering many of the debugger commands. Figure 5–2 points out the menu bar. There are several ways to use the selections on the menu bar, depending on whether the selection has a pulldown menu or not.

Figure 5–2. The Menu Bar in the Debugger Display



Several of the selections on the menu bar have pulldown menus; if they could all be pulled down at once, they'd look like Figure 5–3.

Figure 5–3. All of the Pulldown 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 commands 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 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 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 .
 - 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:

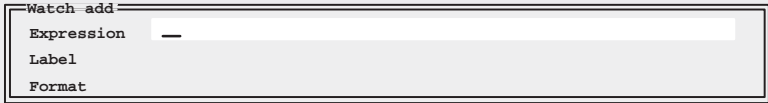
- ☐ If you press  in the middle of a string of text, the debugger truncates the string at that point.
- ☐ When you display a dialog box for the first time during a debugging session, the parameter fields are empty. When you bring up the same dialog box again, though, the box displays the last values 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 three parameters:


wa *expression* [, *label*] [, *display format*]

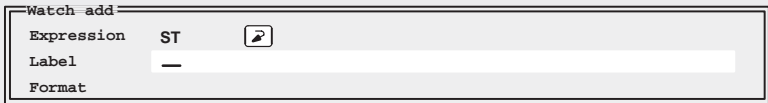
When you select Add from the menu, the debugger displays a dialog box that asks you for this parameter information. The dialog box looks like this:




Watch-add



Expression	
Label	
Format	

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



Watch-add

Expression	ST 
Label	
Format	

In this case, the next two parameters (*label* and *format*) are optional. If you want to enter a parameter, you may do so; if you don't want to use these parameters, don't type anything in their fields—just press . When you've entered  for the final parameter, 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.

5.3 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 would like to suppress the echoing and updating, use the value 0 for the *suppress echo flag* parameter.
 - 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.

Echoing strings in a batch file

When executing a batch file, you can display a string to the COMMAND window by using the ECHO command. The syntax for the command is:

echo *string*

This displays the *string* in the COMMAND window display area.

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

echo **Creating new memory map**

(Notice that the string should not be in quotes.)

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

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

Note that any leading blanks in your string are removed when the ECHO command is executed.

Controlling command execution in a batch file

In batch files, you can control the flow of debugger commands. You can choose to conditionally execute debugger commands or set up a looping situation by using IF/ELSE/ENDIF or LOOP/ENDLOOP, respectively.

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

```
if Boolean expression
    debugger command
    debugger command
.
.
[else
    debugger command
    debugger command
.
.]
endif
```

The debugger includes some predefined constants for use with IF. These constants evaluate to 0 (false) or 1 (true). Table 5–1 shows the constants and their corresponding tools.

Table 5–1. Predefined Constants for Use With Conditional Commands

Constant	Debugger Tool
\$\$EVM\$\$	evaluation module

If the Boolean expression evaluates to true (1), the debugger executes all commands between the IF and ELSE or ENDIF. Note that the ELSE portion of the command is optional. (See Chapter 12 for more information about expressions and expression analysis.)

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

```

loop expression
debugger command
debugger command
.
.
endloop

```

These looping commands evaluate in the same method as in the run conditional command expression. (See Chapter 12 for more information about expressions and expression analysis.)

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

```

loop 10
step 50
.
.
.
endloop

```

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

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

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

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

```
loop !0
step
? PC
? AR0
endloop
```

The IF/ELSE/ENDIF and LOOP/ENDLOOP commands work with the following conditions:

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

5.4 Defining Your Own Command Strings

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

To do this, use the ALIAS command. The syntax for this command is:

alias [*alias name* [, "*command string*"]]

The primary purpose of the ALIAS command is to associate the *alias name* with the debugger command you've supplied as the *command string*. However, the ALIAS command is versatile and can be used in several ways:

- ❑ **Aliasing several commands.** The *command string* can contain more than one debugger command—just separate the commands with semicolons.

For example, suppose you always began a debugging session by loading the same object file and running to a certain point in the code. You could define an alias to do all these tasks at once:

```
alias init,"load test.out;go main"
```

Now you could enter `init` instead of the two commands listed within the quote marks.

- ❑ **Supplying parameters to the command string.** The *command string* can define parameters that you'll supply later. To do this, use a percent sign and a number (%1) to represent the parameter that will be filled in later. The numbers should be consecutive (%1, %2, %3) unless you plan to reuse the same parameter value for multiple commands.

```
alias mfil,"fill %1, %2, %3, %4;mem %1"
```

Then you could enter:

```
mfil 0xff80,1,0x18,0x1122
```

The first value (0xff80) would be substituted for the first FILL parameter and the MEM parameter (%1). The second, third, and fourth values would be substituted for the second, third, and fourth FILL parameters (%2, %3, and %4).

- ❑ **Listing all aliases.** To display a list of all the defined aliases, enter the ALIAS command with no parameters. The debugger will list the aliases and their definitions in the COMMAND window.

For example, assume that the `init` and `mfil` aliases had been defined as shown in the previous two examples. If you entered:


```
alias 
```

you'd see:

Alias	Command
INIT	--> load test.out;go main
MFIL	--> fill %1,%2,%3,%4;mem %1

- ❑ **Finding the definition of an alias.** If you know an alias name but are not sure of its current definition, enter the ALIAS command with just an alias name. The debugger will display the definition in the COMMAND window.

For example, if you had defined the `init` alias as shown in the first example above, you could enter:

```
alias init 
```

Then you'd see:

```
"INIT" aliased as "load test.out;go main"
```

- ☐ **Nesting alias definitions.** You can include a defined alias name in the *command string* of another alias definition. This is especially useful when the command string would be longer than the debugger command line.

Note: Limitation on Command String Length

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

- ☐ **Redefining an alias.** To redefine an alias, re-enter the `ALIAS` command with the same alias name and a new command string.
- ☐ **Deleting aliases.** To get rid of a single alias, use the `UNALIAS` command:

```
unalias alias name
```

To delete *all* aliases, enter the `UNALIAS` command with an asterisk instead of an alias name:

```
unalias *
```

Note that the `*` symbol *does not* work as a wildcard.

Note: Limitations of Alias Definitions

- ☐ Alias definitions are lost when you exit the debugger. If you want to reuse aliases, define them in a batch file.
- ☐ Individual commands within a command string are limited to an expanded length of 132 characters. The expanded length of the command includes the length of any substituted parameter values.

5.5 Entering Operating-System Commands

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

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

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

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


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

Entering a single command from the debugger command line

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

system "copy a:\backup\sample.out sample.out" 

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

- 0** The debugger immediately returns to the debugger environment after the last item of information is displayed.
- 1** The debugger does not return to the debugger environment until you press . (This is the default.)

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

```
1 File(s) copied
Type Carriage Return To Return To Debugger
```

The message would be displayed until you pressed .


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

system "copy a:\backup\sample.out sample.out",0 

Entering several commands from a system shell

If you need to enter several commands, enter the SYSTEM command without parameters. The debugger will open a system shell and display the operating-system prompt. At this point, you can enter any operating-system command.

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

exit 

Note: Memory Limitation When Using a System Shell

On PC systems, available memory may limit the operating-system commands that you can enter from a system shell. For example, you would not be able to invoke another version of the debugger.

Additional system commands

The debugger also provides separate commands for changing directories and for listing the contents of a directory.



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 LOAD and TAKE commands).

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*.

Chapter 6

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
6.1 The Memory Map: What It Is and Why You Must Define It	6-2
Defining the memory map in a batch file	6-2
Potential memory map problems	6-3
6.2 A Sample Memory Map	6-3
6.3 Identifying Usable Memory Ranges	6-4
6.4 Enabling Memory Mapping	6-5
6.5 Checking the Memory Map	6-5
6.6 Modifying the Memory Map During a Debugging Session	6-6
Returning to the original memory map	6-7

6.1 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.

Note: Accessing Nonexistent Memory

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

A special default initialization batch file included with the debugger package defines a memory map for your version of the debugger. This memory map may be sufficient when you first begin using the debugger. However, the debugger provides a complete set of memory-mapping commands that let you modify the default memory map or define a new memory map.

You can define the memory map interactively by entering the memory-mapping 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 defining a memory map is to put the memory-mapping commands in a batch file.

Defining the memory map in a batch file

There are two methods for defining the memory map in a batch file:

- ☐ You can redefine the memory map defined in the initialization batch file.
- ☐ You can define a memory map in a separate batch file of your own.

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

- 1) When you invoke the debugger, it checks to see if you've used the `-t` debugger option. The `-t` option allows you to specify a batch file other than the initialization batch file shipped with the debugger. If it finds the `-t` option, the debugger reads and executes the specified file.
- 2) If you don't use the `-t` option, the debugger next looks for the default initialization batch file. For the EVM, this file is named `evminit.cmd`.
If the debugger finds the `evminit.cmd` file, it reads and executes the file.
- 3) If the debugger does not find the `-t` option or the initialization batch file, it looks for a file called `init.cmd`. This allows you to have one initialization batch file for more than one debugger tool.

Potential memory map problems

The following are potential problems you may experience if the memory map isn't correctly defined and enabled:

- ☐ **Accessing invalid memory addresses.** If you don't supply a batch file containing memory-map commands, 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.)
- ☐ **Accessing an undefined or protected area.** 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.

6.2 A Sample Memory Map

Figure 6–1 shows the memory map commands that are defined in the EVM version of init.cmd. You can use the file as is, edit it, or create your own command file.

Figure 6–1. Sample Memory Map for Use With a 'C16 EVM

```
MA 0x0, 0,0x10000, RAM ;External program RAM
MA 0x0, 1,0x100, RAM ;Internal Data RAM
```

The MA (map add) commands define valid memory ranges and identify the read/write characteristics of the memory ranges.

6.3 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, 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 or data) that the range occupies:

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

- ☐ 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

6.4 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**

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
```

6.5 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 EVM default memory map and you enter the ML command, the debugger displays this:

<u>Page</u>	<u>Memory range</u>	<u>Attributes</u>
0	0000 - ffff	READ WRITE
1	0000 - 00ff	READ WRITE

Page 0 = program memory
 Page 1 = data memory

starting address ending address

6.6 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 or data memory. If you supply an *address* that is not the starting address of a range, the debugger displays this error message in the display area of the COMMAND window:

Specified map not found

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

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

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 6-4.

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		<i>Reset the memory map</i>
take mem.map		<i>Reread the default memory map</i>

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.

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. Many of the commands described in this chapter can also be executed from the Load pulldown menu.

Topic	Page
7.1 Displaying Your Source Programs (or Other Text Files)	7-2
Displaying assembly language code	7-2
Modifying assembly language code	7-3
Additional information about modifying assembly language code	7-5
Displaying other text files	7-5
7.2 Loading Object Code	7-6
Loading code while invoking the debugger	7-6
Loading code after invoking the debugger	7-6
7.3 Where the Debugger Looks for Source Files	7-7
7.4 Running Your Programs	7-8
Defining the starting point for program execution	7-8
Running code	7-9
Single-stepping through code	7-9
Running code while emulation is disabled	7-11
Running code conditionally	7-12
7.5 Halting Program Execution	7-12

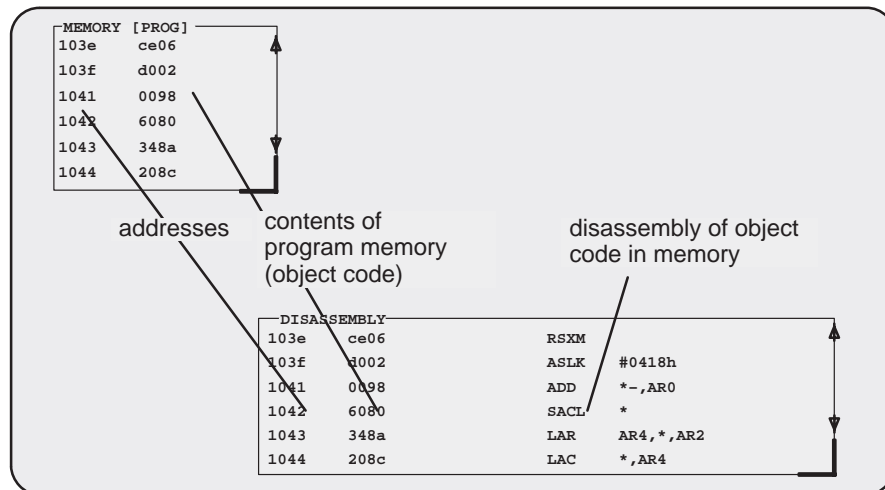
7.1 Displaying Your Source Programs (or Other Text Files)

The debugger displays **assembly language code** in the DISASSEMBLY window. The DISASSEMBLY window is intended primarily for displaying code that the PC points to.

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 window is not large enough to show the entire contents of most assembly language files. You can scroll through the windows. You can also tell the debugger to display specific portions of the disassembly 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.



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.



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*

This command modifies the display so that *address* 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*

In assembly mode, ADDR works like the DASM command, positioning the code starting at *address* as the first line of code in the DISASSEMBLY window.

Modifying assembly language code

You can modify the code in the disassembly window on a statement-by-statement basis. The method for doing this is called *patch assembly*. Patch assembly provides a simple way to temporarily correct minor problems by allowing you to change individual statements and instruction words.

You can patch-assemble code by using a command or by using the mouse.



patch Use the PATCH command to identify the address of the statement you want to change and the new statement you want to use at that address. The format for this command is:

patch *address, assembly language statement*

If you prefer, you can enter just PATCH, or enter PATCH with an *address* only; the debugger will display a dialog box where you can fill in the parameter(s).



Note that for patch assembly, the **right** mouse button is used instead of the left. (Clicking the left mouse button sets a breakpoint.)



1) Point to the statement that you want to modify.



2) Click the right button. The debugger will open a dialog box so that you can enter the new statement. The address field will already be filled in; clicking on the statement defines the address. The statement field will already be filled in with the current statement at that address (this is useful when only minor edits are necessary).

Patch assembly may, at times, cause undesirable side effects:

- ☐ Patching a multiple-word instruction with an instruction of lesser length will leave “garbage” or an unwanted new instruction in the remaining old instruction fragment. This fragment must be patched with either a valid instruction or a NOP, or else unpredictable results may occur when you are running code.
- ☐ Substituting a larger instruction for a smaller one will partially or entirely overwrite the following instruction; you will lose the instruction and may be left with another fragment.

If you want to insert a large amount of new code or if you want to skip over a section of code, you can use a different patch assembly technique:

- ☐ To insert a large section of new code, patch a branch instruction to go to an area of memory not currently in use. Using the patch assembler, add new code to this area of memory, and branch back to the statement following the initial branch.
- ☐ To skip over a portion of code, patch a branch instruction to go beyond that section of code.

Effects of Patch Assembly

The patch assembler changes only the disassembled assembly language code—it does not change your source code. After determining the correct solution to problems in the disassembly, edit your source file, reassemble it, and reload the new object file into the debugger.

Additional information about modifying assembly language code

When using patch assembly to modify code in the disassembly window, keep these things in mind:

- ☐ **Directives.** You cannot use directives (such as `.global` or `.word`).
- ☐ **Expressions.** You can use constants, but you cannot use arithmetic expressions. For example, an expression like `12 + 33` is not valid in patch assembly, but a constant such as `12` is allowed.
- ☐ **Labels.** You cannot define labels. For example, a statement such as the following is not allowed:

```
LOOP: B LOOP
```

However, an instruction can refer to a label as long as it is defined in a COFF file that is already loaded.

- ☐ **Constants.** You can use hexadecimal, octal, decimal, and binary constants. The syntax to input constants is the same as that for the DSP assembler. (Refer to the *TMS320 Fixed-Point DSP Assembly Language Tools User's Guide*.)
- ☐ **Parallel instructions.** You can use parallel instructions. The syntax of these instructions is the same as that for the DSP assembler. (Refer to the *TMS320 Fixed-Point DSP Assembly Language Tools User's Guide*.)
- ☐ **Error messages.** The error messages for the patch assembler are the same as the corresponding DSP assembler error messages. Refer to the *TMS320 Fixed-Point DSP Assembly Language Tools User's Guide* for a detailed list of these messages.

Displaying other text files

The FILE window is for displaying any text file. You may, for example, wish to examine system files such as `autoexec.bat` or the initialization batch file. You can also view your original assembly language source files in the FILE window.



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

file *filename*

This opens the FILE window to display the contents of *filename* and switches you to mixed mode. You can view only one text file at a time. Note that you can also access this command from the Load pulldown menu.

7.2 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 assembling and linking your source files; see Section 3.3, *Preparing Your Program for Debugging*, on page 3-7.)

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 debugger command (evm16) 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 debugger command (evm16), enter 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.

7.3 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 **..\..\code**. The debugger can recognize a cumulative total of 20 paths specified with D_SRC, **-i**, and USE.

7.4 Running Your Programs

To debug your programs, you must execute them on the 'C16 EVM. 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 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 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 7-12).
- ☐ 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 through a specific statement 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.

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.)

The debugger supports several commands for single-stepping through a program. 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 7-12).
- ☐ If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger single-steps *count* assembly language statements.

step Use the STEP command to single-step through assembly language code. The format for this command is:

step [*expression*]

The debugger executes one assembly language statement at a time.

next The NEXT command is similar to the STEP command. It always steps to the next consecutive statement. The format for this commands is:

next [*expression*]





You can also single-step through programs by using function keys:

- F8** Acts as a STEP command.
- F10** 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 emulation is disabled

runf Use the RUNF command to disable emulation while code is executing. The format for this command is:

runf

When you enter RUNF, the debugger clears all breakpoints 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.

halt Use the HALT command to re-enable emulation 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 to the EVM 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 register in a WATCH window, you may want to set breakpoints on statements that affect that register and use that register in the expression.

7.5 Halting Program Execution

Whenever you're running or single-stepping code, program execution halts automatically if the debugger encounters and executes 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.

Chapter 8

Managing Data

The debugger allows you to examine and modify many different types of data related to the 'C16 and to your program. You can display and modify the values of:

- ☐ Individual memory locations or a range of memory
- ☐ 'C16 registers

This chapter tells you how to display and change data.

Topic	Page
8.1 Where Data Is Displayed	8-2
8.2 Basic Commands for Managing Data	8-2
8.3 Basic Methods for Changing Data Values	8-4
Editing data displayed in a window	8-4
Advanced “editing”—using expressions with side effects	8-5
8.4 Managing Data in Memory	8-6
Displaying memory contents	8-6
Displaying program memory	8-7
Saving memory values to a file	8-8
Filling a block of memory	8-9
8.5 Managing Register Data	8-10
Displaying register contents	8-11
8.6 Managing Data in a WATCH Window	8-12
Displaying data in a WATCH window	8-12
Deleting watched values and closing the WATCH window	8-13
8.7 Displaying Data in Alternative Formats	8-14
Changing the default format for specific data types	8-14
Changing the default format with ?, MEM, and WA	8-16

8.1 Where Data Is Displayed

Three 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 'C16 registers
specific memory locations or registers	WATCH window Displays selected data

This group of windows is referred to as **data-display windows**.

8.2 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.



- ? The ? command evaluates an expression and shows the result in the COMMAND window display area. The basic syntax for this command is:

? *expression*[**@prog** | **@data**] [, *display format*]

The *expression* can be any C expression (including an expression with side effects), any 'C16 register, or an address. 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.

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 (*).

When you use the optional *display format* parameter, data will be displayed in one of the formats shown in Table 8–2 on page 8-14.

Here are some examples that use the ? command.

Command	Result displayed in the COMMAND window
? pc	194
? pc,x	0x00c2
? *0x0024	21845
? *0x0024,x	0x5555
? AR0	-21784
? AR0,x	0xaae8

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*[@prog | @data]
or **e** *expression*[@prog | @data]

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.

8.3 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
- ☐ 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 **ESC** 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 **F6**. For more detail, see Section 4.4, *The Active Window*, on page 4-13.)
 - 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 additional data-management methods that take advantage of the fact that C expressions are accepted as parameters by most debugger commands, and that C expressions can have *side effects*. When an expression has a side effect, 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 `AR0`, you can enter:

```
? AR0
```

You can also use this type of command to modify `AR0`'s contents. Here are some examples of how you might do this:

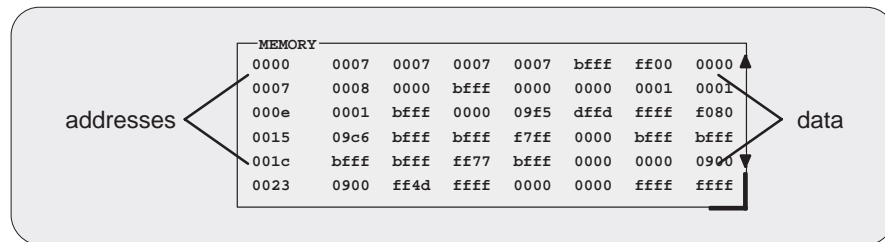
<code>? AR0++</code>	<i>Side effect: increments the contents of AR0 by 1</i>
<code>eval --AR0</code>	<i>Side effect: decrements the contents of AR0 by 1</i>
<code>? AR0 = 8</code>	<i>Side effect: sets AR0 to 8</i>
<code>eval AR0/=2</code>	<i>Side effect: divides contents of AR0 by 2</i>

Note that not all expressions have side effects. For example, if you enter `? AR0+4`, the debugger displays the result of adding 4 to the contents of `AR0` but does not modify `AR0`'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>	

8.4 Managing Data in Memory

The debugger maintains a MEMORY window that displays the contents of memory. For details concerning the MEMORY window, see the *MEMORY window* discussion (page 4-6).



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 8.3, *Basic Methods for Changing Data Values* (page 8-4), 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 4-16, for more information).

The *expression* can be an absolute 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.

- ❑ **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 ST - 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 4-22) 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 (ST - AR0 + label)@prog
? *0x26@prog
wa *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.

Saving memory values to 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. (For more information about COFF, refer to the *TMS320 Fixed-Point DSP Assembly Language Tools User's Guide*.) 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 or data) 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, page, length, data*

- ☐ The *address* parameter identifies the first address in the block.
- ☐ 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 *length* parameter defines the number of words to fill.
- ☐ 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,0,0xf,0xabcd

If you want to check to see that memory has been filled as you have asked, you can enter:

mem 0x10ff@prog

This changes the MEMORY window display to show the block of memory beginning at program memory address 0x10FF.

The FILL command can also be executed from the Memory pulldown menu.

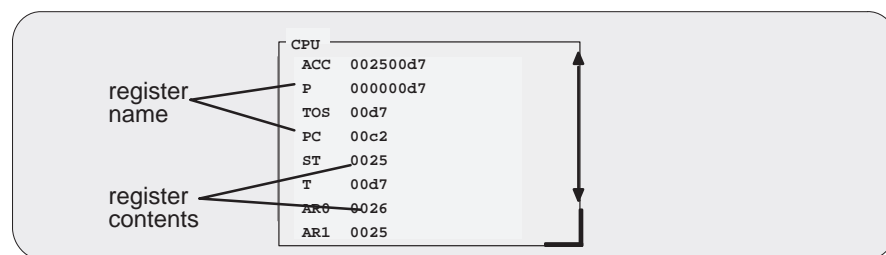
8.5 Managing Register Data

While using the 'C16 debugger, you can display and edit the registers and pseudoregisters listed in Table 8–1 below. For more information about these registers, refer to the *TMS320C1x User's Guide* (literature number SPRU013).

Table 8–1. Registers and Pseudoregisters for Use With the 'C16

Register Acronym	Size (in bits)	Description
PC	16	Program counter
ACC	32	Accumulator
ACCL	16	Accumulator low word
ACCH	16	Accumulator high word
P	32	Product register
PL	16	Product register low word
PH	16	Product register high word
T	16	Temporary register
AR0	16	Auxiliary register 0
AR1	16	Auxiliary register 1
ST	16	Status register
TOS	16	Top of stack

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 4-11).



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 8.3, *Basic Methods for Changing Data Values* (page 8-4), 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 MEMORY 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 PC'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
```

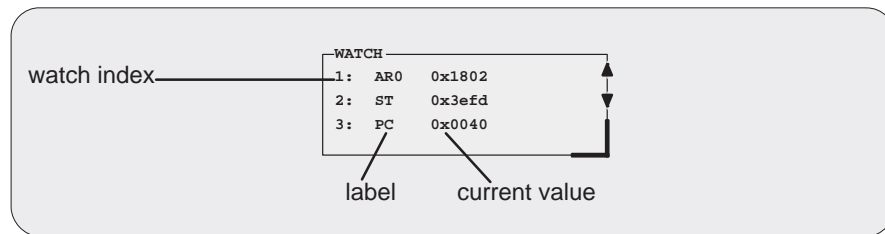
This adds the ST to the WATCH window and labels it as *Status Reg*. The register's contents are continuously updated, just as if you were observing the register in the CPU window.

Note: Loading the P Register Value

Since the 'C16 isn't able to load the P (product) register directly, the debugger must load the P value by using two multiplier operands, one of which is the T (temporary) register. If you specify a P value that can't be expressed in this manner (for example, it can't be factored or is prime), the debugger displays an error message and does not load the new P value.

8.6 Managing Data in a WATCH Window

The debugger allows you to create a WATCH window that shows you how program execution affects specific expressions, symbols, registers, or memory locations.

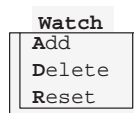


The debugger displays a WATCH window *only when you specifically request a WATCH window* with the WA command (described below). Note that there is only one WATCH window. For additional details concerning the WATCH window, see the *WATCH window* discussion (page 4-12).

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 8.3, *Basic Methods for Changing Data Values* (page 8-4), 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 5.2, *Using the Menu Bar and the Menu Selections* (page 5-7).



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*] [, *display format*]]

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.

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

```
wa *0x26
```

- ❑ 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.
- ❑ When you use the optional *display format* parameter, data will be displayed in one of the formats shown in Table 8-2 on page 8-14.

If you want to use a *display format* parameter without a *label* parameter, just insert an extra comma. For example:

```
wa PC,,x
```

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 8-12 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 on LOAD and SLOAD on WATCH Windows

The debugger automatically closes the WATCH window when you execute a LOAD or SLOAD command.

8.7 Displaying Data in Alternative Formats

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

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

However, any data displayed in the COMMAND, MEMORY, or WATCH window can be displayed in a variety of formats.

Changing the default format for specific data types

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

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

The *display format* parameter identifies the new display format for any data of type *data type*. Table 8–2 lists the available formats and the corresponding characters that can be used as the *display format* parameter.

Table 8–2. Display Formats for Debugger Data

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

Table 8–3 lists the C data types that can be used for the *data type* parameter. Only a subset of the display formats applies to each data type, so Table 8–3 also shows valid combinations of data types and display formats.

Table 8–3. Data Types for Displaying Debugger Data

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

Here are some examples:

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

```
setf short, u
```

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

```
setf short, *
```

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

```
setf
```

You'll see a display that looks something like this:

Display Format Defaults

```
Type char:          ASCII
Type unsigned char: Decimal
Type int:           Decimal
Type unsigned int:  Decimal
Type short:         Decimal
Type unsigned short: Decimal
Type long:          Decimal
Type unsigned long: Decimal
Type float:         Exponential floating point
Type double:        Exponential floating point
Type pointer:       Address
```

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

```
setf *
```

Changing the default format with **?**, **MEM**, and **WA**

You can also use the **?**, **MEM**, and **WA** commands to show data in alternative display formats.

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


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

Here are some examples:

- ☐ To watch the PC in decimal, enter:

```
wa pc,,d 
```

- ☐ To display memory contents in octal, enter:

```
mem 0x0,o 
```

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

Chapter 9

Using Breakpoints

During the debugging process, you may want to halt execution temporarily so that you can examine the contents of selected registers and memory locations before continuing with program execution. You can do this by setting **breakpoints** at critical points in your code. A breakpoint halts program execution *after* executing the breakpointed statement, whether you're running or single-stepping through code.

Breakpoints are especially useful in combination with conditional execution (described on page 7-12).

Note that the commands described in this chapter can also be executed from the Break pulldown menu.

Topic	Page
9.1 Setting a Breakpoint	9-2
9.2 Clearing a Breakpoint	9-3
9.3 Finding the Breakpoints That Are Set	9-4

9.1 Setting a Breakpoint

When you set a breakpoint in your code, the debugger treats the breakpoint as an instruction-execution breakpoint. This means that the 'C16 processor executes the breakpoint and then stops execution. Note that this breakpoint feature differs from that in TI's other versions of the debugger (for example, the 'C3x or 'C5x version of the debugger). These versions use instruction-acquisition breakpoints, where the debugger stops executing when it reaches the breakpointed statement.

When you set a breakpoint, the debugger highlights the breakpointed lines in two ways:

- ☐ It prefixes the statement with the characters BP>.
- ☐ It shows the line in a bolder or brighter font. (You can use screen-customization commands to change this highlighting method.)

Note: 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.
-

There are several ways to set a breakpoint:







-
- 1) Point to the line of assembly language code where you'd like to set a breakpoint.



- 2) Click the left button.

Repeating this action clears the breakpoint.



-
- 1) Make the 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  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, or the name of an assembly language label. You cannot set multiple breakpoints at the same statement.

9.2 Clearing a Breakpoint

There are several ways to clear a breakpoint.



- 1) Point to a breakpointed assembly language statement.
- 2) Click the left button.



- 1) Use the arrow keys or the DASM command to move the cursor to a breakpointed assembly language 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, or the name of an assembly language label. If no breakpoint is set at *address*, the debugger ignores the command.

9.3 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>
0006	
000a	
000b	
0007	meminit
0000	main

The address is the memory address of the breakpoint. If the statement you set a breakpoint at has a label, the debugger also displays the label name.

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
10.1 Changing the Colors of the Debugger Display	10-2
<i>Area names:</i> common display areas	10-3
<i>Area names:</i> window borders	10-4
<i>Area names:</i> COMMAND window	10-4
<i>Area names:</i> DISASSEMBLY window	10-5
<i>Area names:</i> data-display windows	10-6
<i>Area names:</i> menu bar and pulldown menus	10-7
10.2 Changing the Border Styles of the Windows	10-8
10.3 Saving and Using Custom Displays	10-9
Changing the default display for monochrome monitors	10-9
Saving a custom display	10-10
Loading a custom display	10-10
Invoking the debugger with a custom display	10-11
Returning to the default display	10-11
10.4 Changing the Prompt	10-12

10.1 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 10–1 lists the valid values for the *attribute* parameters.

Table 10–1. 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 10–2 lists valid values for the *area name* parameters. This is a long list; the subsections following the table further identify these areas.

Table 10–2. Summary of Area Names for the COLOR and SCOLOR Commands

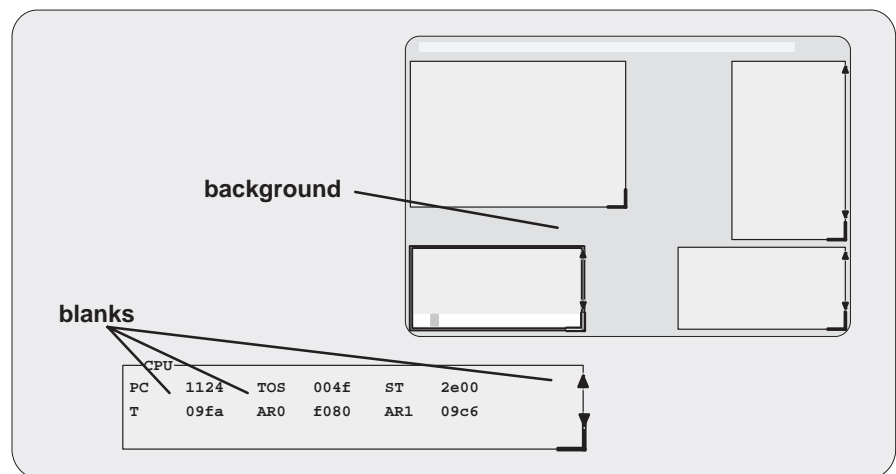
menu_bar	menu_border	menu_entry	menu_cmd
menu_hilite	menu_hicmd	win_border	win_hiborder
win_resize	field_text	field_hilite	field_edit
field_label	field_error	cmd_prompt	cmd_input
cmd_cursor	cmd_echo	asm_data	asm_label
background	blanks	error_msg	file_text
file_brk	file_pc	file_pc_brk	

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 10–2 (left to right, top to bottom).

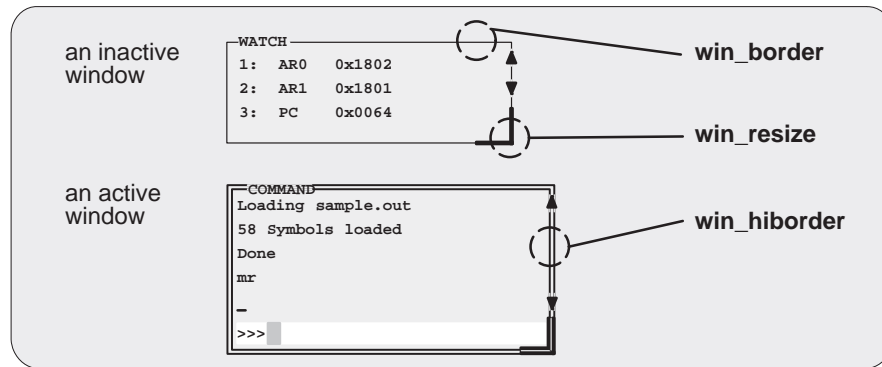
The remainder of this section identifies these areas.

Area names: common display areas



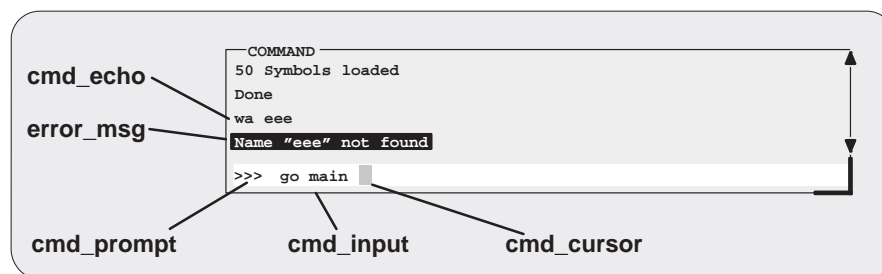
Area identification	Parameter name
Screen background (behind all windows)	background
Window background (inside windows)	blanks

Area names: window borders



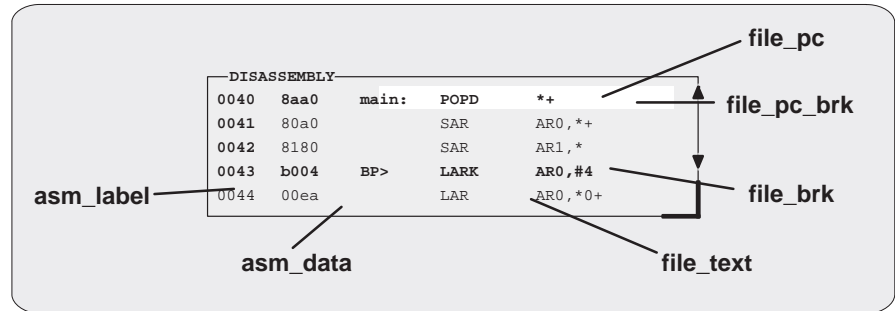
Area identification	Parameter name
Window border for any window that isn't active	win_border
The reversed "L" in the lower right corner of a resizable window	win_resize
Window border of the active window	win_hiborder

Area names: COMMAND window



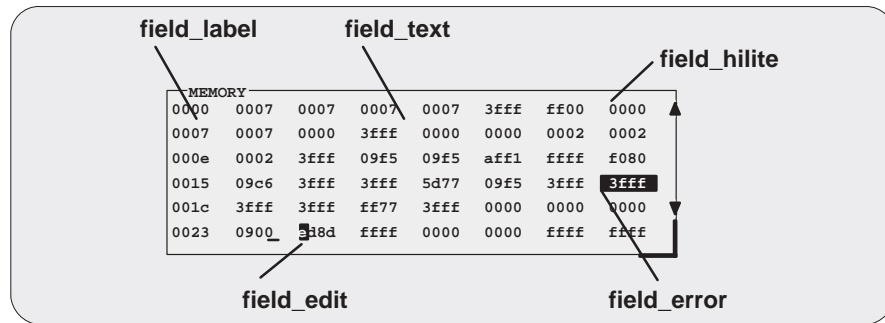
Area identification	Parameter name
Echoed commands in display area	cmd_echo
Errors shown in display area	error_msg
Command-line prompt	cmd_prompt
Text that you enter on the command line	cmd_input
Command-line cursor	cmd_cursor

Area names: *DISASSEMBLY* window

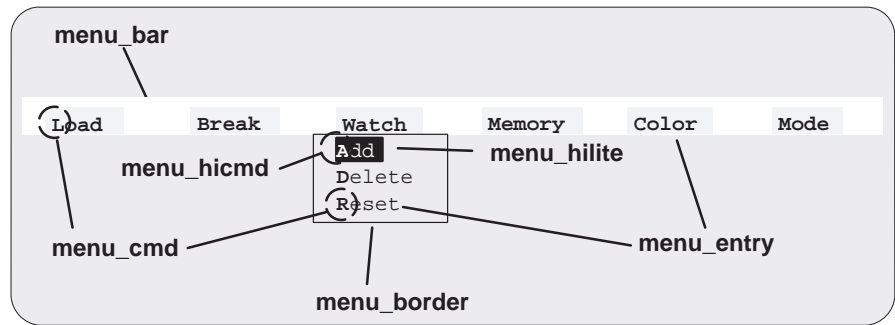


Area identification	Parameter name
Object code in DISASSEMBLY window	asm_data
Addresses in DISASSEMBLY window	asm_label
Text in DISASSEMBLY window	file_text
Breakpointed text in DISASSEMBLY window	file_brk
Current PC in DISASSEMBLY window	file_pc
Breakpoint at current PC in DISASSEMBLY window	file_pc_brk

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)	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	<code>menu_bar</code>
Border of any pulldown menu	<code>menu_border</code>
Text of a menu entry	<code>menu_entry</code>
Invocation key for a menu or menu entry	<code>menu_cmd</code>
Text for current (selected) menu entry	<code>menu_hilite</code>
Invocation key for current (selected) menu entry	<code>menu_hicmd</code>

10.2 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*.

10.3 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 called `init.clr`. 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` defines 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.

The debugger package includes another screen configuration file named `mono.clr`, which 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) 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`.
- ☐ Add a line to the initialization batch file (`init.cmd`) that the debugger executes at invocation time. 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` 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.

10.4 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 (init.cmd.)

You can also execute this command as the Prompt selection on the Color pulldown menu.

Chapter 11

Summary of Commands and Special Keys

This chapter summarizes the debugger's commands and special key sequences.

Topic	Page
11.1 Functional Summary of Debugger Commands	11-2
Changing modes	11-3
Managing windows	11-3
Displaying and changing data	11-3
Performing system tasks	11-4
Displaying files and loading programs	11-5
Managing breakpoints	11-5
Customizing the screen	11-5
Memory mapping	11-6
Running programs	11-6
11.2 How the Menu Selections Correspond to Commands	11-7
Program-execution commands	11-7
File/load commands	11-7
Breakpoint commands	11-7
Watch commands	11-8
Memory commands	11-8
Screen-configuration commands	11-8
Mode commands	11-8
11.3 Alphabetical Summary of Debugger Commands	11-9
11.4 Summary of Special Keys	11-34
Editing text on the command line	11-34
Using the command history	11-34
Halting or escaping from an action	11-35
Displaying the menu selections	11-35
Running code	11-35
Selecting a window	11-36
Moving or sizing a window	11-36
Editing data or selecting the active field	11-36
Scrolling through a window's contents	11-37

11.1 Functional Summary of Debugger Commands

This section summarizes the debugger commands according to these categories:

- ☐ **Changing modes.** These commands enable you to switch between the two debugging modes (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.
- ☐ **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.
- ☐ **Performing system tasks.** These commands enable you to perform several DOS-like functions and provide you with some control over the target system.
- ☐ **Displaying files and loading programs.** These commands enable you to change the displays in the DISASSEMBLY window 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 hardware 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	11-10
Put the debugger in mixed mode	mix	11-21

Managing windows

To do this	Use this command	See page
Make the active window as large as possible	zoom	11-33
Reposition the active window	move	11-21
Resize the active window	size	11-28
Select the active window	win	11-33

Displaying and changing data

To do this	Use this command	See page
Change the default format for displaying data values	setf	11-27
Continuously display the value of a variable, register, or memory location within the WATCH window	wa	11-32
Delete a data item from the WATCH window	wd	11-32
Delete all data items from the WATCH window and close the WATCH window	wr	11-33
Display a different range of memory in the MEMORY window	mem	11-20
Evaluate an expression without displaying the results	eval	11-15
Evaluate and display the result of an expression	?	11-9

Performing system tasks

To do this	Use this command	See page
Associate a beeping sound with the display of error messages	sound	11-29
Change the current working directory from within the debugger environment	cd/chdir	11-12
Clear all displayed information from the COMMAND window display area	cls	11-12
Conditionally execute debugger commands in a batch file	if/else/endif	11-17
Define your own command string	alias	11-10
Delete an alias definition	unalias	11-31
Display a string to the COMMAND window while executing a batch file	echo	11-15
Enter any operating-system command or exit to a system shell	system	11-30
Execute commands from a batch file	take	11-31
Exit the debugger	quit	11-24
List the contents of the current directory or any other directory	dir	11-14
Loop debugger commands in a batch file	loop/endloop	11-18
Name additional directories that can be searched when you load source files	use	11-31
Record the information shown in the COMMAND window display area	dlog	11-14
Reset the target system	reset	11-24

Displaying files and loading programs

To do this	Use this command	See page
Display a text file in the FILE window	file	11-15
Display assembly language code at a specific address	dasm	11-13
Display assembly language code at a specific point	addr	11-9
Load an object file	load	11-17
Load only the object-code portion of an object file	reload	11-24
Load only the symbol-table portion of an object file	sload	11-29
Modify disassembly with the patch assembler	patch	11-23

Managing breakpoints

To do this	Use this command	See page
Add a breakpoint	ba	11-10
Delete a breakpoint	bd	11-11
Display a list of all the breakpoints that are set	bl	11-11
Reset (delete) all breakpoints	br	11-12

Customizing the screen

To do this	Use this command	See page
Change the border style of any window	border	11-11
Change the command-line prompt	prompt	11-24
Change the screen colors and update the screen immediately	scolor	11-25
Change the screen colors, but don't update the screen immediately	color	11-13
Load and use a previously saved custom screen configuration	sconfig	11-26
Save a custom screen configuration	ssave	11-29

Memory mapping

To do this	Use this command	See page
Add an address range to the memory map	ma	11-18
Delete an address range from the memory map	md	11-19
Display a list of the current memory map settings	ml	11-21
Enable or disable memory mapping	map	11-19
Initialize a block of memory	fill	11-16
Reset the memory map (delete all ranges)	mr	11-22
Save a block of memory to a system file	ms	11-22

Running programs

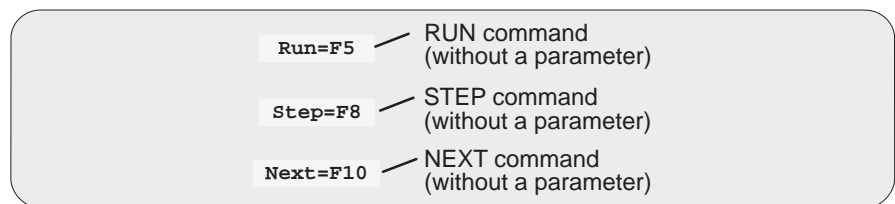
To do this	Use this command	See page
Disable emulation and run free	runf	11-25
Execute commands from a batch file	take	11-31
Re-enable emulation after executing a RUNF command	halt	11-16
Reset the program entry point	restart	11-24
Reset the target system	reset	11-24
Run a program	run	11-25
Run a program up to a certain point	go	11-16
Single-step through assembly language code	step	11-30
Single-step through assembly language code	next	11-23

11.2 How the Menu Selections Correspond to Commands

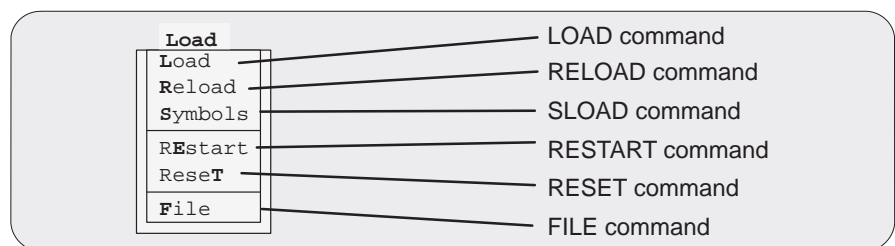
The following sample screens illustrate the relationship of the basic debugger commands to the menu bar and menu selections.

Remember, you can use the menus with or without a mouse. To access a menu from the keyboard, press the **ALT** key and the letter that's highlighted in the menu name. (For example, to display the Load menu, press **ALT L**.) Then, to make a selection from the menu, press the letter that's highlighted in the command you've selected. (For example, on the Load menu, to execute File, press **F**.) If you don't want to execute a command, press **ESC** to close the menu.

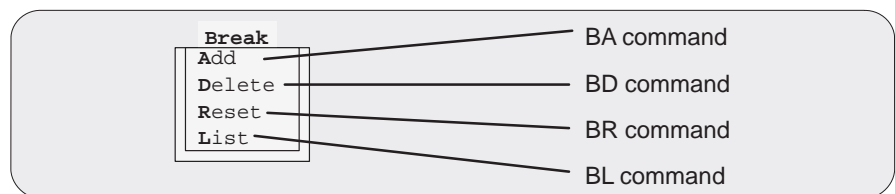
Program-execution commands



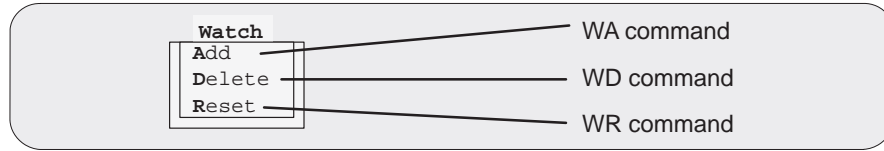
File/load commands



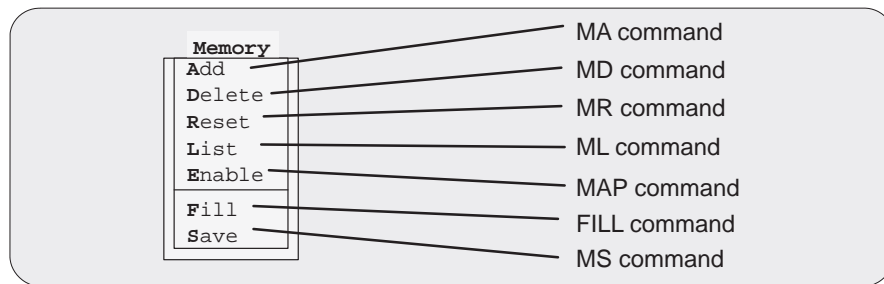
Breakpoint commands



Watch commands



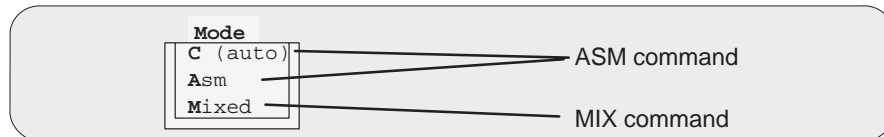
Memory commands



Screen-configuration commands



Mode commands



11.3 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**] [, *display format*]

Menu selection

none

Description

The **?** command evaluates an expression and shows the result in the display area of the COMMAND window.

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.

When you use the optional *display format* parameter, data will be displayed in one of the following formats:

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

addr

Display Code at Selected Address

Syntax

addr *address*[@**prog** | @**data**]

Menu selection

none

Description

Use the **ADDR** command to display disassembly code at a specific point. **ADDR** works like the **DASM** command, positioning the code starting at *address* as the first line of code in the DISASSEMBLY window.

By default, the *address* parameter is treated as a program-memory address.

alias *Define Custom Command String*

Syntax **alias** [*alias name* [, "*command string*"]]

Menu selection none

Description The ALIAS command allows you to associate one or more debugger commands with a single *alias name*. You can include as many debugger commands in the *command string* as you like, as long you separate them with semicolons and enclose the entire string of commands in quotation marks. You can also identify debugger-command parameters by a percent sign followed by a number (%1, %2, etc.). The total number of characters for an individual command (expanded to include parameter values) is limited to 132.

Previously defined alias names can be included as part of the definition for a new alias.

To find the current definition of an alias, enter the ALIAS command with the *alias name* only. To see a list of all defined aliases, enter the ALIAS command with no parameters.

asm *Enter Assembly-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, 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, 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.

cd, chdir *Change Directory*

Syntax	cd [<i>directory name</i>] chdir [<i>directory name</i>]
Menu selection	none
Description	<p>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:</p> <pre>cd c: cd d:\code cd c:\c16dbg</pre>

cls *Clear Screen*

Syntax	cls
Menu selection	none
Description	The CLS command clears all displayed information from the COMMAND window display area.

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_label
background	blanks	error_msg	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).

dasm*Display Selected Disassembly***Syntax****dasm** *address*[@prog | @data]**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	<i>Show Directory Contents</i>
Syntax	dir [<i>directory name</i>]
Menu selection	none
Description	<p>The DIR command displays a directory listing in the display area of the COMMAND window. If you use the optional <i>directory name</i> 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.</p>
dlog	<i>Record Information From COMMAND Window</i>
Syntax	dlog <i>filename</i> [{ a w }] or dlog close
Menu selection	none
Description	<p>The DLOG command allows you to record the information displayed in the command window into a log file.</p> <ul style="list-style-type: none"><input type="checkbox"/> To begin recording the information shown in the display area of the COMMAND window, use: dlog <i>filename</i> <p>Log files can be executed by using the TAKE command. When you use DLOG to record the information from the display area of the COMMAND window into a log file called <i>filename</i>, the debugger automatically precedes all error or progress messages and command results with a semicolon to turn them into comments. This way, you can easily re-execute the commands in your log file by using the TAKE command.</p><input type="checkbox"/> To end the recording session, enter: dlog close  <p>If necessary, you can write over existing log files or append additional information to existing files. The optional parameters of the DLOG command control how existing log files are used:</p> <ul style="list-style-type: none"><input type="checkbox"/> Appending to an existing file. Use the a parameter to open an existing file to which to append the information in the display area.<input type="checkbox"/> Writing over an existing file. Use the w parameter to open an existing file to write over the current contents of the file. Note that this is the default action if you specify an existing filename without using either the a or w options; you will lose the contents of an existing file if you don't use the append (a) option.

echo

Echo String to Command Window

Syntax

echo *string*

Menu selection

none

Description

The ECHO command displays *string* in the COMMAND window display area. This command works only in a batch file, and you don't need quote marks around the *string*. Note that any leading blanks in your command string are removed when the ECHO command is executed.

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 display area of the COMMAND window. 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. You can view only one text file at a time.

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 *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 *length* parameter defines the number of words to fill.
- ☐ The *data* parameter is the value that is placed in each word in the block.

go

Run to Specified Address

Syntax

go [*address*]

Menu selection

none

Description

The GO command executes code through a specific statement 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 re-enables emulation 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 enable emulation and run the debugger in its normal mode of operation.

if/else/endif

Conditionally Execute Debugger Commands

Syntax

```
if Boolean expression
  debugger command
  debugger command
.
.
[else
  debugger command
  debugger command
.
.]
endif
```

Menu selection

none

Description

These commands allow you to conditionally execute debugger commands in a batch file. If the Boolean expression evaluates to true (1), the debugger executes the commands between the IF and the ELSE or ENDIF. Note that the ELSE portion of the command is optional.

The IF/ELSE/ENDIF conditional commands work with the following provisions:

- ☐ You can use IF/ELSE/ENDIF commands only in a batch file.
- ☐ You must enter each debugger command on a separate line in the batch file.
- ☐ You can't nest IF/ELSE/ENDIF commands within the same batch file.

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 window.

loop/endloop *Loop Debugger Commands*

Syntax	loop <i>expression</i> <i>debugger command</i> <i>debugger command</i> . . endloop
Menu selection	none
Description	<p>The LOOP/ENDLOOP commands allow you to set up a looping situation in a batch file. These looping commands evaluate in the same method as in the run conditional command expression:</p> <ul style="list-style-type: none"><input type="checkbox"/> If you use an <i>expression</i> that is not Boolean, the debugger evaluates the expression as a loop count.<input type="checkbox"/> If you use a Boolean <i>expression</i>, the debugger executes the command repeatedly as long as the expression is true. <p>The LOOP/ENDLOOP commands work under the following conditions:</p> <ul style="list-style-type: none"><input type="checkbox"/> You can use LOOP/ENDLOOP commands only in a batch file.<input type="checkbox"/> You must enter each debugger command on a separate line in the batch file.<input type="checkbox"/> You can't nest LOOP/ENDLOOP commands within the same batch file.

ma *Memory Map Add*

Syntax	ma <i>address, page, length, type</i>
Menu selection	Memory→Add
Description	<p>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.</p> <ul style="list-style-type: none"><input type="checkbox"/> The <i>address</i> parameter defines the starting address of a range in data or program memory. This parameter can be an absolute address, any C expression, or an assembly language label.

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

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

- ☐ 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

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.

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 or data 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 or data) that the range occupies:

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

mem*Modify MEMORY Window Display***Syntax**

mem *expression*[@**prog** | @**data**] [, *display format*]

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.

When you use the optional *display format* parameter, memory will be displayed in one of the following formats:

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

mix	<i>Enter Mixed Debugger Mode</i>
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.
ml	<i>Memory Map List</i>
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.
move	<i>Move Active Window</i>
Syntax	move [<i>X position</i> , <i>Y position</i> [, <i>width</i> , <i>length</i>]]
Menu selection	none
Description	<p>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 <i>width</i> and <i>length</i> values). You can use the MOVE command in one of two ways:</p> <ul style="list-style-type: none"><input type="checkbox"/> By supplying a specific <i>X position</i> and <i>Y position</i> or<input type="checkbox"/> By omitting the <i>X position</i> and <i>Y position</i> parameters and using function keys to interactively move the window. <p>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.</p>

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 running under Microsoft Windows



[‡] PC with EGA card

[§] 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 or data) 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.

next

Single-Step, Next Statement

Syntax

next [*expression*]

Menu selection

Next=F10

Description

The NEXT command is similar to the STEP command. The debugger executes one assembly language statement at a time. NEXT always steps to the next consecutive statement.

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 7-12, discusses this in detail).

patch

Patch Assemble

Syntax

patch *address, assembly language instruction*

Menu selection

none

Description

The PATCH command allows you to patch-assemble disassembly statements. The *address* parameter identifies the address of the statement you want to change. The *assembly language instruction* parameter is the new statement you want to use at *address*. If you enter the command without parameters or with only the *address* parameter, the debugger will open a dialog box so that you can enter the remaining parameter(s).

prompt	<i>Change Command-Line Prompt</i>
Syntax	prompt <i>new prompt</i>
Menu selection	Color→P rompt
Description	The PROMPT command changes the command-line prompt. The <i>new prompt</i> can be any string of characters (note that a semicolon or comma ends the string).
quit	<i>Exit Debugger</i>
Syntax	quit
Menu selection	none
Description	The QUIT command exits the debugger and returns to the operating system (or Microsoft Windows).
reload	<i>Reload Object Code</i>
Syntax	reload <i>object filename</i>
Menu selection	Load→R eload
Description	The RELOAD command loads only an object file <i>without</i> loading its associated symbol table. This is useful for reloading a program when target memory has been corrupted.
reset	<i>Reset Target System</i>
Syntax	reset
Menu selection	Load→R eseT
Description	The RESET command resets the EVM and reloads the monitor. Note that this is a <i>software</i> reset.
restart	<i>Reset PC to Program Entry Point</i>
Syntax	restart rest
Menu selection	Load→R E start
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.)

run	<i>Run Code</i>
Syntax	run [<i>expression</i>]
Menu selection	Run=F5
Description	<p>The RUN command is the basic command for running an entire program. The command's behavior depends on the type of parameter you supply:</p> <ul style="list-style-type: none"> <input type="checkbox"/> If you don't supply an <i>expression</i>, the program executes until it encounters and executes a breakpoint or until you press the left mouse button or press ESC. <input type="checkbox"/> If you supply a logical or relational <i>expression</i>, this becomes a conditional run (described in detail on page 7-12). <input type="checkbox"/> If you supply any other type of <i>expression</i>, the debugger treats the expression as a <i>count</i> parameter. The debugger executes <i>count</i> instructions, halts, and updates the display.
runf	<i>Run Free</i>
Syntax	runf
Menu selection	none
Description	<p>The RUNF command disables emulation while code is executing. When you enter RUNF, the debugger clears all breakpoints and causes the processor to begin execution at the current PC. You can quit the debugger, or you can continue to enter commands. However, any command that causes the debugger to access the target at this time produces an error.</p> <p>The HALT command stops a RUNF; note that the debugger automatically executes a HALT when the debugger is invoked.</p>
scolor	<i>Change Screen Colors</i>
Syntax	scolor <i>area name</i> , <i>attribute</i> ₁ [, <i>attribute</i> ₂ [, <i>attribute</i> ₃ [, <i>attribute</i> ₄]]]
Menu selection	Color→Config
Description	<p>The SCOLOR command changes the color of specified areas of the debugger display and updates the display immediately. The <i>area name</i> parameter identifies the areas of the display that are affected. The <i>attributes</i> identify how the area is affected. The first two <i>attribute</i> 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.</p>

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_label
background	blanks	error_msg	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.

setf
Set Default Data-Display Format
Syntax
setf [*data type, display format*]

Menu selection

none

Description

The SETF command changes the display format for a specific data type. If you enter SETF with no parameters, the debugger lists the current display format for each data type.

- ☐ The *data type* parameter can be any of the following C data types:

char	short	uint	ulong	double
uchar	int	long	float	ptr


- ☐ The *display format* parameter can be any of the following characters:

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

Only a subset of the display formats can be used for each data type. Listed below are the valid combinations of data types and display formats.

Data Type	Valid Display Formats										Data Type	Valid Display Formats									
	c	d	o	x	e	f	p	s	u			c	d	o	x	e	f	p	s	u	
char (c)	√	√	√	√					√		long (d)	√	√	√	√						√
uchar (d)	√	√	√	√					√		ulong (d)	√	√	√	√						√
short (d)	√	√	√	√					√		float (e)			√	√	√	√				
int (d)	√	√	√	√					√		double (e)			√	√	√	√				
uint (d)	√	√	√	√					√		ptr (p)			√	√			√	√		

To return all data types to their default display format, enter:

setf * 

size*Size Active 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 running under Microsoft Windows





[‡] PC with EGA card



[§] 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  or .

sload

Load Symbol Table

Syntax

sload *object filename*

Menu selection

Load→**S**ymbols

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 window.

sound

Enable Error Beep

Syntax

sound **on** | **off**

Menu selection

none

Description

You can cause a beep to sound every time a debugger error message is displayed. This is useful if the COMMAND window is hidden (because you wouldn't see the error message). By default, sound is off.

ssave

Save Screen Configuration

Syntax



ssave [*filename*]

Menu selection

Color→**S**ave

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. If you don't supply a *filename*, then the debugger saves the current configuration into a file named init.clr and places the file in the current directory.

step	<i>Single-Step</i>
Syntax	step [<i>expression</i>]
Menu selection	Step=F8
Description	<p>The STEP command single-steps through assembly language code. The debugger executes one statement at a time.</p> <p>The <i>expression</i> parameter specifies the number of statements that you want to single-step. You can also use a conditional <i>expression</i> for conditional single-step execution (<i>Running code conditionally</i>, page 7-12, discusses this in detail).</p>
system	<i>Enter Operating-System Command</i>
Syntax	system [<i>operating-system command</i> [, <i>flag</i>]]
Menu selection	none
Description	<p>The SYSTEM command allows you to enter operating-system commands without explicitly exiting the debugger environment.</p> <p>If you enter SYSTEM with no parameters, the debugger will open a system shell and display the operating-system prompt. At this point, you can enter any operating-system command. Note that available memory may limit the commands that you can enter. When you finish, return to the debugger environment by typing:</p> <p>exit </p> <p>If you prefer, you can supply the operating-system command as a parameter to the SYSTEM command. If the result of the command is a message or other display, the debugger will blank the top of the debugger display to show the information. In this case, you can use the <i>flag</i> parameter to tell the debugger whether or not it should hesitate after displaying the information. <i>Flag</i> may be a 0 or a 1.</p> <p>0 If you supply a value of 0 for <i>flag</i>, the debugger immediately returns to the debugger environment after the last item of information is displayed.</p> <p>1 If you supply a value of 1 for <i>flag</i>, the debugger does not return to the debugger environment until you press . (This is the default.)</p>

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.

unalias

Remove Custom Command String

Syntax

unalias *alias name*
unalias *

Menu selection

none

Description

The UNALIAS command deletes defined aliases.

- ☐ To delete a **single alias**, enter the UNALIAS command with an alias name. For example, to delete an alias named NEWMAP, enter:

```
unalias NEWMAP 
```

- ☐ To delete **all aliases**, enter an asterisk instead of an alias name:

```
unalias * 
```

Note that the * symbol *does not* work as a wildcard.

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*] [, *display format*]]

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.

When you use the optional *display format* parameter, data will be displayed in one of the following formats:

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

If you want to use a *display format* parameter without a *label* parameter, just insert an extra comma. For example:

`wa PC,,d`

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.

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, 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.

zoom

Enlarge Active Window

Syntax

zoom

Menu selection

none

Description










The ZOOM command makes the active window as large as possible. To "unzoom" a window, enter the ZOOM command a second time; this returns the window to its prezoom size and position.

11.4 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
- ☐ Halting or escaping from an action
- ☐ Displaying the menu selections
- ☐ Running code
- ☐ Selecting a window
- ☐ Moving or sizing a window
- ☐ Editing data or selecting the active field
- ☐ Scrolling through a window's contents

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
Repeat the last command that you entered	
Move backward, one command at a time, through the command history	
Move forward, one command at a time, through the command history	 

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
<input type="checkbox"/> Halt program execution	ESC
<input type="checkbox"/> Close a pulldown menu	
<input type="checkbox"/> Undo an edit of the active field in a data-display window (pressing this key leaves the field unchanged)	
<input type="checkbox"/> Halt the display of a long list of data in the COMMAND window display area	

Displaying menu selections

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)	F5
Single-step code from the current PC (equivalent to the STEP command without an <i>expression</i> parameter)	F8
Single-step code from the current PC (equivalent to the NEXT command without an <i>expression</i> parameter)	F10

Selecting 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)	F6

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
<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	









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
<input type="checkbox"/> Set or clear a breakpoint	F9
<input type="checkbox"/> <i>Any data-display window:</i> Edit the contents of the current field	

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	
<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	

Chapter 12

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
12.1 C Expressions for Assembly Language Programmers	12-2
12.2 Restrictions and Features Associated With Expression Analysis in the Debugger	12-4
Restrictions	12-4
Additional features	12-4

12.1 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)		

☐ Increment and decrement operators

++ increment -- decrement

These unary operators can precede or follow a symbol. When the operator precedes a symbol, the symbol value is incremented/decremented before it is used in the expression; when the operator follows a symbol, the symbol value is incremented/decremented after it is used in the expression. Because these operators affect the symbol's final value, they have side effects.

☐ Bitwise operators

&	bitwise AND		bitwise OR
^	bitwise exclusive-OR	<<	left shift
>>	right shift	~	1s complement (unary)

☐ Assignment operators

=	assignment	+=	assignment with addition
-=	assignment with subtraction	/=	assignment with division
%=	assignment with modulo	&=	assignment with bitwise AND
^=	assignment with bitwise XOR	=	assignment with bitwise OR
<<=	assignment with left shift	>>=	assignment with right shift
*=	assignment with multiplication		

These operators support a shorthand version of the familiar binary expressions; for example, $X = X + Y$ can be written in C as $X += Y$. Because these operators affect a symbol's final value, they have side effects.

12.2 Restrictions and Features Associated With Expression Analysis in the Debugger

The debugger's expression analysis is based on C expression analysis and includes all mathematical, relational, pointer, and assignment operators. However, the debugger's expression analysis has 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).
- ☐ Only these forms of type casts 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 TMS320C16's auxiliary registers are treated as integers and/or pointers.
- ☐ Void expressions are legal (treated like integers).
- ☐ 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
 - *AR0
 - *(AR1 + 123)

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 command 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.

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.

Appendix A

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) Looks for the `init.clr` screen configuration file.
(The debugger searches for the screen configuration file in directories named with `D_DIR`.)
- 5) Initializes the debugger screen and windows but initially displays only the `COMMAND` window.
- 6) Finds the batch file that defines your memory map by searching in directories named with `D_DIR`. The debugger expects this file to set up the memory map and follows these steps to look for the batch file:
 - a) When you invoke the debugger, it checks to see if you've used the `-t` debugger option. If it finds the `-t` option, the debugger reads and executes the specified file.
 - b) If you don't use the `-t` option, the debugger looks for the default initialization batch file. For the EVM, this file is named *evminit.cmd*.
If the debugger finds the file, it reads and executes the file.
 - c) If the debugger does not find the `-t` option or the initialization batch file, it looks for a file called *init.cmd*. This allows you to have one initialization batch file for more than one debugger tool.
- 7) Loads any object filenames specified with `D_OPTIONS` or specified on the command line during invocation.
- 8) Determines the initial mode (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.

Appendix B

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.

Topic	Page
B.1 Associating Sound With Error Messages	B-2
B.2 Alphabetical Reference of Debugger Messages	B-2
B.3 Additional Instructions for Expression Errors	B-14
B.4 Additional Instructions for Hardware Errors	B-14

B.1 Associating Sound With Error Messages

You can associate a beeping sound with the display of error messages. To do this, use the SOUND command. The format for this command is:

sound on | off

By default, no beep is associated with error messages (SOUND OFF). The beep is helpful if the COMMAND window is hidden behind other windows.

B.2 Alphabetical Summary of Debugger Messages

Symbols

']' expected

<i>Description</i>	This is an expression error—it means that the parameter contained an opening [symbol but didn't contain a closing] symbol.
<i>Action</i>	See Section B.3 (page B-14).

(') expected

<i>Description</i>	This is an expression error—it means that the parameter contained an opening (symbol but didn't contain a closing) symbol.
<i>Action</i>	See Section B.3 (page B-14).

A

Aborted by user

<i>Description</i>	The debugger halted a long COMMAND display listing (from DIR, ML, or BL) because you pressed the ⏏ key.
<i>Action</i>	None required; this is normal debugger behavior.

B**Breakpoint already exists at address**

<i>Description</i>	During single-step execution, the debugger attempted to set a breakpoint where one already existed. (This isn't necessarily a breakpoint that you set—it may have been an internal breakpoint that was used for single-stepping).
<i>Action</i>	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 breakpoints are set in your program. Use the BR command to delete all breakpoints, or use the BD command to delete individual 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.

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:\c16dbg</code> , not just <code>c16dbg</code>).

Cannot edit field

<i>Description</i>	Expressions that are displayed in the WATCH window cannot be edited.
<i>Action</i>	If you attempted to edit an expression in the WATCH window, you may have actually wanted to change the value of a register used in the expression. Use the ? or EVAL command to edit the actual register. The expression value will automatically be updated.

Cannot find/open initialization file

<i>Description</i>	The debugger can't find the init.cmd file.
<i>Action</i>	Be sure that init.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 <i>Setting Up the Debugger Environment</i> , page 1-8.

Cannot halt the processor

<i>Description</i>	This is a fatal error—for some reason, pressing ESC didn't halt program execution.
<i>Action</i>	Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger again.

Cannot open config file

<i>Description</i>	The SCONFIG command can't find the screen-customization file that you specified.
<i>Action</i>	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"

<i>Description</i>	The debugger attempted to show <i>filename</i> in the FILE window but could not find the file.
<i>Action</i>	Be sure that the file exists as named. If it does, enter the USE command to identify the file's directory.

Cannot open object file: “filename”

<i>Description</i>	The file specified with the LOAD, SLOAD, or RELOAD command is not an object file that the debugger can load.
<i>Action</i>	Be sure that you're loading an actual object file. Be sure that the file was linked.

Cannot open new window

<i>Description</i>	A maximum of 127 windows can be open at once. The last request to open a window would have made 128, which isn't possible.
<i>Action</i>	Close any unnecessary WATCH windows by entering WD.

Cannot read processor status

<i>Description</i>	This is a fatal error—for some reason, pressing [ESC] didn't halt program execution.
<i>Action</i>	Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger again.

Cannot reset the processor

<i>Description</i>	This is a fatal error—for some reason, pressing [ESC] didn't halt program execution.
<i>Action</i>	Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger again.

Cannot restart processor

<i>Description</i>	If a program doesn't have an entry point, then RESTART won't reset the PC to the program entry point.
<i>Action</i>	Don't use RESTART if your program doesn't have an explicit entry point.

Cannot set/verify breakpoint at address

<i>Description</i>	Either you attempted to set a breakpoint in read-only or protected memory, or there are hardware problems with the target system.
<i>Action</i>	Check your memory map. If the address that you wanted to breakpoint wasn't in ROM, see Section B.4 (page B-14).

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.3 (page B-14).

Command “cmd” not found

Description The debugger didn’t recognize the command that you typed.

Action Re-enter the correct command. Refer to Chapter 11 or the Quick Reference Card for a list of valid debugger commands.

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 the 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.3 (page B-14).

F

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”

<i>Description</i>	The filename specified for the LOAD, RELOAD, SLOAD, or TAKE command was not found in the current directory or any of the directories identified with D_SRC.
<i>Action</i>	Be sure that the filename was typed correctly. If it 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.

Float not allowed

<i>Description</i>	This is an expression error—a floating-point value was used invalidly.
<i>Action</i>	See Section B.3 (page B-14).

I

Illegal cast

<i>Description</i>	This is an expression error—the expression parameter uses a cast that doesn't meet the C language rules for casts.
<i>Action</i>	See Section B.3 (page B-14).

Illegal left hand side of assignment

<i>Description</i>	This is an expression error—the left-hand side of an assignment expression doesn't meet C language assignment rules.
<i>Action</i>	See Section B.3 (page B-14).

Illegal memory access

<i>Description</i>	There was an attempt to access to unconfigured/reserved/nonexistent memory.
<i>Action</i>	Check your memory map to see if you're trying to access non-existent, unconfigured, or reserved memory.

Illegal operand of &

<i>Description</i>	This is an expression error—the expression attempts to take the address of an item that doesn't have an address.
<i>Action</i>	See Section B.3 (page B-14).

Illegal pointer math

Description This is an expression error—some types of pointer math are not valid in C expressions.

Action See Section B.3 (page B-14).

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.3 (page B-14).

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.3 (page B-14).

Integer not allowed

Description This is an expression error—the command did not accept an integer as a parameter.

Action See Section B.3 (page B-14).

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 11.

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 10–2 (page 10-3).

Invalid color name

<i>Description</i>	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.
<i>Action</i>	Re-enter the COLOR or SCOLOR command with a valid color parameter. Valid color attributes are listed in Table 10–1 (page 10-2).

Invalid memory attribute

<i>Description</i>	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.								
<i>Action</i>	Re-enter the MA command. Use one of the following valid parameters to identify the memory type: <table data-bbox="727 913 1328 1058"> <tr> <td>R, ROM, READONLY</td><td>(read-only memory)</td></tr> <tr> <td>W, WOM, WRITEONLY</td><td>(write-only memory)</td></tr> <tr> <td>RW, RAM</td><td>(read/write memory)</td></tr> <tr> <td>PROTECT</td><td>(no-access memory)</td></tr> </table>	R, ROM, READONLY	(read-only memory)	W, WOM, WRITEONLY	(write-only memory)	RW, RAM	(read/write memory)	PROTECT	(no-access memory)
R, ROM, READONLY	(read-only memory)								
W, WOM, WRITEONLY	(write-only memory)								
RW, RAM	(read/write memory)								
PROTECT	(no-access memory)								


Invalid object file

<i>Description</i>	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.
<i>Action</i>	Be sure that you're loading an actual object file. Be sure that the file was linked. If the file you attempted to load was a valid executable object file, then it was probably corrupted.


Invalid watch delete

<i>Description</i>	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.
<i>Action</i>	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 4-2 (page 4-20) 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 4-1 (page 4-17) 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)

<i>Description</i>	Either the target cable is disconnected, or the target system is faulty.
<i>Action</i>	Check the target cable connections. If the target seems to be connected correctly, see Section B.4 (page B-14).

Lost processor clock

<i>Description</i>	Either the target cable is disconnected, or the target system is faulty.
<i>Action</i>	Check the target cable connections. If the target seems to be connected correctly, see Section B.4 (page B-14).

Lval required

<i>Description</i>	This is an expression error—an assignment expression was entered that requires a legal left-hand side.
<i>Action</i>	See Section B.3 (page B-14).

M

Memory access error at *address*

<i>Description</i>	Either the processor is receiving a bus fault, or there are problems with target system memory.
<i>Action</i>	See Section B.4 (page B-14).

Memory map table full

<i>Description</i>	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).
<i>Action</i>	Stop adding blocks to the memory map. Consolidate any adjacent blocks that have the same memory attributes.

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.

P

Pointer not allowed

Description This is an expression error.

Action See Section B.3 (page B-14).

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

Register access error

Description Either the processor is receiving a bus fault, or there are problems with target-system memory.

Action See Section B.4 (page B-14).

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.

T**Take file stack too deep**

<i>Description</i>	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.
<i>Action</i>	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

<i>Description</i>	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.
<i>Action</i>	Enter a BL command to see where breakpoints are set in your program. Use the BR command to delete all breakpoints or use the BD command to delete individual breakpoints.

Too many paths

<i>Description</i>	More than 20 paths have been specified cumulatively with the USE command, D_SRC environment variable, and -i debugger option.
<i>Action</i>	If you are entering the USE command before entering another command that has a <i>filename</i> parameter, don't enter the USE command. Instead, enter the second command and specify full path information for the <i>filename</i> .

U**User halt**

<i>Description</i>	The debugger halted program execution because you pressed the ESC key.
<i>Action</i>	None required; this is normal debugger behavior.

W

Window not found

Description The parameter supplied for the WIN command is not a valid window name.

Action 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:

CPU	MEMORY	COMMAND
DISASSEMBLY	FILE	WATCH

B.3 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.4 Additional Instructions for Hardware Errors

If you continue to receive the messages that send you to this section, this indicates persistent hardware problems.

Appendix C

Glossary

A

active window: The window that is currently selected for moving, sizing, editing, closing, or some other function.

aliasing: A method of customizing debugger commands; aliasing provides a shorthand method for entering often-used command strings.

assembly mode: A debugging mode that shows assembly language code in the DISASSEMBLY window.

autoexec.bat: A batch file that contains DOS commands for initializing your PC.

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

C: A high-level, general-purpose programming language useful for writing compilers and operating systems and for programming microprocessors.

casting: A feature of C expressions that allows you to use one type of data as if it were a different type of data.

click: To press and release a mouse button without moving the mouse.

COFF: *Common Object File Format.* An implementation of the object file format of the same name developed by AT&T. The 'C16 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 that provides a display area where you enter commands and where the debugger echoes command entry, shows command output, and lists progress or error messages.

CPU window: A window that displays the contents of 'C16 on-chip registers, including the program counter and status register.

current-field cursor: An 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, 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 'C16 programs running on a 'C16 EVM.

disassembly: A reverse assembly of the contents of memory to form assembly language code.

DISASSEMBLY window: A window that displays the disassembly of memory contents.

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.

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.

EVM: *Evaluation Module*. A development tool that lets you execute and debug applications programs by using the 'C16 debugger.

F

FILE window: A window that displays the contents of a specified text file.

I

initdb.bat: As part of normal debugger use, you must enter DOS commands to set up the debugger environment. The most convenient method for doing this is to edit your PC's autoexec.bat or to create a separate initdb.bat file that is used only for this purpose.

I/O switches: Hardware switches on the EVM board that identify the PC I/O memory space used for EVM-debugger communications.

ISA: *Industry Standard Architecture*. A subset of the EISA standard.

M

memory map: A set of special commands 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 a text file in the FILE window.

mouse cursor: Block-shaped cursor that tracks mouse movements over the entire display.

P

PC: Either of two meanings, 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 EVM. The port address is selected via switches on the EVM 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

scroll: To move 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.

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.

symbol table: A file that contains the names of all variables and functions in your 'C16 program.

system shell: With the SYSTEM command, the debugger may blank the debugger display and temporarily exit to the operating-system prompt. This allows you to enter operating-system commands *or* allows the debugger to display information resulting from an operating-system command.

T

TMS320C16: A 114-ns, fixed-point, CMOS digital signal processor, capable of executing 8.77 million instructions per second. It has 256 words of on-chip data RAM, 8K words of on-chip ROM, and 64K of external program space.

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.

Index

? command 7-8, 8-2 to 8-3, 8-11, 11-9
 display formats 8-2, 8-16, 11-9
 modifying PC 7-8
 side effects 8-5
\$\$EVM\$\$ 5-14

A

absolute addresses 8-6, 9-3
ACC register 8-10
ACCH register 8-10
ACCL register 8-10
active window 2-5 to 2-6, 4-10 to 4-12
 breakpoints 9-2
 current field 2-6, 4-10
 customizing its appearance 10-4
 default appearance 4-11
 effects on command entry 5-3
 identifying 2-6, 4-11
 moving 2-9, 4-16 to 4-18
 selecting 4-11 to 4-12, 11-33
 function key method 2-6, 4-12, 11-36
 mouse method 2-6, 4-11
 WIN command 2-6, 4-12, 11-33
 sizing 2-7, 4-13 to 4-15
 zooming 2-8, 4-15 to 4-16, 11-33
ADDR command 4-5, 7-3, 7-8, 11-9
addresses
 absolute addresses 8-6, 9-3
 accessible locations 6-2
 contents of (indirection), 8-13
 data-memory notation 2-5, 4-7
 hexadecimal notation 8-6
 I/O address space 1-4, 1-5
 invalid memory 6-3
 MEMORY window 2-5, 8-6
 nonexistent memory locations 6-2
 program-memory notation 2-5, 4-7
 addresses (continued)
 protected areas 6-3, 6-5
 undefined areas 6-3, 6-5
ALIAS command 2-21, 5-15 to 5-17, 11-10
 See also aliasing
 supplying parameters 5-16
aliasing 5-15 to 5-17
 deleting aliases 5-17
 finding alias definitions 5-16
 limitations 5-17
 listing aliases 5-16
 redefining an alias 5-17
AR0 register 8-10
AR1 register 8-10
area names (for customizing the display)
 COMMAND window 10-4
 common display areas 10-3
 data-display windows 10-6
 DISASSEMBLY window 10-5
 menus 10-7
 summary of valid names 10-3
 window borders 10-4
arithmetic operators 12-2
arrow keys
 COMMAND window 5-3
 editing 8-4
 moving a window 2-9, 4-18, 11-36
 moving to adjacent menus 5-9
 scrolling 2-10, 4-20, 11-37
 sizing a window 2-7, 4-15, 11-36
ASM command 2-13, 11-10
 menu selection 11-8
assembler 1-3, 3-6, 3-7
assembly language code
 displaying 4-2, 7-2
 modifying 7-3 to 7-4
assembly mode 2-12, 4-2
 ASM command 2-13, 11-10

assignment operators 8-5, 12-3
 attributes 10-2
 autoexec.bat file 1-8 to 1-10
 invoking 1-9
 sample 1-8
 auxiliary registers 8-10

B

-b debugger option 1-12
 effect on window positions 4-17
 effect on window sizes 4-14
 with D_OPTIONS environment variable 1-10
 BA command 9-3, 11-10
 menu selection 11-7
 background 10-3
 batch files 5-12
 autoexec.bat 1-8 to 1-10
 controlling command execution 5-13 to 5-15
 conditional commands 5-13 to 5-14, 11-4, 11-17
 looping commands 5-14 to 5-15, 11-4, 11-18
 displaying 7-5
 displaying text when executing 5-13, 11-4, 11-15
 echoing messages 5-13, 11-4, 11-15
 evminit.cmd 6-2, A-1
 execution 11-31
 halting execution 5-12
 init.clr 1-3, 10-9, A-1
 init.cmd 1-3, 6-2, 6-3, A-1
 initdb.bat 1-8 to 1-11
 initialization 6-2, A-1
 evminit.cmd 6-2, A-1
 init.cmd 1-3, 6-2, 6-3, A-1
 invoking
 autoexec.bat 1-9
 initdb.bat 1-9
 mem.map 6-7
 memory maps 6-7
 mono.clr 1-3, 10-9
 TAKE command 5-12, 6-7, 11-31
 BD command 9-3, 11-11
 menu selection 11-7
 bitwise operators 12-3
 BL command 9-4, 11-11
 menu selection 11-7
 blanks 10-3

BORDER command 10-8, 11-11
 menu selection 11-8
 borders
 colors 10-4
 styles 10-8
 BR command 9-3, 11-12
 menu selection 11-7
 breakpoints 9-1 to 9-4
 active window 2-6, 4-12
 adding 11-10
 command method 9-3
 function key method 9-2, 11-36
 mouse method 9-2
 clearing 2-15, 9-3, 11-11, 11-12
 command method 9-3
 function key method 9-3, 11-36
 mouse method 9-3
 commands 11-5
 BA command 9-3, 11-10
 BD command 9-3, 11-11
 BL command 9-4, 11-11
 BR command 9-3, 11-12
 menu selections 11-7
 listing set breakpoints 9-4, 11-11
 restrictions 9-2
 setting 2-14 to 2-15, 9-2
 command method 9-3
 function key method 9-2, 11-36
 mouse method 9-2

C

C expressions 8-5, 12-1 to 12-5
 c16dbg directory 1-7, 1-9
 casting 12-4
 CHDIR (CD) command 2-18, 5-19, 7-7, 11-12
 clearing the display area 2-18, 5-5, 11-12
 "click and type" editing 2-19, 4-21, 8-4
 closing
 a window 4-21
 debugger 1-13, 2-21, 11-24
 log files 5-6, 11-14
 WATCH window 4-21, 8-13, 11-33
 CLS command 2-18, 5-5, 11-12
 code, developing for the 'C16, 3-5
 code-execution (run) commands. *See* run commands
 COLOR command 10-2, 11-13

- colors 10-2
 - area names 10-3 to 10-7
 - comma operator 12-4
 - command history 5-5
 - function key summary 11-34
 - command line 4-4, 5-2
 - changing the prompt 10-12, 11-24
 - cursor 4-10
 - customizing its appearance* 10-4, 10-12
 - editing 5-3
 - function key summary* 11-34
 - COMMAND window 4-3, 4-4, 5-2
 - colors 10-4
 - command line 2-4, 4-4, 5-2
 - editing keys* 11-34
 - customizing 10-4
 - display area 2-4, 4-4, 5-2
 - clearing* 11-12
 - recording information from the display area 5-6, 11-4, 11-14
 - commands
 - alphabetical summary 11-9 to 11-33
 - batch files 5-12
 - controlling command execution*
 - conditional commands 5-13 to 5-14, 11-4, 11-17
 - looping commands 5-14 to 5-15, 11-4, 11-18
 - breakpoint commands 9-1 to 9-4, 11-5
 - See also breakpoints, commands*
 - code-execution (run) commands 7-8, 11-6
 - See also run commands*
 - command line 5-2 to 5-5
 - command strings 5-15 to 5-17
 - conditional commands 5-13 to 5-14, 11-17
 - customizing 5-15 to 5-17
 - data-management commands 8-2 to 8-16, 11-3
 - See also data-management commands*
 - entering and using 5-1 to 5-20
 - file-display commands 7-2, 11-5
 - See also file/load commands*
 - load commands 7-6, 11-5
 - See also load/file commands*
 - looping commands 5-14 to 5-15, 11-18
 - memory commands 6-4 to 6-8
 - See also memory, commands*
 - memory-map commands 11-6
 - See also memory, mapping, commands*
 - menu selections 5-7
 - mode commands 11-3
 - See also modes, commands*
 - commands (continued)
 - run commands. *See* run commands
 - screen-customization commands 10-1 to 10-12, 11-5
 - See also screen-customization commands*
 - system commands 5-18 to 5-20, 11-4
 - See also system commands*
 - window commands 11-3
 - See also windows, commands*
 - conditional commands 5-13 to 5-14, 11-17
 - constants, while editing disassembly 7-5
 - CPU window 2-11, 4-3, 4-8, 8-2, 8-10 to 8-11
 - colors 10-6
 - customizing 10-6
 - editing registers 8-4
 - loading the P register value 8-11
 - current directory, changing 5-19, 7-7, 11-12
 - current field
 - cursor 4-10
 - dialog box 5-4
 - editing 8-4 to 8-16
 - current PC 2-4, 4-5
 - finding 7-8
 - selecting 7-8
 - cursors 4-10
 - command-line cursor 4-10
 - current-field cursor 4-10
 - mouse cursor 4-10
 - customizing the display 10-1 to 10-12
 - changing the prompt 10-12
 - colors 10-2 to 10-7
 - init.clr file 1-3
 - loading a custom display 10-10, 11-26
 - mono.clr file 1-3
 - saving a custom display 10-10, 11-29
 - window border styles 10-8
- D**
- D_DIR environment variable 1-9, 5-12, 10-10, 11-26
 - effects on debugger invocation A-1
 - D_OPTIONS environment variable 1-10, 1-12
 - effects on debugger invocation A-1
 - D_SRC environment variable 1-10, 7-7
 - effects on debugger invocation A-1
 - DASM command 4-5, 7-3, 7-8, 11-13
 - data formats 8-14
 - data types 8-15

- data memory 4-7
 - adding to memory map 6-4, 11-18
 - deleting from memory map 6-6, 11-19
 - filling 8-9, 11-16
 - saving 8-8, 11-22
- data types 8-15
 - See also* display formats
- data-display windows 4-3, 8-2
 - colors 10-6
 - CPU window 2-11, 4-3, 4-8, 8-2, 8-10 to 8-11
 - MEMORY window 2-5, 4-3, 4-6 to 4-7, 8-2, 8-6
 - WATCH window 2-16, 4-3, 4-9, 8-2, 8-12
- data-management commands 2-17, 4-8, 8-2, 11-3
 - ? command 7-8, 8-2 to 8-3, 8-11, 11-9
 - data-format control 8-14 to 8-16
 - EVAL command 7-8, 8-3, 11-15
 - FILL command 8-9, 11-16
 - MEM command 2-5, 4-6, 8-6 to 8-7, 11-20
 - MS command 8-8, 11-22
 - SETF command 8-14 to 8-16, 11-27
 - side effects 8-5
 - WA command 2-16, 4-9, 8-11, 8-12 to 8-13, 11-32
 - WD command 2-17, 4-9, 8-13, 11-32
 - WR command 2-18, 4-9, 8-13, 11-33
- debugger
 - description 3-2
 - display 2-4, 3-2
 - environment setup 1-8 to 1-11
 - EVM version 1-1 to 1-13
 - environment setup* 1-8 to 1-11
 - invocation* 1-12
 - installation 1-7
 - invocation 1-12 to 1-13, 2-3
 - task ordering* A-1
 - key features 3-3
 - messages B-1 to B-14
 - using with Microsoft Windows 1-7, 1-11
- debugging modes 2-12 to 2-13
 - assembly mode 2-12, 4-2
 - commands 11-3
 - ASM command* 2-13, 11-10
 - MIX command* 2-13, 11-21
 - menu selections 2-12
 - mixed mode 2-12, 4-2
 - selection
 - command method* 2-13
 - mouse method* 2-12
- decrement operator 12-3
- default
 - data formats 8-14
 - display 2-4, 10-11
 - I/O address space 1-4
 - memory map 1-3, 2-20, 6-3
 - screen configuration files 1-3, 10-9
 - monochrome displays* 1-3, 10-9
 - switch settings 1-4, 1-5
- dialog boxes 5-8, 5-10
 - effect on entering other commands 5-4
- DIR command 2-18, 5-19, 11-14
- directives, while editing disassembly 7-5
- directories
 - c16dbg directory 1-7, 1-9
 - changing current directory 5-19, 11-12
 - for auxiliary files 1-9
 - for debugger software 1-7, 1-9
 - identifying additional source directories 1-10, 11-31
 - USE command* 11-31
 - identifying current directory 7-7
 - listing contents of current directory 5-19, 11-14
 - relative pathnames 5-19, 11-12
 - search algorithm 5-12, 7-7, A-1
- DISASSEMBLY window 2-5, 4-3, 4-5, 7-2
 - colors 10-5
 - customizing 10-5
 - modifying display 11-13
- display area 4-4, 5-2
 - clearing 2-18, 5-5, 11-12
 - recording information from 5-6, 11-4, 11-14
- display formats 8-14 to 8-16
 - ? command 8-2, 8-16, 11-9
 - data types 8-15
 - MEM command 8-16, 11-20
 - resetting types 8-15
 - SETF command 8-14 to 8-16, 11-27
 - WA command 8-13, 8-16, 11-32
- display requirements 1-2
- displaying
 - assembly language code 7-2
 - batch files 7-5
 - data in nondefault formats 8-14 to 8-16
 - text files 7-5
 - text when executing a batch file 5-13, 11-4, 11-15

DLOG command 5-6, 11-4, 11-14
 ending recording session 5-6
 starting recording session 5-6
 DOS, setting up debugger environment 1-8

E

E command 11-15
 ECHO command 5-13, 11-4, 11-15
 “edit” key (F9), 4-21, 8-4 to 8-5, 11-36
 editing
 “click and type” method 2-19, 4-21, 8-4
 command line 5-3, 11-34
 data values 8-4, 11-36
 dialog boxes 5-10
 disassembly 7-3 to 7-4, 11-23 to 11-33
 side effects 7-4 to 7-5
 expression side effects 8-5
 function key method 2-19, 8-4 to 8-5, 11-36
 MEMORY, CPU, DISASSEMBLY, WATCH 4-21
 mouse method 8-4
 overwrite method 8-4
 window contents 4-21
 ELSE command 5-13 to 5-14, 11-4, 11-17
 end key, scrolling 4-20, 11-37
 ENDIF command 5-13 to 5-14, 11-4, 11-17
 ENDLOOP command 5-14 to 5-15, 11-4, 11-18
 entering commands
 from menu selections 5-7 to 5-11
 on the command line 5-2 to 5-5
 entry point 7-8
 environment variables
 D_DIR 1-9, 5-12, 10-10, 11-26
 effects on debugger invocation A-1
 D_OPTIONS 1-10, 1-12
 effects on debugger invocation A-1
 D_SRC 1-10, 7-7
 effects on debugger invocation A-1
 for debugger options 1-10, 1-12
 identifying auxiliary directories 1-9
 identifying source directories 1-10
 error messages B-1 to B-14
 beeping 11-29, B-2
 EVAL command 7-8, 8-3, 11-15
 modifying PC 7-8
 side effects 8-5

EVM 1-4 to 1-6
 custom switch settings 1-5
 debugger environment 1-8
 debugger installation 1-1 to 1-13
 \$\$EVM\$\$ constant 5-14
 hardware requirements 1-2
 host system 1-2
 I/O address space 1-4
 installation 1-4 to 1-6
 into PC 1-6 to 1-7
 preparation 1-4
 invoking the debugger 1-12, 2-3
 memory requirements 1-2
 power requirements 1-2
 resetting 1-3, 1-11
 software requirements 1-3
 switch settings 1-4
 \$\$EVM\$\$ constant 5-14
 evm16 command 2-3, 7-6
 options 1-12 to 1-13
 -i 7-7
 -s 7-6
 -b 1-12
 D_OPTIONS environment variable 1-10
 -i 1-12
 -p 1-13
 -s 1-13
 -t 1-13
 -v 1-13
 -x 1-13
 evminit.cmd file 6-2, A-1
 executing code 2-11, 7-8 to 7-12
 See also run commands
 conditionally 2-17, 7-12
 function key method 11-35
 halting execution 2-14, 7-12
 program entry point 2-14, 7-8 to 7-12
 single stepping 2-16, 11-23, 11-30
 while disconnected from the target system 7-11, 11-25
 executing commands 5-3
 exiting the debugger 1-13, 2-21, 11-24
 expressions 12-1 to 12-5
 addresses 8-6
 evaluation
 with ? command 8-2 to 8-3, 11-9
 with EVAL command 8-3, 11-15
 with LOOP command 5-14, 11-18
 expression analysis 12-4
 operators 12-2 to 12-3

expressions (continued)
 restrictions 12-4
 side effects 8-5
 void expressions 12-4
 while editing disassembly 7-5
 extensions 3-7

F

F2 key 5-5, 11-34
 F5 key 5-11, 7-9, 11-7, 11-35
 F6 key 2-6, 4-12, 8-4, 11-36
 F8 key 5-11, 7-10, 11-7, 11-35
 F9 key 4-5, 4-21, 8-5, 9-2, 9-3, 11-36
 F10 key 5-11, 7-10, 11-7, 11-35
 FILE command 2-11, 4-5, 7-5, 11-15
 changing the current directory 5-19, 11-12
 menu selection 11-7
 FILE window 2-11, 4-3, 4-5, 7-5
 file/load commands 11-5
 ADDR command 4-5, 7-3, 7-8, 11-9
 DASM command 4-5, 7-3, 7-8, 11-13
 FILE command 2-11, 4-5, 7-5, 11-15
 LOAD command 2-4, 7-6, 11-17
 menu selections 11-7
 PATCH command 7-3, 11-23
 RELOAD command 7-6, 11-24
 RESTART command 2-15, 11-24
 SLOAD command 7-6, 11-29
 files
 log files 5-6, 11-14
 saving memory to a file 8-8, 11-22
 FILL command 8-9, 11-16
 menu selection 11-8
 floating-point operations 12-4

G

GO command 2-11, 7-9, 11-16
 graphics card requirements 1-2
 grouping/reference operators 12-2

H

HALT command 7-11, 11-16
 halting
 batch file execution 5-12
 debugger 1-13, 2-21, 11-24
 program execution 1-13, 2-14, 7-8, 7-12
 function key method 7-12, 11-35
 mouse method 7-12
 target system 11-16
 hardware checklist 1-2
 hexadecimal notation
 addresses 8-6
 data formats 8-14
 history, of commands 5-5
 home key, scrolling 4-20, 11-37
 host system 1-2

I

-i debugger option 1-12, 7-7
 with D_OPTIONS environment variable 1-10
 I/O address space 1-4
 I/O switch settings, default settings 1-4, 1-5
 IF/ELSE/ENDIF commands 5-13 to 5-14, 11-4, 11-17
 conditions 5-15, 11-17
 predefined constants 5-14
 increment operator 12-3
 index numbers, for data in WATCH window 4-9, 8-13
 indirection operator (*), 8-13
 init.clr file 1-3, 10-9, 10-10, 11-26, A-1
 init.cmd file 1-3, 6-2, 6-3, A-1
 initdb.bat file 1-8 to 1-11
 invoking 1-9
 sample 1-8
 initialization batch files 6-2, A-1
 creating using LOOP/ENDLOOP 5-14 to 5-15
 evminit.cmd 6-2, A-1
 init.cmd 1-3, 6-2, 6-3, A-1
 installation 1-4 to 1-6
 board into PC 1-6
 debugger software 1-7
 invalid memory addresses 6-3, 6-5

invoking

- autoexec.bat file 1-9
- custom displays 10-11
- debugger 1-12, 2-3
- initdb.bat file 1-9

K

key sequences

- displaying functions 11-36
- displaying previous commands (command history), 11-34
- editing
 - command line* 5-3, 11-34
 - data values* 4-21, 11-36
- halting actions 11-35
- menu selections 11-35
- moving a window 4-18, 11-36
- running code 11-35
- scrolling 4-20, 11-37
- selecting the active window 4-12, 11-36
- setting/clearing breakpoints 11-36
- single stepping 7-10
- sizing a window 4-15, 11-36

L

labels

- for data in WATCH window 2-16, 4-9, 8-13
- while editing disassembly 7-5

limits

- command aliasing 5-17
- paths 7-7
- window positions 4-17
- window sizes 4-14

linker 1-3, 3-6, 3-7

- command files, MEMORY definition 6-2

LOAD command 2-4, 7-6, 11-17

- effects on WATCH window 8-13

load/file commands 11-5

- ADDR command 4-5, 7-3, 7-8, 11-9
- DASM command 4-5, 7-3, 7-8, 11-13
- FILE command 2-11, 4-5, 7-5, 11-15
- LOAD command 2-4, 7-6, 11-17
- menu selections 11-7
- PATCH command 7-3, 11-23
- RELOAD command 7-6, 11-24
- RESTART command 2-15, 11-24
- SLOAD command 7-6, 11-29

loading

- batch files 5-12
- custom displays 10-10
- object code 2-3, 7-6
 - after invoking the debugger* 7-6
 - symbol table only* 7-6, 11-29
 - while invoking the debugger* 1-12, 7-6
 - without symbol table* 7-6, 11-24

log files 5-6, 11-14

logical operators 12-2

- conditional execution 7-12

LOOP/ENDLOOP commands 5-14 to 5-15, 11-4, 11-18

- conditions 5-15, 11-18

looping commands 5-14 to 5-15, 11-18

M

MA command 2-20, 6-4, 6-6, 11-18 to 11-19

- menu selection 11-8

managing data 8-1 to 8-16

- basic commands 8-2 to 8-3

MAP command 6-5, 11-19

- menu selection 11-8

MD command 2-20, 6-6, 11-19

- menu selection 11-8

MEM command 2-5, 4-6, 8-6 to 8-7, 11-20

- display formats 8-16, 11-20

memory

- batch file search order 6-2, A-1
- commands 11-6

FILL command 8-9, 11-16

menu selections 11-8

MS command 8-8, 11-22

data formats 8-14

data memory 2-20, 4-7

default map 1-3, 2-20, 6-3

filling 8-9, 11-16

invalid addresses 6-3

map

adding ranges 11-18

defining 6-2

in a batch file 6-2

interactively 6-2

deleting ranges 11-19

modifying 6-2

potential problems 6-3

resetting 11-22

- memory (continued)
 - mapping 2-20, 2-21, 6-1 to 6-8
 - adding ranges* 6-4
 - commands* 11-6
 - MA command 2-20, 6-4, 6-6, 11-18 to 11-19
 - MAP command 6-5, 11-19
 - MD command 2-20, 6-6, 11-19
 - menu selections 11-8
 - ML command 2-20, 6-5, 11-21
 - MR command 6-6, 11-22
 - deleting ranges* 6-6
 - enabling/disabling* 6-5
 - init.cmd* file 1-3
 - listing current map* 6-5
 - modifying* 6-2, 6-6
 - resetting* 6-6
 - returning to default* 6-7
 - nonexistent locations 6-2
 - program memory 2-20, 4-7, 8-7
 - protected areas 6-3, 6-5
 - requirements 1-2
 - saving 8-8, 11-22
 - undefined areas 6-3, 6-5
 - valid types 6-4
- MEMORY window 2-5, 4-3, 4-6 to 4-7, 8-2, 8-6, 11-20
 - colors 10-6
 - customizing 10-6
 - editing memory contents 8-4
 - modifying display 11-20
- memory-map commands. *See* memory, mapping, commands
- menu bar 2-4, 5-7
 - customizing its appearance 10-7
 - items without menus 5-11
 - using menus 5-7 to 5-11
- menu selections 5-7
 - colors 10-7
 - customizing their appearance 10-7
 - entering parameter values 5-10
 - escaping 5-9
 - function key methods 5-9, 11-35
 - list of menus 5-7
 - mouse methods 5-8
 - moving to another menu 5-9
 - usage 5-8
- messages B-1 to B-14
- Microsoft Windows, using with the debugger 1-7, 1-11
- MIX command 2-13, 11-21
 - menu selection 11-8
- mixed mode 2-12, 4-2
 - MIX command 2-13, 11-21
- ML command 2-20, 6-5, 11-21
 - menu selection 11-8
- modes
 - assembly mode 2-12, 4-2
 - commands 11-3
 - ASM command* 2-13, 11-10
 - menu selections* 11-8
 - MIX command* 2-13, 11-21
 - menu selections 2-12
 - mixed mode 2-12, 4-2
 - selection
 - command method* 2-13
 - mouse method* 2-12
- modifying
 - assembly language code 7-3
 - colors 10-2 to 10-7
 - command line 5-3
 - command-line prompt 10-12
 - current directory 5-19, 11-12
 - data values 8-4
 - memory map 6-2, 6-6
 - window borders 10-8
- mono.clr file 1-3, 10-9
- monochrome monitors 10-9
- mouse
 - cursor 4-10
 - requirements 1-2
- MOVE command 2-9, 4-17, 11-21
 - effect on entering other commands 5-4
- moving a window 4-16 to 4-18, 11-21
 - function key method 2-9, 4-18, 11-36
 - mouse method 2-9, 4-16
 - MOVE command 2-9, 4-17
 - XY screen limits 4-17
- MR command 6-6, 11-22
 - menu selection 11-8
- MS command 8-8, 11-22
 - menu selection 11-8

MS-DOS

- See also* system commands
- entering from the command line 5-18
- exiting from system shell 11-30
- SYSTEM command 5-18 to 5-19, 11-30

N

- natural format 12-4
- NEXT command 7-10, 11-23
 - from the menu bar 5-11
 - function key entry 5-11, 11-35
- nonexistent memory locations 6-2

O

- object files
 - creating 7-6
 - loading 1-12, 11-17
 - after invoking the debugger* 7-6
 - symbol table only* 1-13, 11-29
 - while invoking the debugger* 1-12, 2-3, 7-6
 - without symbol table* 1-13, 7-6, 11-24
- object format converter 3-6
- operators 12-2 to 12-3
 - * operator (indirection), 8-13
 - side effects 8-5
- overwrite editing 8-4

P

- p debugger option 1-13
 - with D_OPTIONS environment variable 1-10
- P register 8-10
 - loading 8-11
- page-up/page-down keys, scrolling 4-20, 11-37
- parameters
 - entering in a dialog box 5-10
 - evm16 command 1-12 to 1-13
 - patch assembly 7-3
- PATCH command 7-3, 11-23
- PATH statement 1-9
- PC 7-8
 - displaying contents of 2-5
 - finding the current PC 4-5
- PC register 8-10, 8-11
- PH register 8-10

PL register 8-10

- pointers
 - natural format 12-4
 - typecasting 12-5
- port address 1-10, 1-13
 - D_OPTIONS 1-10, 1-13
- power requirements, EVM board 1-2
- program
 - development for the 'C16, 3-5
 - entry point 7-8
 - resetting* 11-24
 - execution, halting 1-13, 2-14, 7-8, 7-12, 11-35
 - preparation for debugging 3-7
- program counter (PC), 8-10, 8-11
- program memory 4-7
 - adding to memory map 6-4, 11-18
 - deleting from memory map 6-6, 11-19
 - displaying 8-7
 - filling 8-9, 11-16
 - saving 8-8, 11-22
- PROMPT command 10-12, 11-24
 - menu selection 11-8
- pseudoregisters. *See* registers

Q

- QUIT command 1-13, 2-21, 11-24

R

- re-entering commands 5-5, 11-34
- recording COMMAND window displays 5-6, 11-4, 11-14
- registers 8-10
 - ACC register 8-10
 - ACCH register 8-10
 - ACCL register 8-10
 - AR0 register 8-10
 - AR1 register 8-10
 - displaying/modifying 8-10 to 8-11
 - P register 8-10
 - PC register 8-11
 - PH register 8-10
 - PL register 8-10
 - program counter (PC), 8-10, 8-11
 - referencing by name 12-4
 - ST register 8-10, 8-11
 - status register (ST), 8-10, 8-11
 - T register 8-10
 - TOS register 8-10

- relational operators 12-2
 - conditional execution 7-12
- relative pathnames 5-19, 7-7, 11-12
- RELOAD command 11-24
 - menu selection 11-7
- repeating commands 5-5, 11-34
- required files 1-3
- required software tools 1-3
- RESET command 2-4, 7-11, 11-24
 - menu selection 11-7
- resetting
 - EVM 1-3, 1-11
 - memory map 11-22
 - program entry point 11-24
 - target system 2-4, 7-11, 11-24
- RESTART (REST) command 2-15, 7-8, 11-24
 - menu selection 11-7
- restrictions
 - breakpoints 9-2
 - C expressions 12-4
- RUN command 2-14, 7-9, 11-25
 - from the menu bar 5-11
 - function key entry 5-11, 7-9, 11-35
 - menu bar selections 5-11
 - with conditional expression 2-17
- run commands 11-6
 - conditional parameters 2-17
 - GO command 2-11, 7-9, 11-16
 - HALT command 7-11, 11-16
 - menu bar selections 5-11, 11-7, 11-35
 - NEXT command 7-10, 11-23
 - RESET command 2-4, 7-11
 - RESTART command 2-15, 7-8
 - RUN command 2-14, 7-9, 11-25
 - RUNF command 7-11, 11-25
 - STEP command 2-16, 7-10, 11-30
- RUNF command 7-11, 11-25
- running programs 7-8 to 7-12
 - conditionally 7-12
 - halting execution 7-12
 - program entry point 7-8 to 7-12
 - while disconnected from the target system 7-11

S

- s debugger option 1-13, 7-6
 - with D_OPTIONS environment variable 1-10
- saving custom displays 10-10

- SCOLOR command 10-2, 11-25
 - menu selection 11-8
- SCONFIG command 10-10, 11-26
 - menu selection 11-8
- screen-customization commands 11-5
 - BORDER command 10-8, 11-11
 - COLOR command 10-2, 11-13
 - menu selections 11-8
 - PROMPT command 10-12, 11-24
 - SCOLOR command 10-2, 11-25
 - SCONFIG command 10-10, 11-26
 - SSAVE command 10-10, 11-29
- scrolling 2-10, 4-19
 - function key method 2-10, 4-20, 11-37
 - mouse method 2-10, 4-19 to 4-20, 8-7
- SETF command 8-14 to 8-16, 11-27
- side effects 8-5, 12-3
 - valid operators 8-5
- single-step
 - commands
 - menu bar selections 5-11
 - NEXT command 7-10, 11-23
 - STEP command 2-16, 7-10, 11-30
 - execution 7-9
 - assembly language code 7-10, 11-30
 - function key method 7-10, 11-35
 - mouse methods 7-10
- SIZE command 2-7, 4-14, 11-28
 - effect on entering other commands 5-4
- sizeof operator 12-4
- sizes
 - display 4-17
 - windows 4-14
- sizing a window 4-13 to 4-15
 - function key method 2-7, 4-15, 11-36
 - mouse method 2-7, 4-13
 - SIZE command 2-7, 4-14
 - size limits 4-14
 - while moving it 4-17, 11-21
- SLOAD command 7-6, 11-29
 - effects on WATCH window 8-13
 - menu selection 11-7
 - s debugger option 1-13
- software, developing for the 'C16, 3-5
- software checklist 1-3
- SOUND command 11-29, B-2
- SSAVE command 10-10, 11-29
 - menu selection 11-8
- ST register 8-10, 8-11

status register (ST), 8-10, 8-11

STEP command 2-16, 7-10, 11-30
 from the menu bar 5-11
 function key entry 5-11, 11-35

structures
 direct reference operator 12-2
 indirect reference operator 12-2

switch settings
 default settings 1-4, 1-5
 I/O address space 1-5, 1-10, 1-13
 your settings 1-5

symbol table, loading without object code 1-13, 7-6, 11-29

SYSTEM command 5-18 to 5-19, 11-30

system commands 5-18 to 5-20, 11-4
 ALIAS command 2-21, 5-15 to 5-17, 11-10
 CD command 2-18, 5-19, 7-7, 11-12
 CLS command 2-18, 5-5, 11-12
 DIR command 2-18, 5-19, 11-14
 DLOG command 5-6, 11-4, 11-14
 ECHO command 5-13, 11-4, 11-15
 from debugger command line 5-18
 IF/ELSE/ENDIF commands 5-13 to 5-14, 11-4, 11-17
 conditions 5-15, 11-17
 predefined constants 5-14
 LOOP/ENDLOOP commands 5-14 to 5-15, 11-4, 11-18
 conditions 5-15, 11-18
 QUIT command 1-13, 2-21, 11-24
 RESET command 2-4, 11-24
 SOUND command 11-29, B-2
 SYSTEM command 11-30
 system shell 5-19
 TAKE command 5-12, 6-7, 11-31
 UNALIAS command 11-31
 USE command 7-7, 11-31

system shells 5-18 to 5-19

T

-t debugger option 1-13
 during debugger invocation 6-2, A-1
 with D_OPTIONS environment variable 1-10

T register 8-10

TAKE command 5-12, 6-7, 11-31
 executing a log file 5-6

target system
 memory definition for debugger 6-1 to 6-8
 resetting 2-4, 11-24

terminating the debugger 11-24

text files, displaying 2-11, 7-5

TOS register 8-10

type casting 12-4

U

UNALIAS command 5-17, 11-31

USE command 7-7, 11-31

V

-v debugger option 1-13
 with D_OPTIONS environment variable 1-10

variables
 displaying in different numeric format 12-5
 displaying/modifying 8-12
 scalar values in WATCH window 4-9, 8-12 to 8-13

void expressions 12-4

W

WA command 2-16, 4-9, 8-11, 8-12 to 8-13, 11-32
 display formats 8-13, 8-16, 11-32
 menu selection 11-8

watch commands
 menu selections 8-12, 11-8
 WA command 2-16, 4-9, 8-11, 8-12 to 8-13, 11-32
 WD command 2-17, 4-9, 8-13, 11-32
 WR command 2-18, 4-9, 8-13, 11-33

WATCH window 2-16, 4-3, 4-9, 8-2, 8-12, 11-32, 11-33
 adding items 8-12 to 8-13, 11-32
 closing 4-21, 8-13, 11-33
 colors 10-6
 customizing 10-6
 deleting items 8-13, 11-32
 editing values 8-4
 effects of LOAD command 8-13
 effects of SLOAD command 8-13
 labeling watched data 8-13, 11-32
 opening 8-12 to 8-13, 11-32

WD command 2-17, 4-9, 8-13, 11-32
 menu selection 11-8

WIN command 2-6, 4-12, 11-33

window commands 11-3

See also windows, commands

windows 4-3 to 4-9

active window 4-10 to 4-12

border styles 10-8, 11-11

closing 4-21

COMMAND window 4-4, 5-2

commands

MOVE command 2-9, 4-17, 11-21

SIZE command 2-7, 4-14, 11-28

WIN command 2-6, 4-12, 11-33

ZOOM command 2-8, 4-16, 11-33

CPU window 4-8, 8-2, 8-10 to 8-11

DISASSEMBLY window 2-5, 4-5, 7-2

editing 4-21

FILE window 2-11, 4-5

MEMORY window 2-5, 4-6 to 4-7, 8-2, 8-6

moving 2-9, 4-16 to 4-18, 11-21

function keys 4-18, 11-36

mouse method 4-16

MOVE command 4-17

XY positions 4-17

windows (continued)

resizing 2-7, 4-13 to 4-15

function keys 4-15, 11-36

mouse method 4-13

SIZE command 4-14

size limits 4-14

while moving 4-17, 11-21

scrolling 2-10, 4-19

size limits 4-14

WATCH window 2-16, 4-9, 8-2, 8-12

zooming 2-8, 4-15 to 4-16

WR command 2-18, 4-9, 8-13, 11-33

menu selection 11-8

X

-x debugger option 1-13

Z

ZOOM command 2-8, 4-16, 11-33

zooming a window 4-15 to 4-16

mouse method 2-8, 4-16

ZOOM command 2-8, 4-16