TMS320C3x C Source Debugger User's Guide

SPRU053
November 1993

PRINTED WITH SOY INK ™

Texas Instruments

Printed on Recycled Paper

TMS320C3x C Source Debugger User's Guide
Reader Response Card

Texas Instruments wants to provide you with the best documentation possible—please help us by answering these questions and returning this card.

How have you used this manual?

- ☐ To look up specific information or procedures when needed (as a reference).
- ☐ To read chapters about subjects of specific interest.
- ☐ To read from front to back before using the product.

Did you use the Tutorial (Chapter 2)?

☐ Yes ☐ No

Which additional subjects should be included in future versions of the Tutorial?

Please describe any mistakes or misleading information that you found (include page numbers).

Which topics should this document describe in greater detail?

Please list information that was difficult to find and why (not in index, not in a logical location, etc.).

Please provide any specific suggestions that you have for improving the content of this document.

Are there specific, useful features of this user's guide that should be retained in future versions of this document?

Additional comments:

Thank you for taking the time to fill out this card.

Name ______________________________________ Title _________________________________

Company ____

Address ____

City ______________________________________ State _________ Zip/Country ______________

May we call you to discuss your comments? If so, please include phone number________________________

November 1993

TEXAS INSTRUMENTS

TMS320C3x C Source Debugger Reference Card

Phone Number

DSP Hotline: (713) 274–2320

Invoking the Debugger

Emulator:	**emu3x**	[*filename*]	[*–options*]
EVM:	**evm30**	[*filename*]	[*–options*]
Simulator:	**sim3x**	[*filename*]	[*–options*]

Debugger Options

Option	Description
–b[b]	**Select the screen size.** Option Characters. *none* 80 × 25 (default) –b 80 × 43 (EGA or VGA) –bb 80 × 50 (VGA only)
–i *pathname*	**Identify additional directories.** Identifies directories that contain source files.
–mm0 –mm1	**Select the mode.** (simulator only) Tells debugger to operate in microprocessor (0) or microcomputer (1) mode (0 is the default).
–mv30 –mv31	**Select the device version.** (simulator only) Identifies ’C30 or ’C31 memory map (’C30 is the default).
–p *port address*	**Identify the port address.** (emulator and EVM) Identifies the I/O port address that the debugger uses for communicating with the device.
–profile	**Enter the profiling environment.** (emulator and simulator) Brings up the debugger in a profiling environment.
–s	**Load symbol table only.** Tells the debugger to load *filename*’s symbol table only.
–t *filename*	**Identify new initialization file.** Allows you to specify an initialization file.
–v	**Load without symbol table.** Loads only global symbols; later, local symbols are loaded as needed. Affects **all** loads.
–x	**Ignore D_OPTIONS.** Ignores options supplied with D_OPTIONS.

Summary of Debugger Commands

? *expression* [, *display format*]	***d***
addr *address* **addr** *function name*	***d***
alias [*alias name* [, "*command string*"]]	***b***
asm	***d***
ba *address*	***d***
bd *address*	***d***
bl	***d***
border [*active*] [, [*inactive*] [,*resize*]]	***d***
br	***d***
c	***d***
calls	***d***
cd [*directory name*] **chdir** [*directory name*]	***b***
cls	***b***
cnext [*expression*]	***d***
color *area, attr*$_1$ [,*attr*$_2$ [,*attr*$_3$ [,*attr*$_4$]]]	***d***
cstep [*expression*]	***d***
dasm *address* **dasm** *function name*	***b***
dir [*directory*]	***b***
disp *expression* [, *display format*]	***d***
dlog *filename* [, {a|w}]	***d***
echo *string*	***b***
else	***d***
endif	***d***
endloop	***d***
eval *expression* **e** *expression*	***b***
file *filename*	***b***
fill *address*, *length, data*	***d***
func *function name* **func** *address*	***b***
go [*address*]	***d***
halt	***d***
if *Boolean expression* *debugger command list* [**else** *debugger command list*] **endif**	***b***
load *object filename*	***b***
loop *expression* *debugger command list* **endloop**	***b***
ma *address, length, type*	***b***
map {**on** | **off**}	***b***
mc *port address, filename,* {**READ** | **WRITE**}	***b*** †

p = profiler only
b = both profiler and basic debugger
d = basic debugger only
† simulator only

Summary of Debugger Commands	
md *address*	***b***
mem *expression* [, *display format*]	***d***
mi *port address,* {**READ**|**WRITE**}	***b*** †
mix	***d***
ml	***b***
move [*X*, *Y* [, *width*, *length*]]	***b***
mr	***b***
ms *address, length, filename*	***b***
next [*expression*]	***d***
patch *address*, *assembly language instruction*	***d***
pf *starting point* [, *update rate*]	***p***
pinc *pinname filename*	***d***
pind *pinname*	***d***
pinl	***d***
pq *starting point* [, *update rate*]	***p***
pr [*clear data* [, *update rate*]]	***p***
prompt *new prompt*	***b***
quit	***b***
reload *object filename*	***b***
reset	***b***
restart **rest**	***b***
return **ret**	***d***
run [*expression*]	***d***
runb	***d***
runf	***d***
sa *address*	***p***
scolor *area*, $attr_1$ [, $attr_2$ [, $attr_3$ [, $attr_4$]]]	***d***
sconfig [*filename*]	***b***
sd *address*	***p***
setf [*data type, display format*]	***d***
size [*width*, *length*]	***b***
sl	***p***
sload *object filename*	***b***
sound on | off	***d***
sr	***p***
ssave [*filename*]	***d***
step [*expression*]	***d***
system [*operating-system command* [, *flag*]]	***b***
take *batch filename* [*, suppress echo flag*]	***b***
unalias *alias name*	***b***
use *directory name*	***b***
vaa *filename*	***p***
vac *filename*	***p***

p = profiler only
b = both profiler and basic debugger
d = basic debugger only
† simulator only

Summary of Debugger Commands	
version	***b***
vr	***p***
wa *expression* [, [*label*], *display format*]	***d***
wd *index number*	***d***
whatis *symbol*	***d***
win *WINDOW NAME*	***b***
wr	***d***
zoom	***b***

p = profiler only
b = both profiler and basic debugger
d = basic debugger only
† simulator only

Border Styles (BORDER Command)

Index	Style
0	Double-lined box
1	Single-lined box
2	Solid 1/2-tone top, double-lined sides/bottom
3	Solid 1/4-tone top, double-lined sides/bottom
4	Solid box, thin border
5	Solid box, heavy sides, thin top/bottom
6	Solid box, heavy borders
7	Solid 1/2-tone box
8	Solid 1/4-tone box

Colors and Attributes (COLOR/SCOLOR Commands)

black	blue	green	cyan
red	magenta	yellow	white
bright		blink	

Area Names (COLOR/SCOLOR Commands)

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

Memory Types

To identify this kind of memory	Use this keyword as the *type* parameter
read-only memory	**R**, **ROM**, or **READONLY**
write-only memory	**W**, **WOM**, or **WRITEONLY**
read/write memory	**R|W** or **RAM**
no-access memory	**PROTECT**
input port	**IPORT** or **IN PORT**
output port	**OPORT** or **OUT PORT**
input/output port	**IOPORT**

Display Formats (?, DISP, MEM, SETF, and WA Commands)

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

† ?, DISP, SETF, and WA commands only

Switching Modes

To do this	Use this function key
Switch debugging modes in this order: → auto → assembly → mixed →	F3

Running Code

To do this	Use these function keys
Run code from the current PC	F5
Single-step from the current PC	F8
Single-step code from the current PC; step over function calls	F10

Selecting or Closing a Window

To do this	Use these function keys
Select the active window	F6
Close the CALLS or DISP window	F4

Editing Text on the Command Line

To do this	Use these function keys
Enter the current command	↵
Move back over text without erasing characters	CTRL H or BACK SPACE
Move forward through text without erasing characters	CTRL L
Move back over text while erasing characters	DELETE
Move forward through text while erasing characters	SPACE
Insert text into the characters that are already on the command line	INSERT

Using the Command History

To do this	Use these function keys
Repeat the last command that you entered	F2
Move backward, one command at a time, through the command history	TAB
Move forward, one command at a time, through the command history	SHIFT TAB

Editing Data or Selecting the Active Field

To do this	Use this function key
❑ *FILE or DISASSEMBLY window:* Set or clear a breakpoint ❑ *CALLS window:* Display the source to a listed function ❑ *Any data-display window:* Edit the contents of the current field ❑ *DISP window:* Open an additional DISP window	F9

Halting or Escaping From an Action

To do this	Use this function key
❑ Halt program execution ❑ Close a pulldown menu ❑ Undo an edit of the active field in a data-display window ❑ Halt the display of a long list of data	ESC

Displaying Pulldown Menus

To do this	Use these function keys
Display the Load menu	ALT L
Display the Break menu	ALT B
Display the Watch menu	ALT W
Display the Memory menu	ALT M
Display the Color menu	ALT C
Display the MoDe menu	ALT D
Display the Pin menu	ALT P
Display an adjacent menu	← or →
Execute any of the choices from a displayed pulldown menu	Press the high-lighted letter cor sponding to you choice

Moving or Sizing a Window

Enter the MOVE or SIZE command without parameters, then use the arrow keys:

To do this	Use these function keys
❑ Move the window down one line ❑ Make the window one line longer	↓
❑ Move the window up one line ❑ Make the window one line shorter	↑
❑ Move the window left one character position ❑ Make the window one character nar-rower	←
❑ Move the window right one character position ❑ Make the window one character wid-er	→

Scrolling the Active Window's Contents

To do this	Use these function keys
Scroll up through the window contents, one window length at a time	PAGE UP
Scroll down through the window contents, one window length at a time	PAGE DOWN
Move the field cursor up one line at a time	↑
Move the field cursor down one line at a time	↓
❑ *FILE window only:* Scroll left 8 characters at a time ❑ *Other windows:* 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	←
❑ *FILE window only:* Scroll right 8 characters at a time ❑ *Other windows:* Move the field cursor right 1 field; at the last field on a line, wrap around to the first field on the next line	→
FILE window only: Adjust the window's contents so that the first line of the text file is at the top of the window	HOME
FILE window only: Adjust the window's contents so that the last line of the text file is at the bottom of the window	END
DISP windows only: Scroll up through an array of structures	CTRL PAGE UP
DISP windows only: Scroll down through an array of structures	CTRL PAGE DOWN

TEXAS INSTRUMENTS

TMS320C3x C Source Debugger Profiler Reference Card

Basic Profiling Commands

Running a Profiling Session

Command	Description
pf *starting point* [, *update rate*]	Run a full profiling session
pq *starting point* [, *update rate*]	Run a quick profil-ing session
pr [*clear data* [, *update rate*]]	Resume a profilng session that has halted
pr [*clear data* [, *update rate*]]	Resume a profilng session that has halted

Defining Stopping Points

Command	Description
sa *address*	Add a stopping point
sd *address*	Delete a stopping point
sr	Delete all the stop-ping points
sl	View a list of all current stopping points

Saving Profile Data to a File

Command	Description
vac *filename*	Save the contents of the PROFILE window to a sys-tem file
vaa *filename*	Save all data for the current view

Phone Number

DSP Hotline: (713) 274–2320

Entering the Profiling Environment

The profiling evironment is supported under all platforms except DOS.

Emulator:	**emu3x –profile**
EVM:	**evm30 –profile**
Simulator:	**sim3x –profile**

Debugger Commands That Can Be Used in the Profiling Environment

?	LOAD	MR	SLOAD
ALIAS	MA	PROMPT	SYSTEM
CD	MAP	QUIT	TAKE
CLS	MC	RELOAD	UNALIAS
DASM	MD	RESET	USE
DIR	MI	RESTART	VERSION
EVAL	ML	SCONFIG	WIN
FILE	MOVE	SIZE	ZOOM
FUNC			

Debugger Commands That Can’t Be Used in the Profiling Environment

ADDR	CNEXT	MS	RUNF
ASM	COLOR	NEXT	SCOLOR
BA	CSTEP	PINC	SSAVE
BD	DISP	PIND	STEP
BL	FILL	PINL	WA
BORDER	GO	RETURN	WD
BR	HALT	RUN	WHATIS
C	MEM	RUNB	WR
CALLS	MIX		

Marking Areas

To mark this area	C only	Disassembly only
Lines		
❑ By line number, address	**MCLE** ***filename, line number***	**MALE** *address*
❑ All lines in a function	**MCLF** *function*	**MALF** *function*
Ranges		
❑ By line numbers	**MCRE** *filename, line number, line number*	**MARE** *address, address*
Functions		
❑ By function name	**MCFE** *function*	not applicable
❑ All functions in a module	**MCFM** *filename*	
❑ All functions everyhwhere	**MCFG**	

Disabling Marked Areas

To disable this area	C only	Disassembly only	C *and* Disassembly
Lines			
❑ By line number, address	**DCLE** ***filename, line number***	**DALE** *address*	not applicable
❑ All lines in a function	**DCLF** *function*	**DALF** *function*	**DBLF** *function*
❑ All lines in a module	**DCLM** *filename*	**DALM** *filename*	**DBLM** *filename*
❑ All lines everywhere	**DCLG**	**DALG**	**DBLG**
Ranges			
❑ By line numbers, addresses	**DCRE** *filename, line number*	**DARE** *address*	not applicable
❑ All ranges in a function	**DCRF** *function*	**DARF** *function*	**DBRF** *function*
❑ All ranges in a module	**DCRM** *filename*	**DARM** *filename*	**DBRM** *filename*
❑ All ranges everywhere	**DCRG**	**DARG**	**DBRG**
Functions			
❑ By function name	**DCFE** *function*	not applicable	not applicable
❑ All functions in a module	**DCFM** *filename*		**DBFM** *filename*
❑ All functions everyhwhere	**DCFG**		**DBFG**
All areas			
❑ All areas in a function	**DCAF** *function*	**DAAF** *function*	**DBAF** *function*
❑ All areas in a module	**DCAM** *filename*	**DAAM** *filename*	**DBAM** *filename*
❑ All areas everyhwhere	**DCAG**	**DAAG**	**DBAG**

Enabling Disabled Areas

To disable this area	C only	Disassembly only	C *and* Disassembly
Lines			
❑ By line number, address	**ECLE *filename, line number***	**EALE** *address*	not applicable
❑ All lines in a function	**ECLF** *function*	**EALF** *function*	**EBLF** *function*
❑ All lines in a module	**ECLM** *filename*	**EALM** *filename*	**EBLM** *filename*
❑ All lines everywhere	**ECLG**	**EALG**	**EBLG**
Ranges			
❑ By line numbers, addresses	**ECRE** *filename, line number*	**EARE** *address*	not applicable
❑ All ranges in a function	**ECRF** *function*	**EARF** *function*	**EBRF** *function*
❑ All ranges in a module	**ECRM** *filename*	**EARM** *filename*	**EBRM** *filename*
❑ All ranges everywhere	**ECRG**	**EARG**	**EBRG**
Functions			
❑ By function name	**ECFE** *function*	not applicable	not applicable
❑ All functions in a module	**ECFM** *filename*		**EBFM** *filename*
❑ All functions everyhwhere	**ECFG**		**EBFG**
All areas			
❑ All areas in a function	**ECAF** *function*	**EAAF** *function*	**EBAF** *function*
❑ All areas in a module	**ECAM** *filename*	**EAAM** *filename*	**EBAM** *filename*
❑ All areas everyhwhere	**ECAG**	**EAAG**	**EBAG**

Unmarking Areas

To disable this area	C only	Disassembly only	C *and* Disassembly
Lines			
❑ By line number, address	**UCLE *filename, line number***	**UALE** *address*	not applicable
❑ All lines in a function	**UCLF** *function*	**UALF** *function*	**UBLF** *function*
❑ All lines in a module	**UCLM** *filename*	**UALM** *filename*	**UBLM** *filename*
❑ All lines everywhere	**UCLG**	**UALG**	**UBLG**
Ranges			
❑ By line numbers, addresses	**UCRE** *filename, line number*	**UARE** *address*	not applicable
❑ All ranges in a function	**UCRF** *function*	**UARF** *function*	**UBRF** *function*
❑ All ranges in a module	**UCRM** *filename*	**UARM** *filename*	**UBRM** *filename*
❑ All ranges everywhere	**UCRG**	**UARG**	**UBRG**
Functions			
❑ By function name	**UCFE** *function*	not applicable	not applicable
❑ All functions in a module	**UCFM** *filename*		**UBFM** *filename*
❑ All functions everyhwhere	**UCFG**		**UBFG**
All areas			
❑ All areas in a function	**UCAF** *function*	**UAAF** *function*	**UBAF** *function*
❑ All areas in a module	**UCAM** *filename*	**UAAM** *filename*	**UBAM** *filename*
❑ All areas everyhwhere	**UCAG**	**UAAG**	**UBAG**

Changing the PROFILE Window Display

Viewing specific areas

To disable this area	C only	Disassembly only	C *and* Disassembly
Lines			
❑ By line number, address	**VFCLE *filename, line number***	**VFALE** *address*	not applicable
❑ All lines in a function	**VFCLF** *function*	**VFALF** *function*	**VFBLF** *function*
❑ All lines in a module	**VFCLM** *filename*	**VFALM** *filename*	**VFBLM** *filename*
❑ All lines everywhere	**VFCLG**	**VFALG**	**VFBLG**
Ranges			
❑ By line numbers, addresses	**VFCRE** *filename, line number*	**VFARE** *address*	not applicable
❑ All ranges in a function	**VFCRF** *function*	**VFARF** *function*	**VFBRF** *function*
❑ All ranges in a module	**VFCRM** *filename*	**VFARM** *filename*	**VFBRM** *filename*
❑ All ranges everywhere	**VFCRG**	**VFARG**	**VFBRG**
Functions			
❑ By function name	**VFCFE** *function*	not applicable	not applicable
❑ All functions in a module	**VFCFM** *filename*		**VFBFM** *filename*
❑ All functions everyhwhere	**VFCFG**		**VFBFG**
All areas			
❑ All areas in a function	**VFCAF** *function*	**VFAAF** *function*	**VFBAF** *function*
❑ All areas in a module	**VFCAM** *filename*	**VFAAM** *filename*	**VFBAM** *filename*
❑ All areas everyhwhere	**VFCAG**	**VFAAG**	**VFBAG**

Viewing different data

To view this information	Use this command
Count	**VDC**
Inclusive	**VDI**
Inclusive, maximum	**VDN**
Exclusive	**VDE**
Exclusive, maximum	**VDX**
Address	**VDA**
All	**VDL**

Sorting the data

To sort on this data	Use this command
Count	**VSC**
Inclusive	**VSI**
Inclusive, maximum	**VSN**
Exclusive	**VSE**
Exclusive, maximum	**VSX**
Address	**VSA**
Data	**VSD**

IMPORTANT NOTICE

Texas Instruments Incorporated (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 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. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Please be aware that TI products are not intended for use in life-support appliances, devices, or systems. Use of TI product in such applications requires the written approval of the appropriate TI officer. Certain applications using semiconductor devices may involve potential risks of personal injury, property damage, or loss of life. In order to minimize these risks, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards. Inclusion of TI products in such applications is understood to be fully at the risk of the customer using TI devices or systems.

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.

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.

Preface

Read This First

What Is This Book About?

This book tells you how to use the TMS320C3x C source debugger with these debugging tools:

- Emulator
- Evaluation module (EVM)
- Simulator

All three tools support code development for both the TMS320C30 and the TMS320C31. Each tool has its own version of the debugger. These versions operate almost identically; however, the executable files that invoke them are very different. For example, the EVM version won't work with the emulator or simulator, and vice versa. Separate commands are provided for invoking each version of the debugger.

There are two debugger environments: the basic debugger environment and the profiling environment. The basic debugger environment is a general-purpose debugging environment. The profiling environment is a special environment for collecting statistics about code execution. Both environments have the same easy-to-use interface.

Before you use this book, you should read the appropriate installation guide to install the C source debugger and any necessary hardware.

How to Use This Manual

The goal of this book is to help you learn to use the Texas Instruments advanced programmer's interface for debugging. 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 lists the key features of the debugger, describes additional 'C3x software tools, tells you how to prepare a 'C3x program for debugging, and provides instructions and options for invoking the 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.

  The chapters in Part II detail the individual topics that are introduced in the tutorial. For example, Chapter 3 describes all of the debugger's windows and tells you how to move them and size them; Chapter 4 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; however, the debugger can also be used to debug assembly language programs. The information about C expressions will aid assembly language programmers who are unfamiliar with C.
  - Part III also includes a glossary and an index.

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

- If you have used TI development tools or other debuggers before, then you may want to:
  - Read the introductory material in Chapter 1.
  - Complete the tutorial in Chapter 2.
  - Read through 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:
  - Read the introductory material in Chapter 1.
  - Complete the tutorial in Chapter 2.
  - Read all of the chapters in Part II.

Notational Conventions

This document uses the following conventions.

- The TMS320C30 and TMS320C31 processors are referred to collectively as the **’C3x**.

- The C source debugger has a very flexible command-entry system; there are 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 pressing function keys. There are three symbols to identify the methods that you can use to perform an action:

Symbol	Description
	Identifies an action that you perform by using the mouse.
	Identifies an action that you perform by using function keys.
	Identifies an action that you perform by typing in a command.

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

Symbol	Action
↖	*Point*. Without pressing a mouse button, move the mouse to point the cursor at a window or field on the display. (Note that the mouse cursor displayed on the screen is not shaped like an arrow; it’s shaped like a block.)
	Press and hold. Press a mouse button. If your mouse has only one button, press it. If your mouse has more than one button, press the left button.
	Release. Release the mouse button that you pressed.
	Click. Press a mouse button and, without moving the mouse, release the button.
	Drag. While pressing the left mouse button, move the mouse.

- Debugger commands are not case sensitive; you can enter them in lowercase, uppercase, or a combination. To emphasize this fact, commands are shown throughout this user's guide in both uppercase and lowercase.

- Program listings and examples, interactive displays, and window contents are shown in a special font. Some examples use a **bold version** to identify code, commands, or portions of an example that *you* enter. Here is an example:

Command	Result displayed in the COMMAND window
`whatis giant`	`struct zzz giant[100];`
`whatis xxx`	`struct xxx { int a; int b; int c; int f1 : 2; int f2 : 4; struct xxx *f3; int f4[10]; }`

  In this example, the left column identifies debugger commands that you type in. The right column identifies the result that the debugger displays in the 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:

  mem *expression* [, *display format*]

  mem is the command. This command has two parameters, indicated by *expression* and *display format*. The first parameter must be an actual C expression; the second parameter, which identifies a specific display format, 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.

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

  sound {**on** | **off**}

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

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

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.

CAUTION

Please read each caution statement carefully.

Related Documentation From Texas Instruments

The following books describe the TMS320C3x DSPs and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C3x User's Guide (literature number SPRU031) describes the 'C3x 32-bit floating-point microprocessor (developed for digital signal processing as well as general applications), its architecture, internal register structure, instruction set, pipeline, specifications, and DMA and serial port operation. Software and hardware applications are included.

TMS320 Floating-Point DSP Assembly Language Tools User's Guide (literature number SPRU035) 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 'C3x and 'C4x generations of devices.

TMS320 Floating-Point DSP Optimizing C Compiler User's Guide (literature number SPRU024) describes the TMS320 floating-point C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C3x and 'C4x generations of devices.

Digital Signal Processing Applications With the TMS320C30 Evaluation Module Selected Application Notes (literature number SPRA021) contains useful information for people who are preparing and debugging code. The book gives additional information about the TMS320C30 EVM, as well as C coding tips.

TMS320C30 Evaluation Module Technical Reference (literature number SPRU069) describes board-level operation of the TMS320C30 EVM.

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

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

If You Need Assistance. . .

If you want to. . .	**Do this. . .**
Request more information about Texas Instruments Digital Signal Processing (DSP) products	Write to: Texas Instruments Incorporated Market Communications Manager, MS 736 P.O. Box 1443 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call the TI Literature Response Center: **(800) 477–8924**
Ask questions about product operation or report suspected problems	Call the DSP hotline: **(713) 274–2320** **FAX: (713) 274–2324**
Report mistakes in this document or any other TI documentation	Fill out and return the reader response card at the end of this book, or send your comments to: Texas Instruments Incorporated Technical Publications Manager, MS 702 P.O. Box 1443 Houston, Texas 77251–1443

Trademarks

PC-DOS and OS/2 are trademarks of International Business Machines Corp.

MS-DOS and MS-Windows are registered trademarks of Microsoft Corp.

Motorola-S is a trademark of Motorola, Inc.

SunOS, Sun-3, Sun-4, and OpenWindows are trademarks of Sun Microsystems, Inc.

Tektronix is a trademark of Tektronix, Inc.

UNIX is a registered trademark of Unix System Laboratories, Inc.

Contents

Part I: Hands-On Information

Part III: Reference Material

Figures

Tables

Chapter 1

Overview of a Code Development and Debugging System

The TMS320C3x C source debugger is an advanced programmer's interface that helps you to develop, test, and refine 'C3x C programs (compiled with the 'C3x optimizing ANSI C compiler) and assembly language programs. The debugger is the interface to the 'C3x simulator, EVM, and unique scan-based, realtime emulator.

This chapter gives an overview of the programmer's interface, describes the 'C3x code development environment, and provides instructions and options for invoking the debugger.

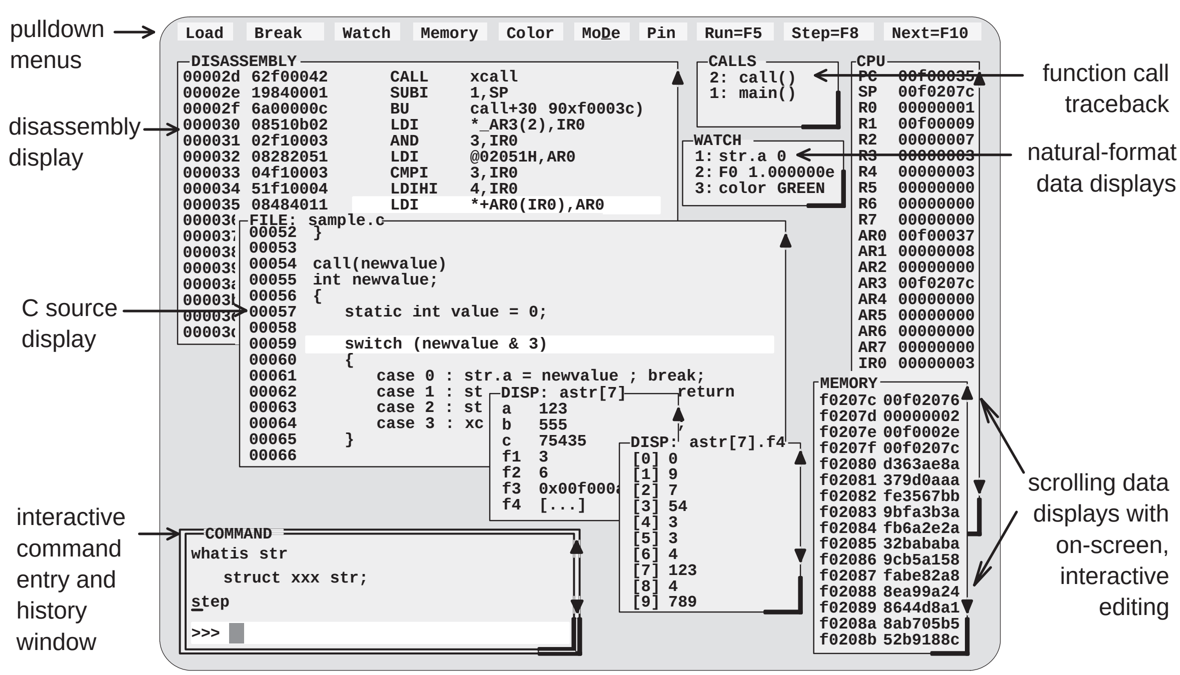
1.1 Description of the ’C3x C Source Debugger

The ’C3x C source debugger interface improves productivity by allowing you to debug a program in the language it was written in. You can choose to debug your programs in C, assembly language, or both. And, unlike many other debuggers, the ’C3x debugger’s higher level features are available even when you’re debugging assembly language code.

The Texas Instruments advanced programmer’s interface 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 1–1 identifies several features of the debugger display.

Figure 1–1. The Basic Debugger Display

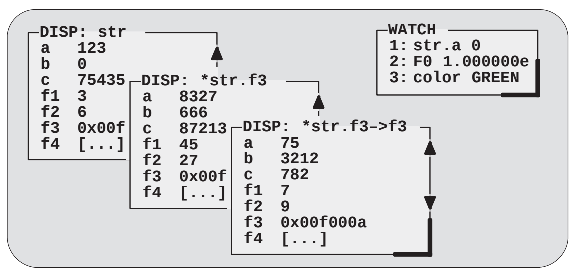


Key features of the debugger

- **Multilevel debugging.** The debugger allows you to debug both C and assembly language code. If you're debugging a C program, you can choose to view just the C source, the disassembly of the object code created from the C source, or both. You can also use the debugger as an assembly language debugger.

- **Fully configurable, state-of-the-art, window-oriented interface.** The C source debugger separates code, data, and commands into manageable portions. Use any of the default displays. Or, select the windows you want to display, size them, and move them where you want them.

- **Comprehensive data displays.** You can easily create windows for displaying *and editing* the values of variables, arrays, structures, pointers—any kind of data—in their natural format (*float*, *int*, *char*, *enum*, or *pointer*). You can even display entire linked lists.



- **On-screen editing.** Change any data value displayed in any window—just point the mouse, click, and type.

- **Continuous update.** The debugger continuously updates information on the screen, highlighting changed values.

- **Powerful command set.** Unlike many other debugging systems, this debugger doesn't force you to learn a large, intricate command set. The 'C3x C source debugger supports a small but powerful command set that makes full use of C expressions. One debugger command performs actions that would take several commands in another system.

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


❑ **Create your own debugger.** The debugger display is completely configurable, allowing you to create the interface that is best suited for your use.

- If you’re using a color display, you can change the colors of any area on the screen.
- You can change the physical appearance of display features such as window borders.
- You can interactively set the size and position of windows in the display.

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

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

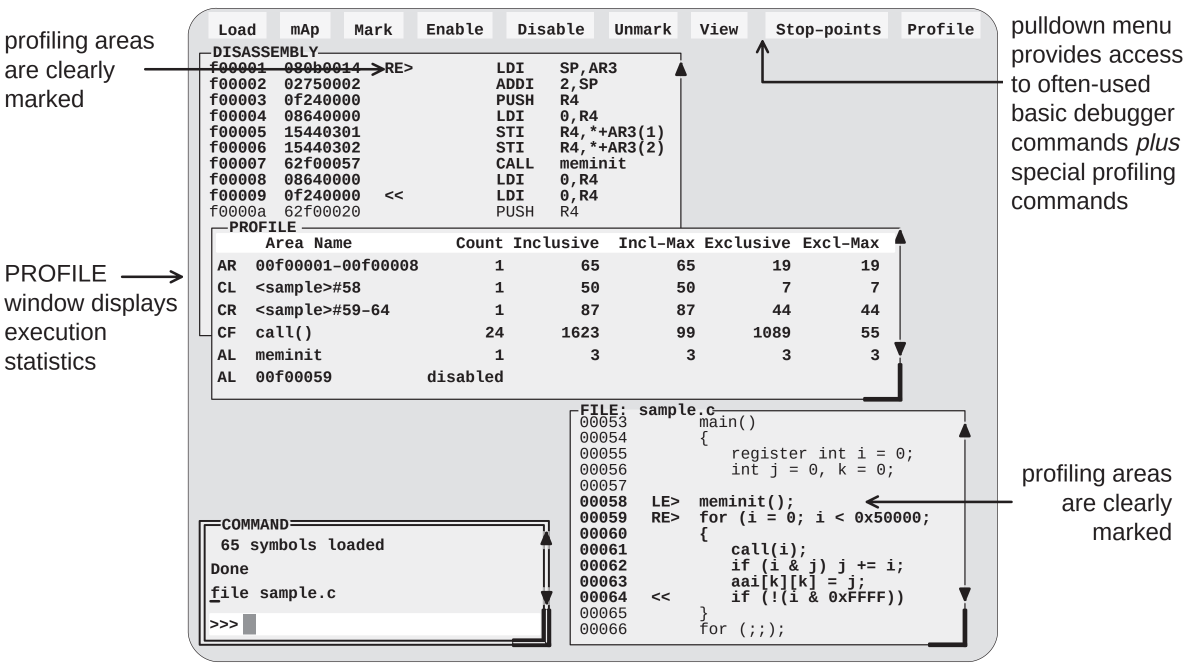
❑ **All the standard features you expect in a world-class debugger.** The debugger provides you with complete control over program execution with features like conditional execution and single-stepping (including single-stepping into or over function calls). You can set or clear a breakpoint with a click of the mouse or by typing commands. You can define a memory map that identifies the portions of target memory that the debugger can access. You can choose to load only the symbol table portion of an object file to work with systems that have code in ROM. The debugger can execute commands from a batch file, providing you with an easy method for entering often-used command sequences.

1.2 Description of the Profiling Environment

In addition to the basic debugging environment, a second environment—the *profiling environment*—is available. The profiling environment provides a method for collecting execution statistics about specific areas in your code. This gives you immediate feedback on your application's performance. The profiler is *not* available when you're running the debugger under DOS.

Figure 1–2 identifies several features of the debugger display within the profiling environment.

Figure 1–2. The Profiling-Environment Display



Key features of the profiling environment

The profiling environment builds on the same easy-to-use interface available in the basic debugging environment and provides these additional features:

- **More efficient code.** Within the profiling environment, you can quickly identify busy sections in your programs. This helps you to direct valuable development time toward streamlining the sections of code that most dramatically affect program performance.

- **Statistics on multiple areas.** You can collect statistics about individual statements in disassembly or C, about ranges in disassembly or C, and about C functions. When you are collecting statistics on many areas, you can choose to view the statistics for all the areas or a subset of the areas.

- **Comprehensive display of statistics.** The profiler provides all the information you need for identifying bottlenecks in your code:

  - The number of times each area was entered during the profiling session.
  - The total execution time of an area, including or excluding the execution time of any subroutines called from within the area.
  - The maximum time for one iteration of an area, including or excluding the execution time of any subroutines called from within the area.

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

- **Configurable display of statistics.** Display the entire set of data, or display one type of data at a time. Display all the areas you're profiling, or display a selected subset of the areas.

- **Visual representation of statistics.** When you choose to display one type of data at a time, the statistics will be accompanied by histograms for each area, showing the relationship of each area's statistics to those of the other profiled areas.

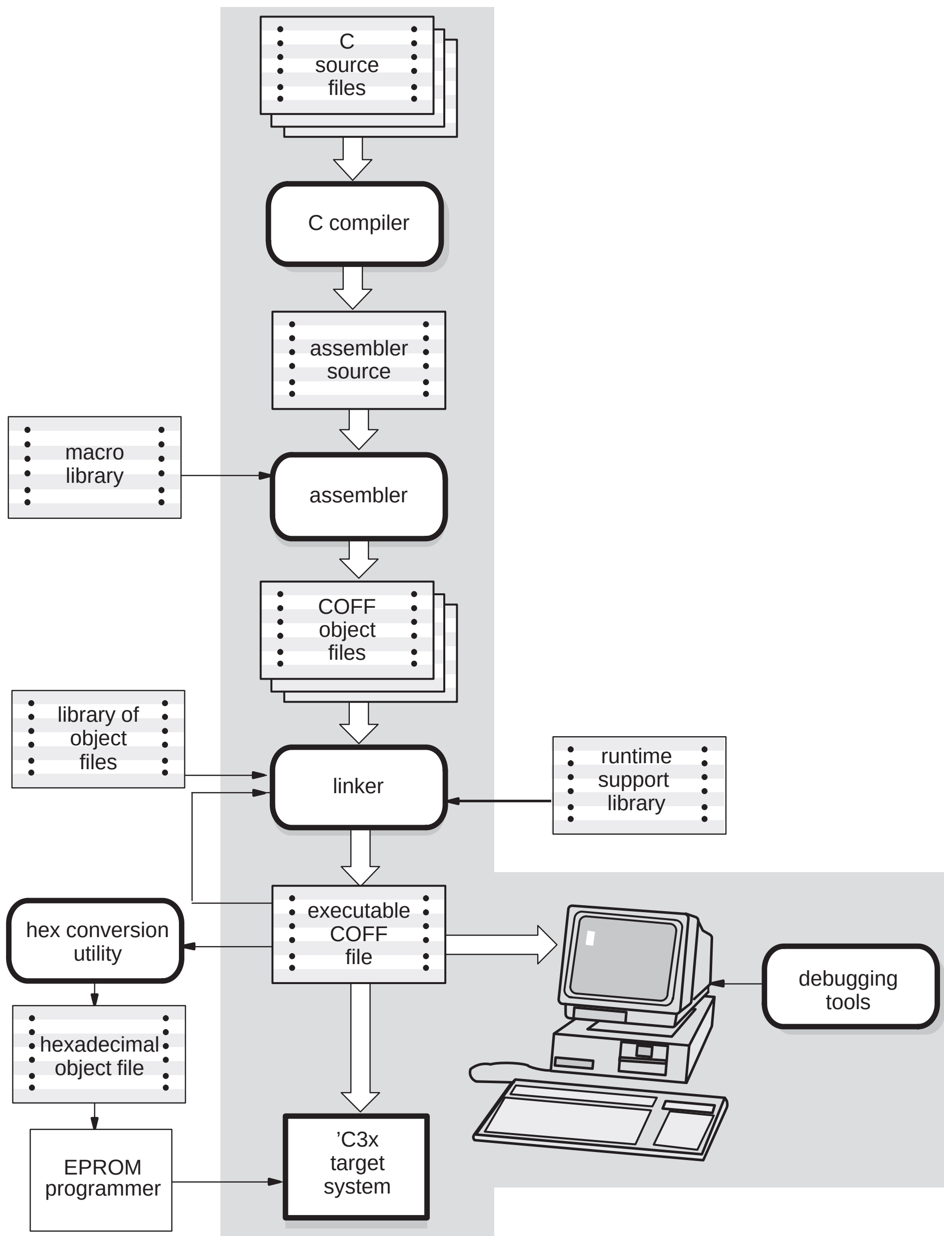
- **Disabled areas.** In addition to identifying areas that you can collect statistics on, you can also identify areas that you don't want to affect the statistics. This removes the timing impact from code such as a standard library function or a fully optimized portion of code.

- **Special profiling commands.** The profiling environment supports a rich set of commands to help you select areas and display information. Some of the basic debugger commands—such as the memory map commands—may be necessary during profiling and are available within the profiling environment. Other commands—such as breakpoint commands and run commands—are not necessary and are therefore not available within the profiling environment.

1.3 Developing Code for the ’C3x

The ’C3x is well supported by a complete set of hardware and software development tools, including a C compiler, assembler, and linker. Figure 1–3 illustrates the ’C3x code development flow. The most common paths of software development are highlighted in grey; the other portions are optional.

Figure 1–3. ’C3x Software Development Flow



These tools use common object file format (COFF), which encourages modular programming. COFF allows you to divide your code into logical blocks, define your system’s memory map, and then link code into specific memory areas. COFF also provides rich support for source-level debugging.

The following list describes the tools shown in Figure 1–3.

C compiler

The ’C3x **optimizing ANSI C compiler** is a full-featured optimizing compiler that translates standard ANSI C programs into ’C3x assembly language source. Key characteristics include:

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

assembler

The **assembler** translates ’C3x 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 **’C3x target system.** You can use one of several **debugging tools** to refine and correct your code. Available products include:

- A realtime in-circuit **emulator**,
- An **evaluation module (EVM)**, and
- A software **simulator.**

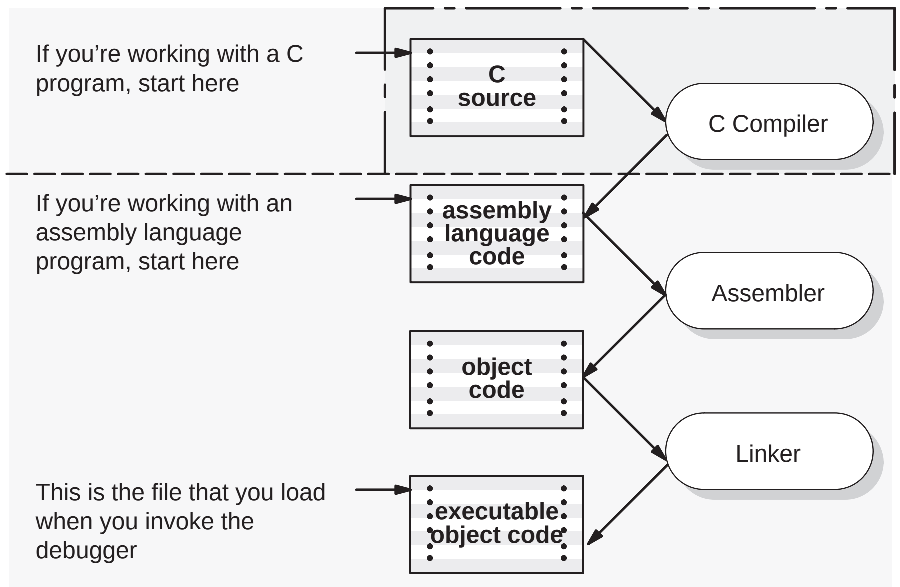
hex conversion utility

A **hex conversion utility** is also available; it converts a COFF object file into an ASCII-Hex, Intel, Motorola-S, Tektronix, or TI-tagged object-format file that can be downloaded to an EPROM programmer.

1.4 Preparing Your Program for Debugging

Figure 1–4 illustrates the steps you must go through to prepare a program for debugging.

Figure 1–4. Steps You Go Through to Prepare a Program



If you're preparing to debug a C program. . .	1) Compile the program; **use the –g option.** If you plan to use the profiler, compile the program with the –as option. 2) Assemble the resulting assembly language program. 3) Link the resulting object file. This produces an object file that you can load into the debugger.
If you're preparing to debug an assembly language program. . .	1) Assemble the assembly language source file. 2) Link the resulting object file. This produces an object file that you can load into the debugger.

You can compile, assemble, and link a program by invoking the compiler, assembler, and linker in separate steps, or you can perform all three actions in a single step by using the cl30 shell program. The *TMS320 Floating-Point DSP Assembly Language Tools User's Guide* and *TMS320 Floating-Point DSP Optimizing C Compiler User's Guide* contain complete instructions for invoking the tools individually and for using the shell program.

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

cl30 [**–***options*] **–g** [*filenames*] [**–z** [*link options*]]

cl30 is the command that invokes the compiler and assembler.

options affect the way the shell processes input files. If you plan to use the debugger's profiling environment, include the –as option.

filenames are one or more C source files, assembly language source files, or object files. Filenames are not case sensitive.

–g is an option that tells the C compiler to produce symbolic debugging information. When preparing a C program for debugging, you must use the –g option.

–z is an option that invokes the linker. After compiling/assembling your programs, you can invoke the linker in a separate step. If you want the shell to automatically invoke the linker, however, use –z.

link options affect the way the linker processes input files; use these options only when you use –z.

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

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

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

1.5 Invoking the Debugger

Here's the basic format for the commands that invoke the debugger:

emulator:	**emu3x**	[*filename*]	[*–options*]
EVM:	**evm30**	[*filename*]	[*–options*]
simulator:	**sim3x**	[*filename*]	[*–options*]

emu3x, evm30
sim3x are the commands that invoke the debugger.

filename is an optional parameter that names an object file that the debugger will load into memory during invocation. The debugger looks for the file in the current directory; if the file isn't in the current directory, you must supply the entire pathname. If you don't supply an extension for the filename, the debugger assumes that the extension is .out, unless you're using multiple extensions; you must specify the *entire* filename if the filename has more than one extension.

–options supply the debugger with additional information (Table 1–1 summarizes the available options).

You can also specify filename and option information with the D_OPTIONS environment variable (see *Setting up the environment variables* in the appropriate installation guide). Table 1–1 lists the debugger options and specifies which debugger tools use the options; the subsections following the table describe the options.

Table 1–1. Summary of Debugger Options

Option	Brief description	Debugger Tools
–b[b]	Select the screen size	All
–i *pathname*	Identify additional directories	All
–mm*mode*	Select the operating mode	Simulator
–mv*version*	Select the device version	Simulator
–p *port address*	Identify the port address	EVM and emulator
–profile	Enter the profiling environment	All, except when running under DOS
–s	Load the symbol table only	All
–t *filename*	Identify a new initialization file	All
–v	Load without the symbol table	All
–x	Ignore D_OPTIONS	All

Selecting the screen size (–b option)

By default, the debugger uses an 80-character-by-25-line screen. You can use one of the options in Table 1–2 to specify a different screen size. On Sun systems, you can resize the screen at runtime also.

Table 1–2. Screen Size Options

Option	Description	Display
none	80 characters by 25 lines	Default display
–b	80 characters by 43 lines	Any EGA or VGA display
–bb	80 characters by 50 lines	VGA only

Note:

On Sun systems, the maximum size of the debugger screen is 132 characters by 60 lines.

Identifying additional directories (–i option)

The –i option identifies additional directories that contain your source files. Replace *pathname* with an appropriate directory name. You can specify several pathnames; use the –i option as many times as necessary. For example:

emu3x **–i** *pathname*$_1$ **–i** *pathname*$_2$ **–i** *pathname*$_3$ . . .

Using –i is similar to using the D_SRC environment variable (see *Setting up the environment variables* in the appropriate installation guide). If you name directories with both –i and D_SRC, the debugger first searches through directories named with –i. The debugger can track a cumulative total of 20 paths (including paths specified with –i, D_SRC, and the debugger USE command).

Selecting the operating mode (–mm option)

The –mm option is valid only when you are using the simulator. The –mm option tells the simulator to operate in either the microprocessor or microcomputer mode:

–mm0 tells the simulator to operate in the microprocessor mode.
–mm1 tells the simulator to operate in the microcomputer mode (default).

If you don’t use the –mm option, the simulator operates in the microcomputer mode.

Selecting the device version (–mv option)

The –mv option is valid only when you are using the simulator. The –mv option tells the simulator to simulate the ’C30 or the ’C31 memory map:

–mv30 tells the simulator to simulate the ’C30 memory map (default).
–mv31 tells the simulator to simulate the ’C31 memory map.

If you don’t use the –mv option, the simulator simulates the ’C30 memory map.

Identifying the port address (–p option)

The –p option is valid only when you are using the EVM or emulator. The –p option identifies the I/O port address that the debugger uses for communicating with the emulator or EVM. If you used the 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 the *Your Settings* table in the appropriate installation guide; depending on your switch settings, replace *port address* with one of these values:

Switch 1	Switch 2	Option
on	on	–p 240 (optional)
on	off	–p 280
off	on	–p 320
off	off	–p 340

If you didn’t note the I/O switch settings, you can use a trial-and-error approach to find the correct –p setting. If you use the wrong setting, you will see an error message when you invoke the debugger. (See the appropriate installation guide for more information.)

Entering the profiling environment (–profile option)

This option is *not* valid when you’re running the debugger under DOS. The –profile option allows you to bring up the debugger in a profiling environment so that you can collect statistics about code execution. Note that only a subset of the basic debugger features is available in the profiling environment.

Loading the symbol table only (–s option)

If you supply a *filename* when you invoke the debugger, you can use the –s option to tell the debugger to load only the file’s symbol table (without the file’s object code). This is similar to loading a file by using the debugger’s SLOAD command.

Identifying a new initialization file (–t option)

The –t option allows you to specify an initialization command file that will be used instead of init.cmd. The format for this option is:

–t *filename*

Loading without the symbol table (–v option)

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

The –v option affects all loads, including those performed when you invoke the debugger and those performed with the LOAD command within the debugger environment.

Ignoring D_OPTIONS (–x option)

The –x option tells the debugger to ignore any information supplied with D_OPTIONS. For more information about D_OPTIONS, refer to the appropriate installation guide.

1.6 Exiting the Debugger

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

`quit` (ENTER)

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 running the debugger under MS-Windows, you can also exit the debugger by selecting the exit option from the MS-Windows menu bar.

1.7 Debugging ’C3x 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.

Step 1

Prepare a C program or assembly language program for debugging.

See Section 1.4, *Preparing Your Program for Debugging*, page 1-10.

Step 2

Ensure that the debugger has a valid memory map.

See Chapter 5, *Defining a Memory Map.*

Step 3

Load the program’s object file.

See Section 6.3, *Loading Object Code*, page 6-10.

Step 4

Run the loaded file. You can run the entire program, run parts of the program, or single-step through the program.

See *Running Your Programs* on page 6-12.

Step 5

Stop the program at critical points and examine important information.

See Chapter 8, *Using Software Breakpoints,* and Chapter 7, *Managing Data.*

Step 6

If you find minor problems in your code, you can temporarily solve them with patch assembly.

See *Modifying assembly language code* on page 6-5.

Step 7

Once you have decided what changes must be made to your program, exit the debugger, edit your source file, and return to Step 1.

Chapter 2

An Introductory Tutorial to the C Source Debugger

This chapter provides a step-by-step, hands-on demonstration of the ’C3x C source 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. The tutorial contains two sets of lessons (11 in the first, 13 in the second) and takes about one hour to complete.

Topic **Page**

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 that 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 debugger (including invoking any files or DOS commands as instructed in the installation guide).

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 accidently 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.

Note:

The –b option is not supported with the VAX/VMS version of the simulator.

Important! When using the emulator or EVM, 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 individual installation guides).

Invoke the debugger and load the sample program:

- ❑ For the **emulator**, enter:

  emu3x –b c:\c3xhll\sample ↵

- ❑ For the **EVM**, enter:

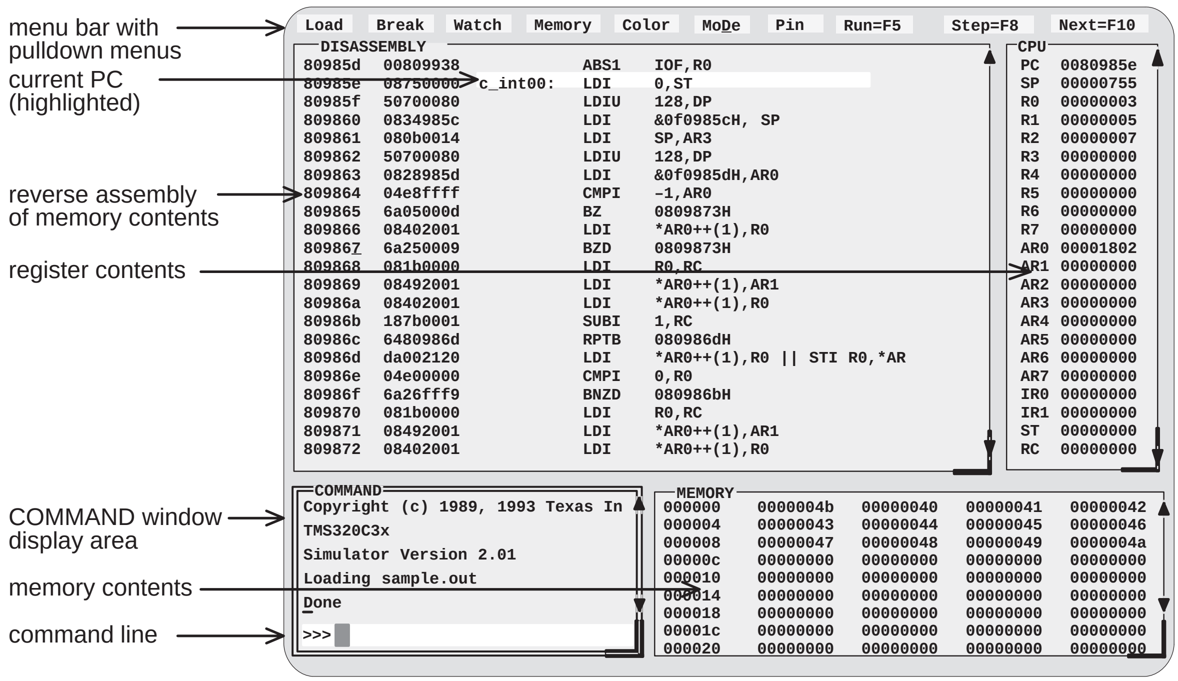
  evm30 –b c:\c3xhll\sample ↵

- ❑ For the **simulator**, enter:

  sim3x –b c:\sim3x\sample ↵

Take a look at the display. . .

Now you should see a display similar to this. The code should be the same on your screen, but your window sizes may vary.



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

  1) Reset the 'C3x processor:

     `reset`

  2) Load the sample program again:

     `load c:\c3xhll\sample` (emulator and EVM)

     `load c:\sim3x\sample` (simulator)

- After reset, if you see a display and the first few lines of the DISASSEMBLY window still show ADD instructions or say *Invalid address*, your EVM or emulator board may not be installed snugly. Check your board to see if it is correctly installed, and re-enter the commands above.

What's in the DISASSEMBLY window?

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

This tutorial step demonstrates that the code shown in the DISASSEMBLY window corresponds to memory contents. Initially, memory is displayed starting at address 0; if you look at the first line of the DISASSEMBLY window, you'll see that its display starts at address 0x0080 985d.

> Modify the MEMORY display to show the same object code that is displayed in the DISASSEMBLY window:
>
> **mem 0x80985d**

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 highlighted statement in the DISASSEMBLY window shows that the PC is currently pointing to address 0x0080 985e. You can modify the MEMORY display to show memory beginning from the current PC:

mem PC

Select the active window

This lesson shows you how to make a window 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 window's border.

2) Click the left mouse button.

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

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

  Press ESC *to get out of this.*

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

  To delete the breakpoint, point to the same statement and press the mouse button again.

Size the active window

This lesson shows you how to resize the active window.

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

Make the CPU window as small as possible:

`size 4,3`

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

Make the CPU window larger:

`size`	*Enter the SIZE command without parameters*
↓ ↓ ↓	*Make the window 3 lines longer*
→ → → →	*Make the window 4 characters wider*
ESC	*Press this key when you finish sizing the window*

You can use ↑ to make the window shorter and ← to make the window narrower.

Try This: You can 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:

`zoom`

The ZOOMcommand will be recognized, even though the COMMAND window is hidden by the CPU window.

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

Try This: You can 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 screen-size option you used when you invoked the debugger and on the position of the window before you tried to move it.

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 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 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.

↓ ↑ PAGE DOWN PAGE UP

These keys don't work the same for all windows; Section 11.5 (page 11-52) summarizes the functions of all the special keys, key sequences, and how they affect different windows.

Display the C source version of the sample file

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

> Display the contents of a C source file:
>
> **file sample.c** ↵

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

Execute some code

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

> Execute a portion of the sample program:
>
> **go main** ↵

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

- The current PC is highlighted in both the DISASSEMBLY and FILE windows.
- The addresses and object codes of four statements in the DISASSEMBLY window are highlighted; this is because these statements are associated with the current C statement (line 33 in the FILE window).
- The CALLS window, which tracks functions as they're called, now points to main().
- The values of the PC and SP (and possibly some additional registers) are highlighted in the CPU window because they were changed by program execution.

Become familiar with the three debugging modes

The debugger has three basic debugging modes:

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

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

The following steps show you how to switch debugging modes.

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

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

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

Switch to auto mode:

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

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

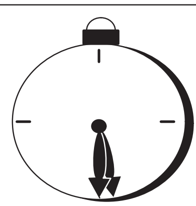
> Run to a point in your program that executes assembly language code:
>
> **`go meminit`**

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

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

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

> Switch back to mixed mode before continuing:
>
> **`mix`**



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.

Open another text file, then redisplay a C source file

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

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

> Display a file that isn't a C source file:
>
> **`file ..\autoexec.bat`** [Enter]
>
> This replaces sample.c in the FILE window with your autoexec.

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

> Redisplay another C source file (sample.c):
>
> **`func call`** [Enter]

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

Use the basic RUN command

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

> Run your entire program:
>
> **`run`** [Enter]

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

> This isn't very exciting: halt program execution:
>
> [ESC]

Set some breakpoints

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

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

Here's an example of one of the debugger's informative capabilities. In this example, you're going to benchmark some code; this means that you'll ask the debugger to count the number of CPU clock cycles that are consumed by a certain portion of code.

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

`file sample.c` ↵

Benchmark some code:

1) Scroll to line 38 in the FILE window (the meminit() statement) and set a breakpoint at that line:

   a) Point the mouse cursor at the statement on line 38.

   b) Click the left mouse button. *Notice how the line is highlighted; this identifies a breakpointed statement.*

2) Set another breakpoint at line 46 (the for (;;); statement).

3) Reset the program entry point:

   `restart` ↵

4) Enter the run command:

   `run` ↵ *This runs to the first breakpoint*

lesson continues on the next page →

5) Enter the runb command:

   `runb` *This runs to the second breakpoint*

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

   `? clk`

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

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

Delete both software breakpoints:

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

Watch some values and single-step through code

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

Set up for the single-step example:

```
restart
go main
```

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

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

> Open a WATCH window:
>
> ```
> wa sp
> wa pc, Program Counter
> wa *0x80981f, Call:
> wa i
> ```

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

If the WATCH window isn't wide enough to display the PC value, resize the window.

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

> Single-step through the sample program:
>
> ```
> step 50
> ```
>
> Observe the FILE, DISASSEMBLY, and WATCH windows.

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

- For example, if you enter:

  ```
  cstep 50
  ```

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

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

  ```
  restart
  go main
  ```

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

  ```
  next 50
  ```

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

Run code conditionally

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

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

> Delete the first three data items from the WATCH window (don't watch them anymore).
>
> ```
> wd 3
> wd 2
> wd 1
> ```

The variable i was the fourth item added to the WATCH window in the previous tutorial step, and it should now be the only remaining item in the window. (The sample program declares two variables named i: one is a global variable, and the other is local to main(). Because you executed code and are now in main() as a result of the previous step, you're watching the i variable that's local to main ().

> Set up for the conditional run examples:
>
> 1) Set software breakpoints at lines 38 and 44.
>
> 2) Set up for conditional run example:
>
> ```
> restart
> run
> ```
>
> 3) Initiate the conditional run:
>
> ```
> run i<10
> ```

This causes the debugger to run through the loop as long as the value of i is less than 10. Each time the debugger encounters the breakpoints in the loop, it updates the value of i in the WATCH window.

When the conditional run completes, close the WATCH window.

> Close the WATCH window:
>
> **wr**

WHATIS that?

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

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

whatis genum

```
      enum yyy genum;
```
genum is an enumerated type

whatis tiny6

```
      struct {
          int u;
          int v;
          int x;
          int y;
          int z;
      } tiny6;
```
tiny6 is a structure

whatis call

```
      int call();
```
call is a function that returns an integer

whatis s

```
      short s;
```
s is a short unsigned integer

whatis zzz

```
      struct zzz {
          int b1;
          int b2;
```
zzz is a very long structure

Press ESC *to halt long listings*

Clear the COMMAND window display area

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

Clear the COMMAND window display area:

cls ↵

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

cd ..	*Change back to the main directory*
dir	*Show a listing of the current directory*
cd c3xhll or **cd sim3x**	*Change back to the debugger directory*

Display the contents of an aggregate data type

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

Show a structure in a DISP window:

disp small ↵

Close the DISP window:

F4

Show another structure in a DISP window:

disp big1 ↵

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

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

(Note that the values displayed in this diagram may be different from what you see on the screen.)

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

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

Display what q3 is pointing to:

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

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

lesson continues on the next page →

Display array q1 in another DISP window:

1) Point at the [. . .] displayed next to the q1 label in big1's display.

2) Click the left mouse button.

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

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

key

Try This: Display structure q2 in another DISP window.

1) Close the additional DISP windows or move them out of the way so that you can clearly see the original DISP window that you opened to display big1.

2) Make big1's DISP window the active window.

↓ ↑ 3) Use these arrow keys to move the field cursor (_) through the list of big1's members until the cursor points to q2.

F9 4) Now press F9.

Close all of the DISP windows:

1) Make big1's DISP window the active window.

2) Press F4.

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

Display data in another format

Usually, when you add an item to the WATCH window or open a DISP window, the data is shown in its *natural format*. This means that ints are shown as integers, floats are shown as floating-point values, etc. Occasionally, you may wish to view data in a different format. This can be especially important if you want to show memory or register contents in another format.

One way to display data in another format is through casting (which is part of the C language). In the expression below, the *(float *) portion of the expression tells the debugger to treat address 0x809c00 as type float (exponential floating-point format).

> Display memory contents in floating-point format:
>
> **`disp *(float *)0x809c00`** ↵

This opens a DISP window to show memory contents in an array format. The array member identifiers don't necessarily correspond to actual addresses—they're relative to the first address you request with the DISP command. In this case, the item displayed as item [0] is the contents of address 0x0080 9c00—*it isn't memory location 0*. Note that you can scroll through the memory displayed in the DISP window; item [1] is at 0x0080 9c01, and item [–1] is at 0x0080 9bff.

You can also change display formats according to data type. This affects all data of a specific C data type.

> Change display formats according to data types by using the SETF (set format) command:
>
> 1) For comparison, watch the following variables. Their C data types are listed on the right.
>
> | | |
> |---|---|
> | **`wa i`** ↵ | *Type int* |
> | **`wa f`** ↵ | *Type float* |
> | **`wa d`** ↵ | *Type double* |
>
> 2) You can list all the data types and their current display formats:
>
> **`setf`** ↵

lesson continues on the next page →

3) Now display the following data types with new formats:

setf int, c	*Ints as characters*
setf float, o	*Floats as octal integers*
setf double, x	*Doubles as hex integers*

4) List the data types to display formats again; note the changes in the display:

setf

5) Add the variables to the WATCH window again; use labels to identify the additions:

wa i, NEWi
wa f, NEWf
wa d, NEWd

Notice the differences in the display formats between the first versions you added and these new versions.

6) Now reset all data types back to their defaults:

setf *

A third way to display data in another format is to use the DISP, ?, MEM, or WA command with an optional parameter that identifies the new display format. The following examples are for ? and WA—DISP and MEM work similarly.

Use display formats with the ? and WA commands:

1) Evaluate a variable and display it as a character:

? small.ra[1],c

2) Add a variable to the watch window and display it as an octal integer:

wa str.a,,o	*Notice that because no label was used with WA, an extra comma was inserted; otherwise, the **o** parameter would have been interpreted as a label.*

Try This: You can also watch registers R0–R7 as floating-point values by using the special symbols F0–F7. You might also want to display memory contents in floating-point format. For example, you can display the contents of location 0x809800 in floating-point format:

`disp *(float *)0x809800`

To get ready for the next step, close the DISP and WATCH windows.

Change some values

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

Change a value in memory:

1) Move or close the WATCH window if it's obscuring the MEMORY window; then display memory beginning with address 0x0080 9800:

   `mem 0x809800`

2) Point to the contents of memory location 0x0080 9800.

3) Click the left mouse button. *Notice that this highlights and identifies the field to be edited.*

4) Type 00000000.

5) Press [Enter] to enter the new value.

6) Press ESC to conclude editing.

lesson continues on the next page →

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 F9.
4) Type 0080985d.
5) Press ↓ twice. You should now be pointing at the contents of register R0.
6) Type 000174f9.
7) Press ↵ to enter the new value.
8) Press ESC to conclude editing.

Define a memory map

You can set up a memory map to tell the debugger which areas of memory it can and can't access. This is called *memory mapping*. When you invoked the debugger for this tutorial, the debugger automatically read a default memory map from the initialization batch file included in the c3xhll or sim3x directory. For the purposes of the sample program, that's fine (which is why this lesson was saved for the end).

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.

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 memory:

   `md 0x0`

   This deletes the block of memory beginning at address 0.

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

   `ma 0x2000,0xfff,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 shorthand 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, defining the same map that was defined in the previous lesson.

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 0x2000,0xfff,RAM;ml"

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

   mymap

   This is equivalent to entering the following three commands:

```
mr
ma 0x2000,0xfff,RAM
ml
```

Close the debugger

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

Close the debugger and return to the operating system:

quit

Chapter 3

The Debugger Display

The ’C3x C source debugger has a window-oriented display. This chapter shows what windows can look like and describes the basic types of windows that you’ll use.

3.1 Debugging Modes and Default Displays

The basic debugger environment has three debugging modes:

- ❑ Auto mode
- ❑ Assembly mode
- ❑ Mixed mode

Each mode changes the debugger display by adding or hiding specific windows. Some windows, such as the COMMAND window, may be present in all modes. The following figures show the default displays for these modes and show the windows that the debugger automatically displays for these modes.

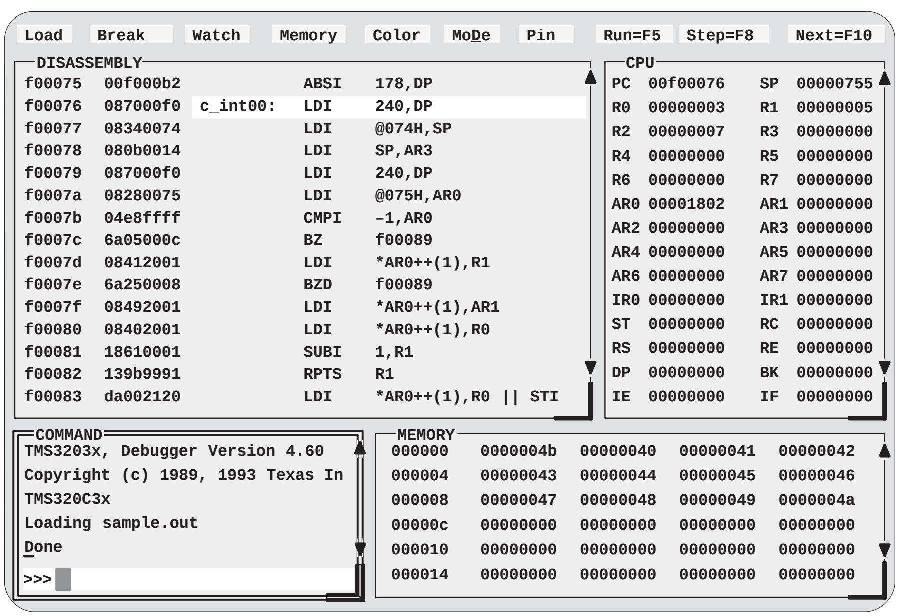
These modes cannot be used within the profiling environment; only the COMMAND, PROFILE, DISASSEMBLY, and FILE windows are available.

Auto mode

In **auto mode**, the debugger automatically displays whatever type of code is currently running—assembly language or C. This is the default mode; when you first invoke the debugger, you'll see a display similar to Figure 3–1. Auto mode has two types of displays:

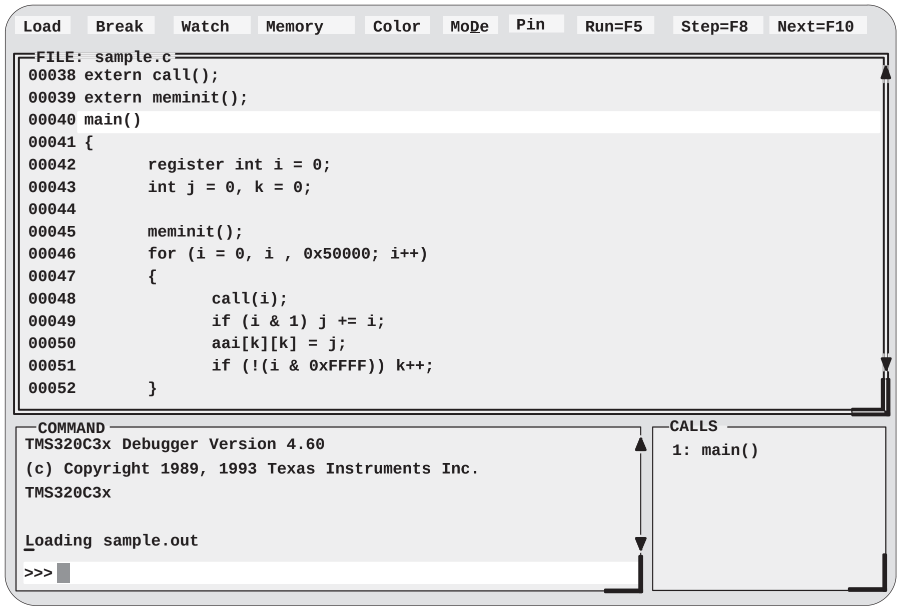
- ❑ When the debugger is running assembly language code, you'll see an assembly display similar to the one in Figure 3–1. The DISASSEMBLY window displays the reverse assembly of memory contents.

Figure 3–1. Typical Assembly Display (for Auto Mode and Assembly Mode)



- ❑ When the debugger is running C code, you'll see a C display similar to the one in Figure 3–2. (This assumes that the debugger can find your C source file to display in the FILE window. If the debugger can't find your source, then it switches to mixed mode.)

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



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

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

Assembly mode

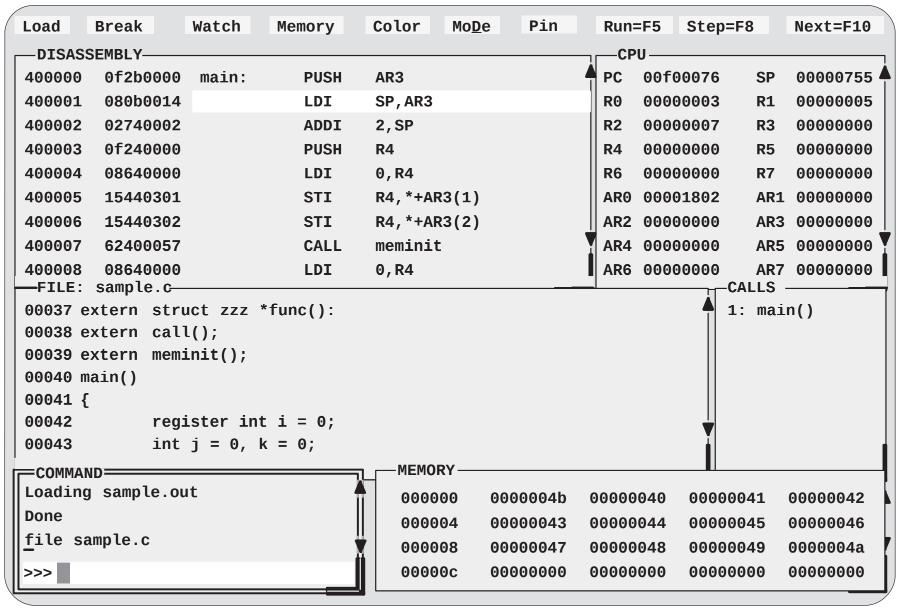
Assembly mode is for viewing assembly language programs only. In this mode, you'll see a display similar to the one shown in Figure 3–1. When you're in assembly mode, you'll always see the assembly display, regardless of whether C or assembly language is currently running.

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

Mixed mode

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

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



In mixed mode, the debugger displays all windows that can be displayed in auto and assembly modes—regardless of whether you're currently running assembly language or C code. This is useful for finding bugs in C programs that exploit specific architectural features of the 'C3x.

Restrictions associated with debugging modes

The assembly language code that the debugger shows you is the disassembly (reverse assembly) of the memory contents. If you load object code into memory, then the assembly language code is the disassembly of that object code. If you don't load an object file, then the disassembly won't be very useful.

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

dasm	func	mem
calls	file	disp

3.2 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.

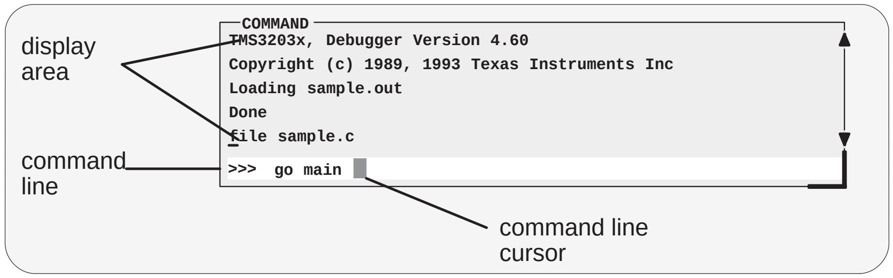
The name at the top of a window identifies the window's name. Each type of window serves a specific purpose and has unique characteristics. There are nine different windows, divided into four general categories:

- 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.
- **Code-display windows** are for displaying assembly language or C code. There are three code-display windows:
  - The DISASSEMBLY window displays the disassembly (assembly language version) of memory contents.
  - The FILE window displays any text file that you want to display; its main purpose, however, is to display C source code.
  - The CALLS window identifies the current function traceback (when C code is running).
- The **PROFILE window** displays statistics about code execution. This window is available only when you are in the profiling environment.
- **Data-display windows** are for observing and modifying various types of data. There are four data-display windows:
  - A MEMORY window displays the contents of a range of memory. You can display up to four MEMORY windows at one time.
  - The CPU window displays the contents of 'C3x registers.
  - A DISP window displays the contents of an aggregate type such as an array or structure, showing the values of the individual members. You can display up to 120 DISP windows at one time.
  - The 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 3.4, *The Active Window*, on page 3-19.

The remainder of this section describes the individual windows.

COMMAND window



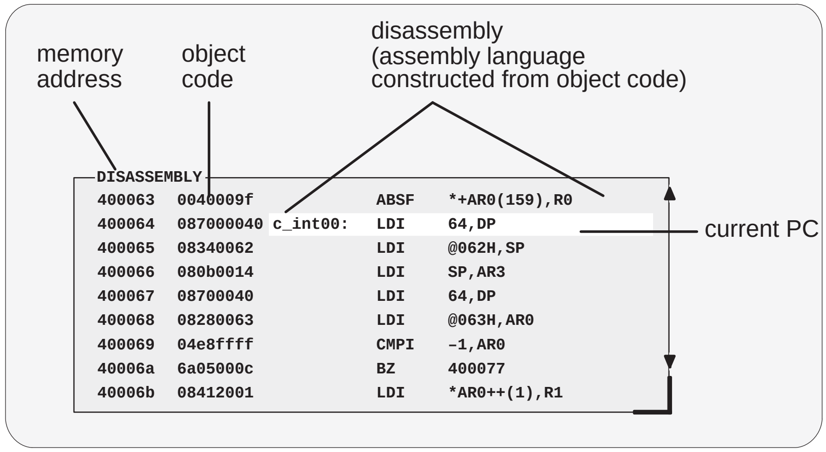
Purpose	❑ Provides an area for entering commands ❑ Provides an area for echoing commands and displaying command output, errors, and messages
Editable?	Command line is editable; command output isn't
Modes	All modes
Created	Automatically
Affected by	❑ All commands entered on the command line ❑ All commands 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 a list 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 4-5.)
- **Display area**. This area of the COMMAND window echoes the command that you entered, shows any output from the command, and displays debugger messages.

For more information about the COMMAND window and entering commands, refer to Chapter4 , *Entering and Using Commands*.

DISASSEMBLY window



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

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

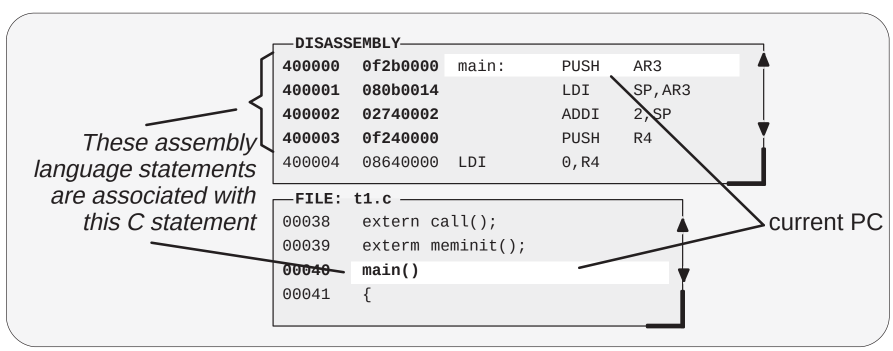
Modes Auto (assembly display only), assembly, and mixed

Created Automatically

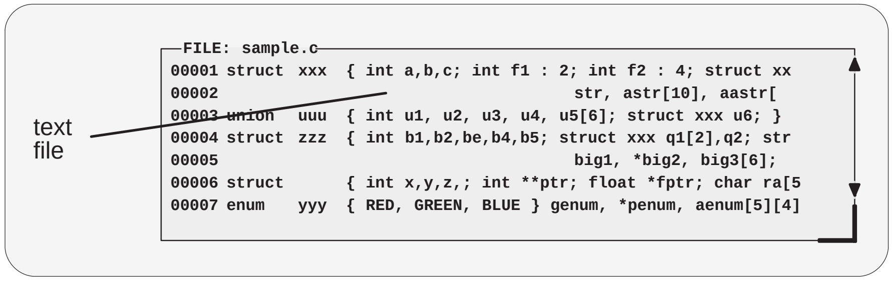
Affected by
- ❑ DASM and ADDR commands
- ❑ Breakpoint and run commands

Within the DISASSEMBLY window, the debugger highlights

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



FILE window



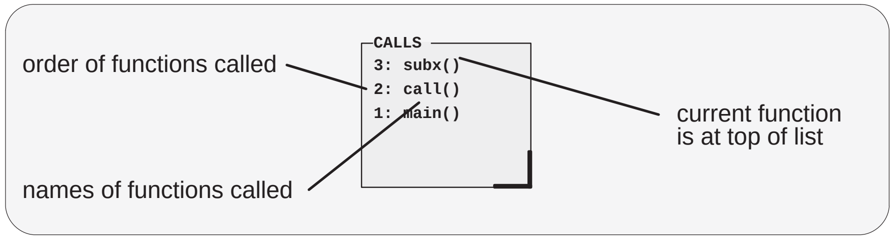
Purpose	Shows any text file you want to display
Editable?	No; if the FILE window displays C code, pressing the edit key (F9) or the left mouse button sets a software breakpoint on a C statement
Modes	Auto (C display only) and mixed
Created	❑ With FILE command ❑ Automatically when you’re in auto or mixed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

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

Within the FILE window, the debugger highlights:

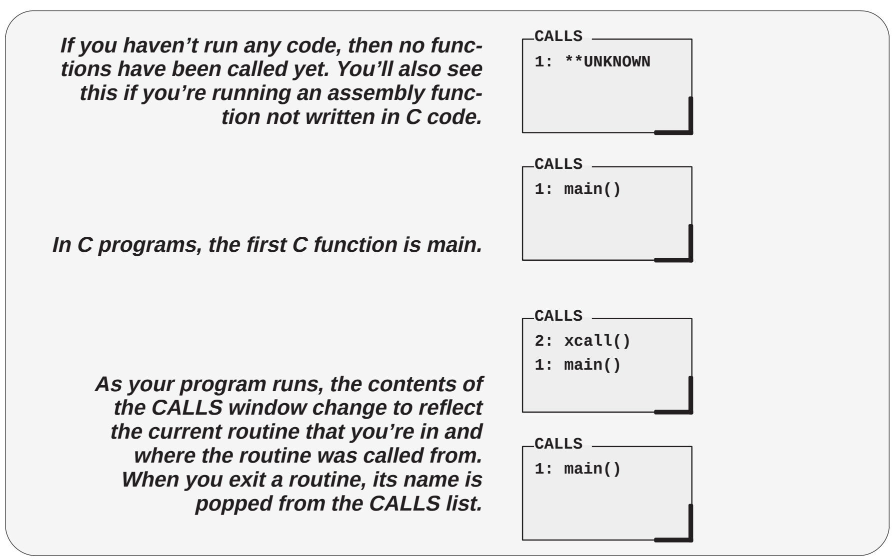
- The statement that the PC is pointing to (if that line is in the current display)
- Any statements where you’ve set a software breakpoint

CALLS window



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

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



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

1) Point the mouse cursor at the appropriate function name that is listed in the CALLS window.

2) Click the left mouse button. This displays the selected function in the FILE window.

1) Make the CALLS window the active window (see Section 3.4, *The Active Window,* page 3-19).

↓ ↑ 2) Use the arrow keys to move up/down through the list of function names until the appropriate function is indicated.

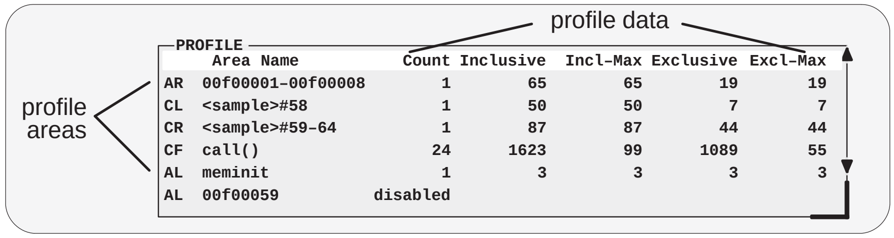
F9 3) Press F9 . This displays the selected function in the FILE window.

You can close and reopen the CALLS window.

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

  calls

PROFILE window



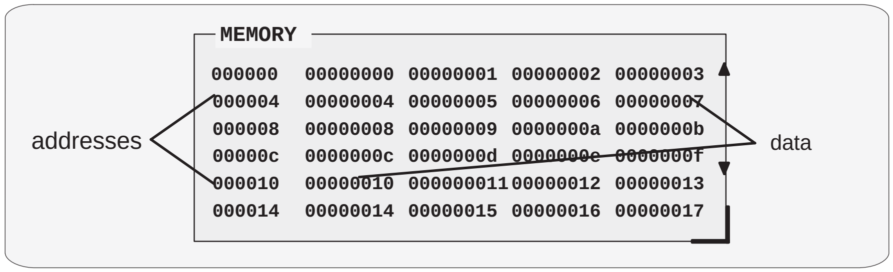
Purpose	Displays statistics collected during a profiling session
Editable?	No
Modes	Auto
Created	By invoking the debugger with the –profile option
Affected by	❑ The PF and PQ commands ❑ Any commands on the View menu ❑ Clicking in the header area of the window

The PROFILE window is visible only when you are in the profiling environment. The illustration above shows the window with a default set of data, but the display can be modified to show specific sets of data collected during a profiling session.

Note that within the profiling environment, the only other available windows are the COMMAND window, the DISASSEMBLY window, and the FILE window.

For more information about the PROFILE window (and about profiling in general), refer to Chapter 10, *Profiling Code Execution.*

MEMORY windows



Purpose	Displays the contents of memory
Editable?	Yes—you can edit the data (but not the addresses)
Modes	Auto (assembly display only), assembly, and mixed
Created	❑ Automatically (the default MEMORY window only) ❑ With the MEM# commands (up to three additional MEMORY windows)
Affected by	MEM commands: MEM, MEM1, MEM2, and MEM3

A 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.

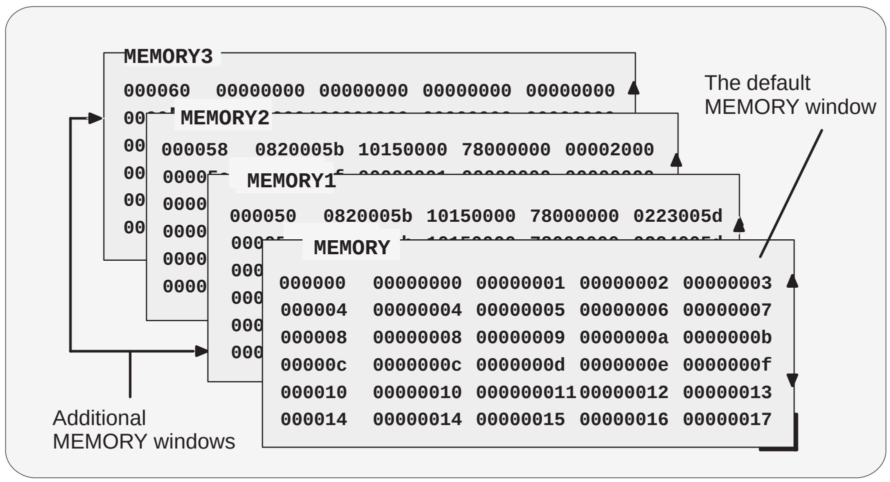
- ❑ **Data.** The remaining columns display the values at the listed addresses. You can display more data by making the window wider and/or longer.

  The MEMORY window above has four columns of data, so each new address is incremented by four. Although the window shows four columns of data, there is still only one column of addresses; the first value is at address 0x000000, the second at address 0x000001, etc.; the fifth value (first value in the second row) is at address 0x000004, the sixth at address 0x000005, 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).

Three additional MEMORY windows called MEMORY1, MEMORY2, and MEMORY3 are available. The default MEMORY window does not have an extension number in its name; this is because MEMORY1, MEMORY2, and MEMORY3 are optional windows and can be opened and closed throughout your debugging session. Having four windows allows you to view four different memory ranges. Refer to Figure 3–4.

Figure 3–4. The Default and Additional MEMORY Windows



To create an additional MEMORY window or to display another range of memory in the current window, use the MEM command.

- **Creating a new MEMORY window.**

  If the default MEMORY window is the only MEMORY 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 address 0x8000, you would enter:

  `mem1 0x8000`

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

- **Displaying a new memory range in the current MEMORY window.**

  Displaying another block of memory identifies a new starting address for the memory range shown 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:

  mem *address*

  To view different memory locations in the optional MEMORY windows, use the MEM command with the appropriate extension number on the end. For example:

To do this. . .	Enter this. . .
View the block of memory starting at address 0x0000 8000 in the MEMORY1 window	**mem1** `0x8000`
View another block of memory starting at address 0x0000 002f in the MEMORY2 window	**mem2** `0x002f`

Note:

If you want to view a different block of memory explicitly in the default MEMORY window, you can use the alias command MEM0. This works *exactly* the same as the MEM command. To use this command, enter:

mem0 *address*

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

- **Closing an additional MEMORY window.**

  Closing a window is a two-step process:

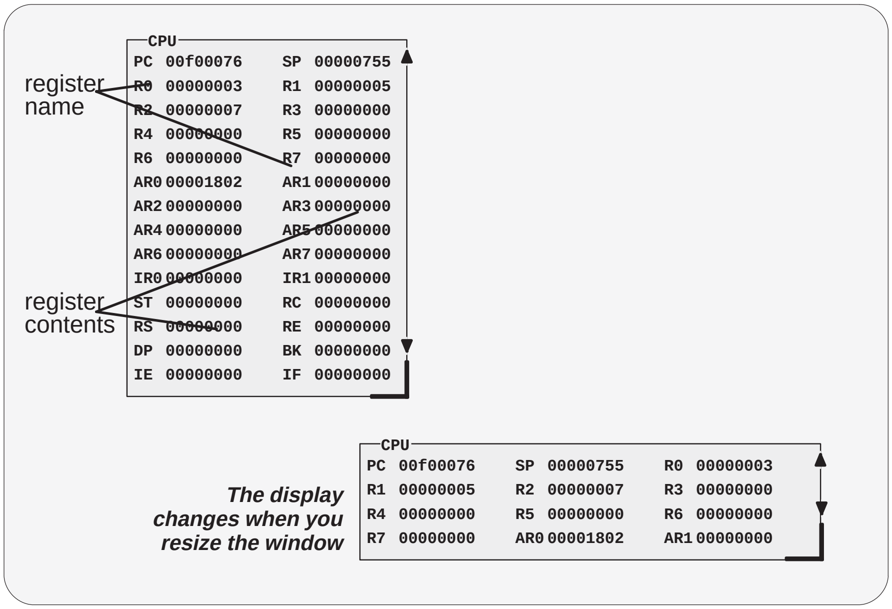
  1) Make the appropriate MEMORY window the active window (see Section 3.4, on page 3-19).
  2) Press F4 .

  Remember, you cannot close the default MEMORY window.

- **Reopening an additional MEMORY window.**

  To reopen an additional MEMORY window after you've closed it, enter the MEM command with its appropriate extension number.

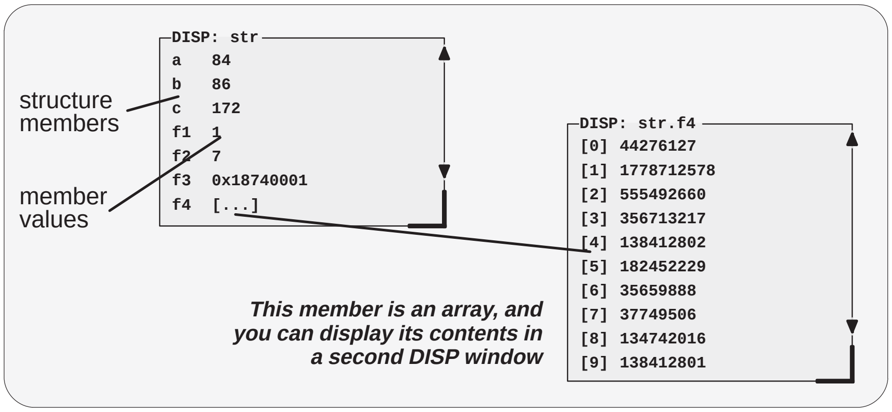
CPU window



Purpose	Displays the contents of the ’C3x registers
Editable?	Yes—you can edit the value of any displayed register
Modes	Auto (assembly display only), assembly, and mixed
Created	Automatically
Affected by	Data-management commands

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

DISP windows



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

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

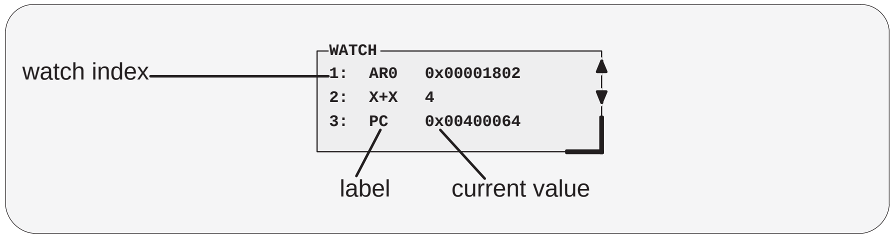
disp *expression*

By default, data is displayed in its natural format:

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

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

WATCH window



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

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

wa *expression* [, *label*]

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

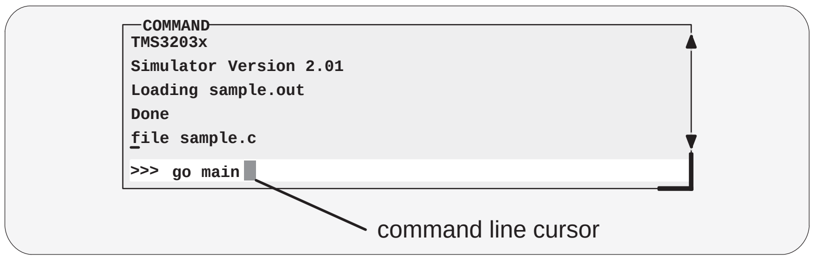
To delete individual entries from the WATCH window, use the WD command. To delete all entires at once and close the WATCH window, use the WR command.

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.

3.3 Cursors

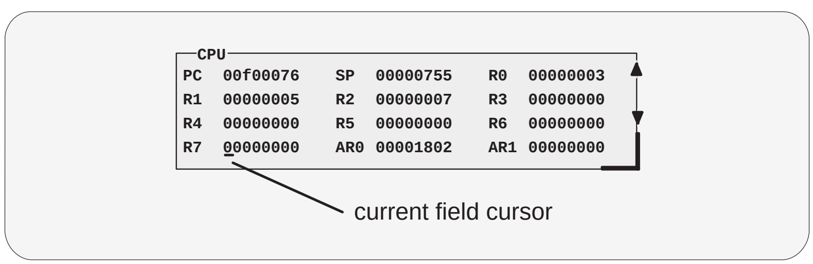
The debugger display has three types of cursors:

- 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. On PCs, this is the hardware cursor that is associated with your graphics card. Arrow keys *do* affect this cursor's movement.



3.4 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**.

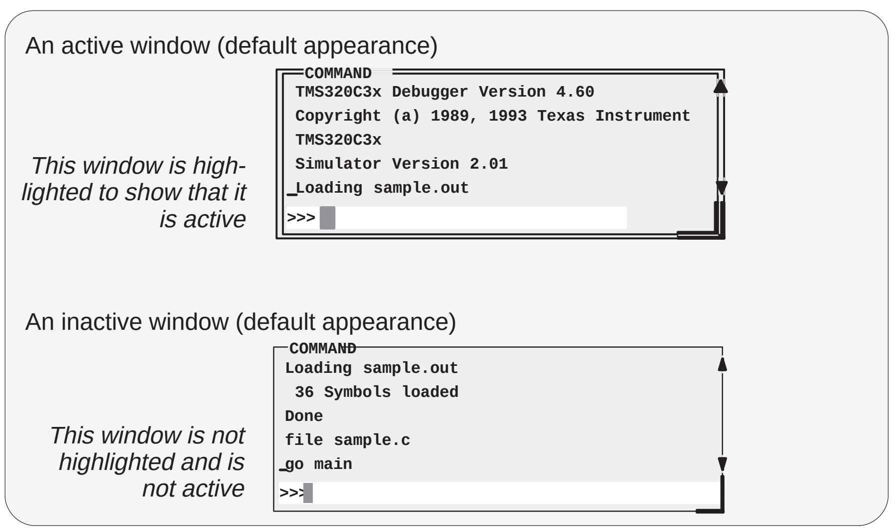
You can move, resize, zoom, or close *only one window at a time*; thus, only one window at a time can be the **active window**. Whether or not a window is active doesn't affect the debugger's ability to update information in a window—it affects only your ability to manipulate a window.

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 3–5 illustrates the default appearance of an active window and an inactive window.

Figure 3–5. 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 when the window becomes active).

Selecting the active window

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

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

2) Click the left mouse button.

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

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

  Press ESC *to get out of this.*

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

  Press the button again to clear the breakpoint.

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 to specify only enough letters to identify the window.

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

```
   win  DISASSEMBLY ⏎
or win  DISA ⏎
```

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

If you supply an ambiguous name (such as C, which could stand for CPU or CALLS), the debugger selects the first window it finds whose name matches the name you supplied. If the debugger doesn't find the window you asked for (because you closed the window or misspelled the name), then the WIN command has no effect.

3.5 Manipulating Windows

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

Note:

You can resize or move any window, but first the window must be **active**. For information about selecting the active window, refer to Section 3.4 (page 3-19).

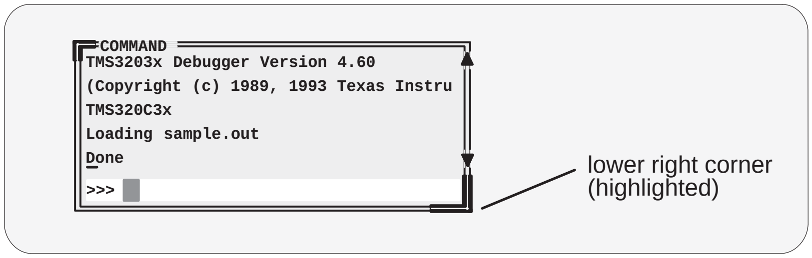
Resizing a window

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 the *width* and *length* parameters. Valid values for the width and length depend on the screen size and the window position on the screen. If the window is in the upper left corner of the screen, the maximum size of the window is the same as the screen size minus one line. (The extra line is needed for the menu bar.) For example, if the screen size is 80 characters by 25 lines, the largest window size is 80 characters by 24 lines.

If a window is in the middle of the display, 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 3-23.

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

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

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

↓	Makes the active window one line longer.
↑	Makes the active window one line shorter.
←	Makes the active window one character narrower.
→	Makes the active window one character wider.

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

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

win CPU ↵
size ↵
↓ ↓ ↓ ← ← ESC

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.

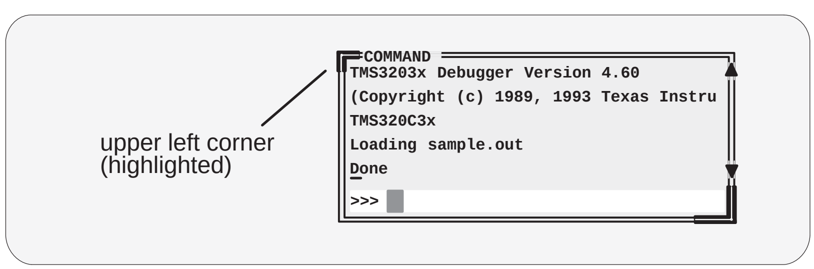
To "unzoom" a window, repeat the same steps you used to zoom it. This will return the window to its prezoom size and position.

There are two basic ways to zoom or unzoom 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

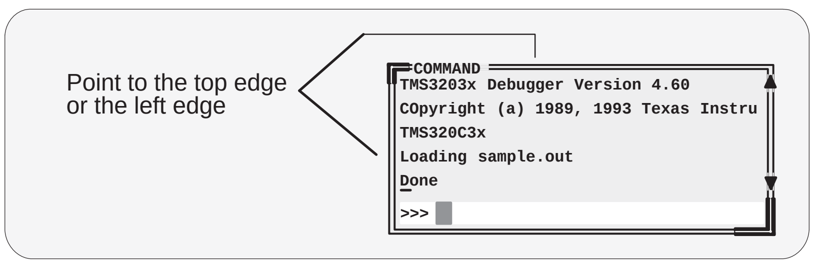
Moving a window

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. X positions are valid if the X position plus the window width in characters is less than or equal to the screen width in characters. Y positions are valid if the Y position plus the widow height is less than or equal to the screen height in lines.

For example, if the window is 10 characters wide and 5 lines high and the screen size is 80 x 25, the command **move 70, 20** would put the lower right-hand corner of the window in the lower right-hand corner of the screen. No X value greater than 70 or Y value greater than 20 would be valid in this example.

Note:

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.

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

↓	Moves the active window down one line.
↑	Moves the active window up one line.
←	Moves the active window left one character position.
→	Moves the active window right one character position.

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

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

```
win  COM ↵
move ↵
↑ ↑    → → → → →    ESC
```

3.6 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:**
>
> You can scroll and edit only the **active window**. For information about selecting the active window, refer to Section 3.4 (page 3-19).

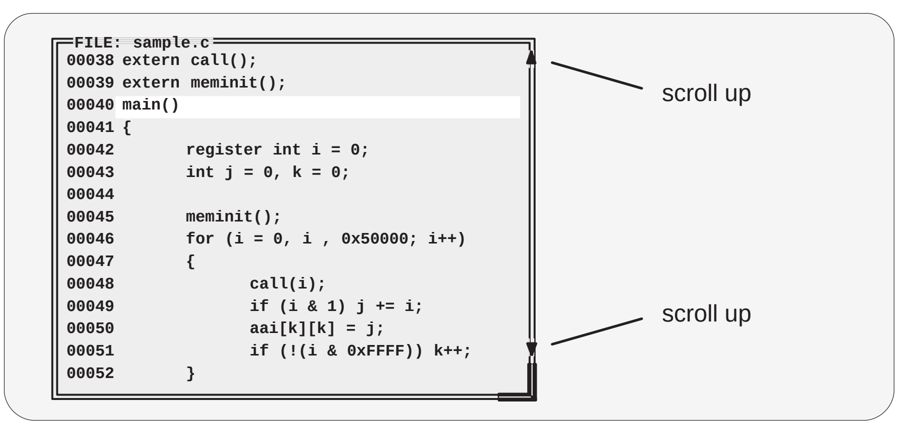
Scrolling through a window’s contents

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

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

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

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



To scroll window contents up or down:

1) Point to the appropriate scroll arrow.

2) Press the left mouse button; continue to press it until the information you're interested in is displayed within the window.

3) Release the mouse button when you're finished scrolling.

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

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

PAGE UP

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

PAGE DOWN

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

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

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

↑ — Pressing this key moves the field cursor up one line at a time.

↓ — Pressing this key moves the field cursor down one line at a time.

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

Editing the data displayed in windows

You can edit the data displayed in the MEMORY, CPU, DISP, and WATCH windows by using an overwrite "click and type" method or by using commands that change the values. (This is described in detail in Section 7.3, *Basic Methods for Changing Data Values,* page 7-4.)

Note:

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

- In the FILE and DISASSEMBLY windows, pressing F9 or the mouse button sets or clears a breakpoint on any line of code that you select. You can't modify text in a FILE or DISASSEMBLY window.
- In the CALLS window, pressing F9 or the mouse button shows the source for the function named on the selected line.
- In the PROFILE window, pressing F9 has no effect. Clicking the mouse button in the header displays a different set of data; clicking the mouse button on an area name shows the code associated with the area.

3.7 Closing a Window

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

Most of the windows remain open—you can't close them. However, you can close the CALLS, DISP, WATCH, and additional MEMORY windows. To close one of these windows:

1) Make the appropriate window active.

2) Press F4.

> **Note:**
>
> You cannot close the default MEMORY window.

You can also close the WATCH window by using the WR command:

wr ↵

When you close a window, the debugger remembers the window's size and position. The next time you open the window, it will have the same size and position. That is, if you close the CALLS window, then reopen it, it will have the same size and position as it did before you closed it. Since you can open numerous DISP and MEMORY windows, when you open one, it will occupy the same position as the last one of that type that you closed.

Chapter 4

Entering and Using Commands

The debugger provides you with several methods for entering commands:

- From the command line
- From the pulldown menus (using keyboard combinations or the mouse)
- With function keys
- From a batch file

Mouse use and function key use differ from situation to situation and are described throughout this book whenever applicable. This chapter includes specific rules that apply to entering commands and using pulldown menus. Also included is information about entering DOS commands and defining your own command strings.

Some restrictions apply to command entry for VAX and Sun versions of the simulator. For descriptions of these restrictions, refer to the installation guide.

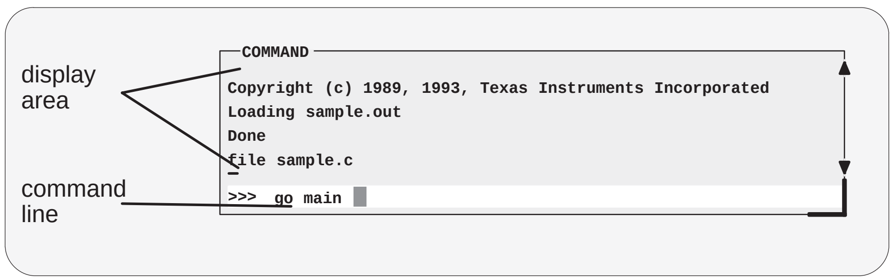
Topic **Page**

4.1 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 the various sections throughout this book, as they apply to the current topic. Chapter 11 summarizes all of the debugger commands with an alphabetic 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 4–1 shows the COMMAND window.

Figure 4–1. 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 4–1 shows that a GO command was typed in (but not yet entered).
- The **display area** provides the debugger with a space for echoing commands, displaying command output, or displaying errors and messages for you to read. For example, the command output in Figure 4–1 shows the messages that are displayed when you first bring up the debugger and also shows that a FILE command was entered.

  If you enter a command through 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	CONTROL H or BACK SPACE
Move forward through text without erasing characters	CONTROL L
Move back over text while erasing characters	DELETE
Move forward through text while erasing characters	SPACE
Insert text into the characters that are already on the command line	INSERT

Note:

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

Sometimes, you can't type a command

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

- When you're pressing the ALT key, typing certain letters causes the debugger to display a pulldown menu.
- When a pulldown menu is displayed, typing a letter causes the debugger to execute a selection from the menu.
- When you're pressing the CONTROL key, pressing H or L moves the command-line cursor backward or forward through the text on the command line.
- When you're editing a field, typing enters a new value in the field.
- When you're using the MOVE or SIZE command interactively, pressing keys affects the size or position of the active window. Before you can enter any more commands, you must press ESC to terminate the interactive moving or sizing.
- When you've brought up a dialog box, typing enters a parameter value at the current field in the box. Refer to Section 4.3 on page 4-11 for more information on dialog boxes.

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 that you entered. If you want to re-enter a command, you can move through this list, select a command that you've already executed, and re-execute it.

Use these keystrokes to move through the command history.

To...	Press...
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.

You can execute log files by using the TAKE command. When you use DLOG to record the information from the COMMAND window display area, 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 COMMAND window display area, 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**}]

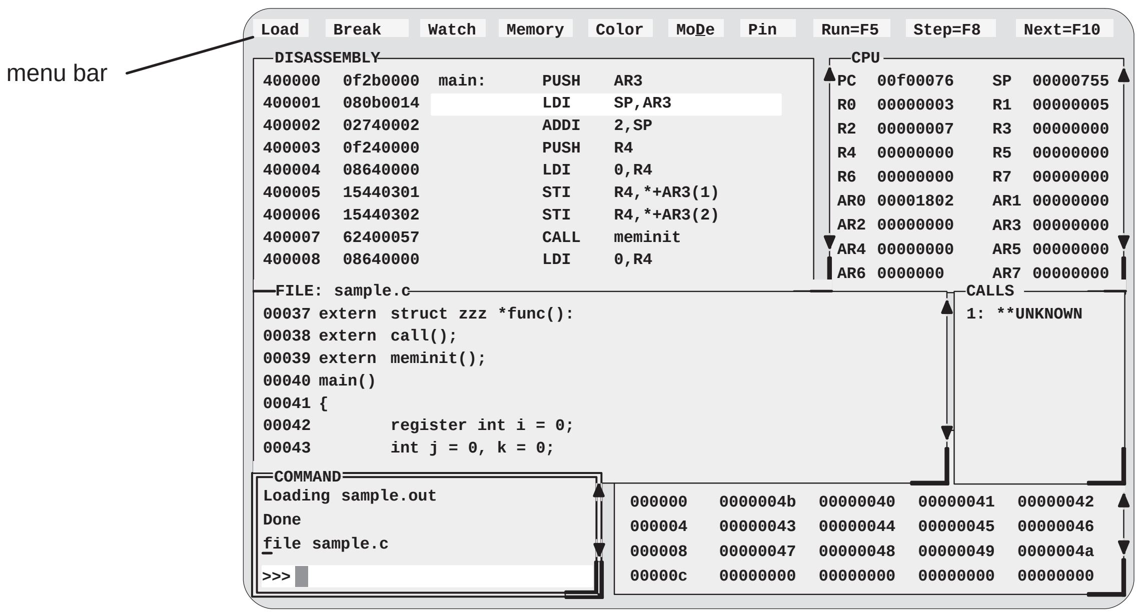
The optional parameters of the DLOG command control how the log file is created and/or used:

- **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.

4.2 Using the Menu Bar and the Pulldown Menus

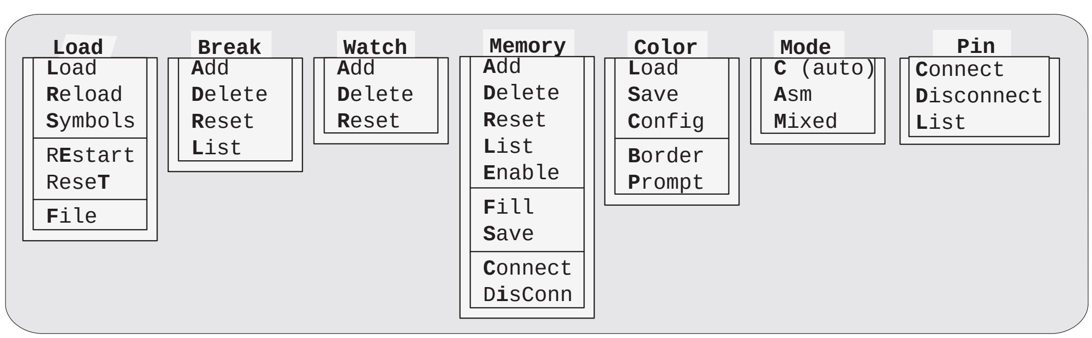
In all three of the debugger modes, you'll see a menu bar at the top of the screen. The menu selections offer you an alternative method for entering many of the debugger commands. Figure 4–2 points out the menu bar in a mixed-mode display. There are several ways to use the selections on the menu bar, depending on whether the selection has a pulldown menu or not.

Figure 4–2. The Menu Bar in the Basic 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 4–3.

Figure 4–3. All of the Pulldown Menus (Basic Debugger Display)



Note: The Connect and DisConn entries are available for the simulator only.

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.

Pulldown menus in the profiling environment

The debugger displays a different menu bar in the profiling environment:

```
Load   mAp   Mark  Enable  Disable  Unmark   View   Stop–points  Profile
```

The Load menu corresponds to the Load menu in the basic debugger environment. The mAp menu provides memory map commands available from the basic Memory menu. The other entries provide access to profiling commands.

Using the pulldown menus

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

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

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

Mouse method 1

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

Mouse method 2

1) Point the cursor at one of the appropriate selections in the menu bar.

2) Click the left mouse button. This displays the menu until you are ready to make a selection.

3) Point the mouse cursor at your selection on the pulldown menu.

4) When your selection is highlighted, click the left mouse button.

Keyboard method 1

ALT 1) Press the ALT key; don't release it.

X 2) Press the key that corresponds to the highlighted letter in the selection name; release both keys. This displays the menu and freezes it.

X 3) Press and release the key that corresponds to the highlighted letter of your selection in the menu.

Keyboard method 2

ALT 1) Press the ALT key; don't release it.

X 2) Press the key that corresponds to the highlighted letter in the selection name; release both keys. This displays the menu and freezes it.

↓ ↑ 3) Use the arrow keys to move up and down through the menu.

4) When your selection is highlighted, press .

Escaping from the pulldown menus

- If you display a menu and then decide that you don't want to make a selection from this menu, you can:
  - Press ESC

    or
  - Point the mouse outside of the menu; press and then release the left mouse button.
- If you pull down a menu and see that it is not the menu you wanted, you can point the mouse at another entry and press the left mouse button, or you can use the ← and → keys to display adjacent menus.

Using menu bar selections that don't have pulldown menus

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

```
Run=F5    Step=F8    Next=F10
```

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.

F5 Pressing this key is equivalent to typing in the RUN command without an *expression* parameter.

F8 Pressing this key is equivalent to typing in the STEP command without an *expression* parameter.

F10 Pressing this key is equivalent to typing in the NEXT command without an *expression* parameter.

4.3 Using Dialog Boxes

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

Entering text in a dialog box

Entering text in a dialog box is much like entering commands on the command line. For example, the Add entry on the Watch menu is equivalent to entering the WA command. This command has three parameters:

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

When you select Add from the Watch 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
                                        <<OK>>   <Cancel>
```

You can enter an *expression* just as you would if you were to type the WA command; then press TAB or ↓. The cursor moves down to the next parameter:

```
Watch Add
Expression  MY_VAR
Label
Format
                                        <<OK>>   <Cancel>
```

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.

In the case of the WA command, the 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 optional parameters, don't type anything in their fields—just continue to the next parameter.

Modifying text in a dialog box is similar to editing text on the command line:

- ❑ 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 TAB or ↓ to move to the next parameter.
- ❑ You can edit what you type (or values that remain from a previous entry) in the same way that you can edit text on the command line. See Section 4.1 for more information on editing text on the command line.

When you've entered a value for the final parameter, point and click on <OK> to save your changes, or <Cancel> to discard your changes; the debugger closes the dialog box and executes the command with the parameter values you supplied. You can also choose between the <OK> and <Cancel> options by using the arrow keys and pressing ↵ on your desired choice.

4.4 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 this 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 in the specified directory only.
  - ■ If you don't supply path information with the *filename*, the debugger looks for the file in the current directory.
  - ■ On PC systems, 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=*pathname;pathname*

This allows you to name several directories that the debugger can search. If you often use the same directories, it may be convenient to set D_DIR in your autoexec.bat file or initdb.bat file. On DOS systems, you can also set D_DIR from within the debugger by using the SYSTEM command (see Section 4.6, *Entering Operating-System Commands*, page 4-19).

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

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 execute debugger commands conditionally 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 4–1 shows the constants and their corresponding tools.

Table 4–1. Predefined Constants for Use With Conditional Commands

Constant	Debugger Tool
$$EMU$$	emulator
$$EVM$$	evaluation module
$$SIM$$	simulator

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.)

One way you can use these predefined constants is to create an initialization batch file that works for any debugger tool. This is useful if you are using, for example, both the emulator and the EVM. To do this, you can set up the following batch file:

```
if $$EMU$$
echo Invoking initialization batch file for emulator.
use \c3xhll
take emuinit.cmd
.
.
.
endif

if $$EVM$$
echo Invoking initialization batch file for EVM.
use \c3xhll
take evminit.cmd
.
.
.
endif
.
.
.
```

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

- 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
  runb
  .
  .
  .
  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 trace some register values continuously, 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 in a batch file only.
- ❑ 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.

4.5 Defining Your Own Command Strings

The debugger provides a shorthand method of entering often-used commands or command sequences. This process is called *aliasing*. Aliasing 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, displaying the same C source 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;file source.c;go main"
  ```

  Now you could enter init instead of the three 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.

  For example, suppose that every time you filled an area of memory you also wanted to display that block in the MEMORY window:

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

  Then you could enter:

  ```
  mfil 0x014,0x18,0x11112222
  ```

  The first value (0x014) would be substituted for the first FILL parameter and the MEM parameter (%1). The second and third values would be substituted for the second and third FILL parameters (%2 and %3).

- **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;file source.c;go main
 MFIL    -->  fill %1,%2,%3;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; file source.c;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.

- **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 delete 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:

- 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.

4.6 Entering Operating-System Commands (DOS Only)

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

system [*DOS command* [, *flag*]]

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

- If you enter the SYSTEM command with a DOS command as a parameter, then you stay within the debugger environment.
- If you enter the SYSTEM command without parameters, the debugger opens a *system shell*. This means that the debugger 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 DOS command, supply it as a parameter to the SYSTEM command. For example, if you want to copy a file from another directory into the current directory, you might enter:

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

If the DOS 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 DOS 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.c sample.c”,0
```

Entering several commands from a system shell

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

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

```
exit
```

> **Note:**
>
> Available memory limits the DOS commands that you can enter from a system shell. For example, you will 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 FILE, 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 5

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 by using the Memory pulldown menu.

5.1 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:

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) It checks to see whether 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 looks for the default initialization batch file called *init.cmd*. If the debugger finds this file, it reads and executes the commands.

Potential memory map problems

You may experience these problems 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 memory map. If you attempt to access an undefined or protected area, the debugger displays an error message.

- **Loading a COFF file with sections that cross a memory range.** Be sure that the map ranges you specify in a COFF file match those that you define with the MA command (described on page 5-7). Alternatively, you can turn memory mapping off during a load by using the MAP OFF command (see page 5-9).

5.2 Sample Memory Maps

Because you must define a memory map before you can run any programs, it's convenient to define the memory map in the initialization batch files. Figure 5–1 *(a),* Figure 5–2 *(a),* and Figure 5–3 *(a)* show the memory commands that are redefined in the initialization batch file that accompanies the simulator, emulator, and EVM, respectively. You can use any of the files as they are, edit them, or create your own memory map batch files.

The MA (map add) commands define valid memory ranges and identify the read/write characteristics of the memory ranges. The MAP command enables mapping (note that by default, mapping is enabled when you invoke the debugger). Figure 5–1 *(b),* Figure 5–2 *(b),* and Figure 5–3 *(b)* illustrate the memory map defined by the default initialization batch file.

Figure 5–1. Sample Memory Map for Use With a 'C3x Simulator

(a) Memory map commands (init.cmd)

```
MA 0x000000,0x1000,ROM
MA 0x808000,0x0010,RAM
MA 0x808020,0x0400,RAM
MA 0x809800,0x0800,RAM
MAP ON
```

(b) Memory map for 'C3x local memory

0x000000 to 0x000FFF	Internal ROM
0x001000 to 0x807FFF	reserved
0x808000 to 0x80800F	Memory Map Register 1
0x808010 to 0x80801F	reserved
0x808020 to 0x80841F	Memory Map Register 2
0x808420 to 0x8097FF	reserved
0x809800 to 0x809FFF	Internal RAM
0x80A000 to 0xFFFFFF	reserved

The 'C3x application board can be used as a target system with the 'C3x emulator. Figure 5–2 (page 5-5) shows a sample memory map for the application board.

Figure 5–2. Sample Memory Map for Use With a ’C3x Application Board / Emulator

(a) Memory map commands (init.cmd)

```
MA 0x000000,0x000800,ROM
MA 0x400000,0x080000,RAM
MA 0x800000,0x002000,RAM
MA 0x804000,0x001000,RAM
MA 0x805ff7,0x000009,RAM
MA 0x808000,0x000010,RAM
MA 0x808020,0x000010,RAM
MA 0x808030,0x000010,RAM
MA 0x808040,0x000010,RAM
MA 0x808050,0x000010,RAM
MA 0x808060,0x000001,RAM
MA 0x808064,0x000001,RAM
MA 0x809800,0x000400,RAM
MA 0x809C00,0x000400,RAM
MA 0xF00000,0x004000,RAM
MAP ON
```

(b) Memory map for ’C3x local memory

0x000000 to 0x0007FF	EPROM
0x000800 to 0x3FFFFF	reserved
0x400000 to 0x47FFFF	DRAM
0x480000 to 0x7FFFFF	reserved
0x800000 to 0x801FFF	SRAM1
0x802000 to 0x803FFF	reserved
0x804000 to 0x804FFF	DPRAM
0x805000 to 0x805FF6	reserved
0x805FF7 to 0x805FFF	DPSEM
0x806000 to 0x807FFF	reserved
0x808000 to 0x80800F	DMA
0x808010 to 0x80801F	reserved
0x808020 to 0x80802F	Timer 0
0x808030 to 0x80803F	Timer 1
0x808040 to 0x80804F	Serial Port 0
0x808050 to 0x80805F	Serial Port 1
0x808060	XBUSCTL
0x808061 to 0x808063	reserved
0x808064	PBUSCTL
0x808065 to 0x8097FF	reserved
0x809800 to 0x809BFF	’C3x Internal RAM Block 0
0x809C00 to 0x809FFF	’C3x Internal RAM Block 1
0x80A000 to 0xEFFFFF	reserved
0xF00000 to 0xF03FFF	SRAM0
0xF04000 to 0xFFFFFF	reserved

Figure 5–3. Sample Memory Map for Use With a ’C3x EVM

(a) Memory map commands (init.cmd)

```
MA 0x000000,0x004000,RAM
MA 0x804000,0x001000,RAM
MA 0x808000,0x000010,RAM
MA 0x808020,0x000010,RAM
MA 0x808030,0x000010,RAM
MA 0x808040,0x000010,RAM
MA 0x808050,0x000010,RAM
MA 0x808060,0x000001,RAM
MA 0x808064,0x000001,RAM
MA 0x809800,0x000400,RAM
MA 0x809C00,0x000400,RAM
MAP ON
```

(b) Memory map for ’C3x local memory

0x000000 to 0x003FFF	SRAM
0x004000 to 0x803FFF	reserved
0x804000 to 0x804FFF	Communication Port
0x805000 to 0x807FFF	reserved
0x808000 to 0x80800F	DMA
0x808010 to 0x80801F	reserved
0x808020 to 0x80802F	Timer 0
0x808030 to 0x80803F	Timer 1
0x808040 to 0x80804F	Serial Port 0
0x808050 to 0x80805F	Serial Port 1
0x808060	XBUSCTL
0x808061 to 0x808063	reserved
0x808064	PBUSCTL
0x808065 to 0x8097FF	reserved
0x809800 to 0x809BFF	’C3x Internal RAM Block 0
0x809C00 to 0x809FFF	’C3x Internal RAM Block 1
0x80A000 to 0xFFFFFF	reserved

5.3 Identifying Usable Memory Ranges

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

ma *address, length, type*

- The *address* parameter defines the starting address of a range. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.

  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 *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 *type* parameter
Read-only memory	**R** or **ROM**
Write-only memory	**W** or **WOM**
Read/write memory	**R|W** or **RAM**
No-access memory	**PROTECT**
Input port	**IPORT**
Output port	**OPORT**
Input/output port	**IOPORT**

Notes:

- The debugger caches memory that is not defined as a port type (IPORT, OPORT, or IOPORT). For ranges that you don't want cached, be sure to map them as ports.
- When you are using the simulator, you can use the parameter values IPORT, OPORT, and IOPORT to simulate I/O ports. See Section 5.9, *Simulating I/O Space.*
- Be sure that the map ranges that you specify in a COFF file match those that you define with the MA command. Moreover, a command sequence such as:

  ```
  ma x,y,ram; ma x+y,z,ram
  ```

  doesn't equal

  ```
  ma x,y+z,ram
  ```

  If you were planning to load a COFF block that spanned the length of y + z, you should use the second MA command example. Alternatively, you could turn memory mapping off during a load by using the MAP OFF command.

Memory mapping with the simulator

Unlike the emulator and EVM, the 'C3x simulator has memory cache capabilities that allow you to allocate as much memory as you need. However, to use memory cache capabilities effectively with the 'C3x, do not allocate more than 20K words of memory in your memory map. For example, the following memory map allocates 64K words of 'C3x program memory.

Example 5–1. Sample Memory Map for the TMS320C3x Using Memory Cache Capabilities

```
MA 0,0,R|W
MA 0x5000,0x5000,R|W
MA 0xa000,0x5000,R|W
MA 0xf000,0x5000,R|W
```

The simulator creates temporary files in a separate directory on your disk. For example, when you enter an MA (memory add) command, the simulator creates a temporary file in the root directory of your current disk. Therefore, if you are currently running your simulator on the C drive, temporary files are placed in the C:\ directory. This prevents the processor from running out of memory space while you are executing the simulator.

Note:

If you execute the simulator from a floppy drive (for example, drive A), the temporary files will be created in the A:\ directory.

All temporary files are deleted when you leave the simulator via the QUIT command. If, however, you exit the simulator with a soft reboot of your computer, the temporary files will not be deleted; you must delete these files manually. (Temporary files usually have numbers for names.)

Your memory map is restricted only by your PC's capabilities. As a result, there should be sufficient free space on your disk to run any memory map you want to use. If you use the MA command to allocate 20K words (80K bytes) of memory in your memory map, then your disk should have at least 80K bytes of free space available. To do this, you can enter:

```
ma 0x80a000, 0x5000, ram
```

5.4 Enabling Memory Mapping

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

map on
or **map off**

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

Note:

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
```

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

```
Memory     Range         Attributes
00000000 – 000007ff      READ
00400000 – 0047ffff      READ WRITE
00800000 – 00801fff      READ WRITE
00804000 – 00804fff      READ WRITE
00805ff7 – 00805fff      READ WRITE
00808000 – 0080800f      READ WRITE
00808020 – 0080802f      READ WRITE
00808030 – 0080803f      READ WRITE
00808040 – 0080804f      READ WRITE
00808050 – 0080805f      READ WRITE
00808060 –               READ WRITE
00808064 –               READ WRITE
00809800 – 00809bff      READ WRITE
```

starting address

ending address

5.6 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*

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

```
Specified map not found
```

Note:

If you are using the simulator and want to use the MD command to remove a simulated I/O port, you must first disconnect the port with the MI command. Refer to Section 5.9, page 5-13.

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, length, type*

The MA command is described in detail on page 5-7.

Returning to the original memory map

If you modify the memory map, you may want to go back to the original memory map without quitting and reinvoking the debugger. You can do this by resetting the memory map and then using the TAKE command to read in your original memory map from a batch file.

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

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

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

5.7 Using Multiple Memory Maps for Multiple Target Systems

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

Step 1: Let the initialization batch file define the memory map for one of your applications.

Step 2: Create a separate batch file that defines the memory map for the additional target system. The filename is unimportant, but for the purposes of this example, assume that the file is named *filename.x*. The general format of this file's contents should be:

`mr` ⏎	*Reset the memory map*
MA commands	*Define the new memory map*
`map on` ⏎	*Enable mapping*

(Of course, you can include any other appropriate commands in this batch file.)

Step 3: Invoke the debugger as usual.

Step 4: The debugger reads initialization batch file as usual. Before you begin debugging, read in the commands from the new batch file:

`take filename.x` ⏎

This redefines the memory map for the current debugging session.

You can also use the –t option instead of at the TAKE command when you invoke the debugger. The –t option allows you to specify a new batch file to be used instead of the default initialization batch file.

5.8 Simulating Serial Ports (Simulator Only)

The simulator supports serial port simulation with the global port control register, the FSX/DX/CLKX port control register, and the FSR/DR/CLKR port control register.

The simulator supports serial port I/O transfers on a limited basis. Because the simulator does not support any external signals, you can simulate serial port operations only by using the internal serial clocks. You must also enable the DR and DX pins as the serial receive pin and serial transmit pin, respectively.

To enable the internal clocks for both transmit and receive operations, you must ensure that the XCLKSRCE and RCLKSRCE bits of the global port control register are set to 1. To enable the DX and DR pins for serial transmit and receive, set both the DXFUNC bit (in the FSX port control register) and the DRFUNC bit (in the FSR port control register) to 1.

5.9 Simulating I/O Space (Simulator Only)

In addition to adding memory ranges to the memory map, you can use the MA command to add I/O ports to the memory map. To do this, use IPORT (input port), OPORT (output port), or IOPORT (input/output port) as the memory type. Then, you can use the MC command to connect a port to an input or output file. This simulates external I/O cycle reads and writes by allowing you to read data in from a file and/or write data out to a file.

Connecting an I/O port

mc The MC (memory connect) command connects IPORT, OPORT, or IOPORT to an input or output file. Before you can connect the port, you must add it to the memory map with the MA command. The syntax for this command is:

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

- The *port address* parameter defines the address of the I/O port. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.
- The *filename* parameter can be any filename. If you connect a port to read from a file, the file must exist, or the MC command will fail.
- The final parameter is specified as **READ** or **WRITE** and defines how the file will be used (for input or output, respectively).

The file is accessed as an LDI or STI instruction accesses the associated port address. Any port in I/O space can be connected to a file. A maximum of one input and one output file can be connected to a single port; multiple ports can be connected to a single file. Memory-mapped ports can also be connected to files; any instruction that reads or writes to the memory-mapped port will read or write to the associated file.

Note:

When using the MS-DOS version of the simulator, you can connect a maximum of 15 ports.

Example 5–2 shows how an input port can be connected to an input file named in.dat.

Example 5–2. Connecting an Input Port to an Input File

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

```
0x0A000000
0x10000000
0x20000000
  .
  .
  .
```

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

```
MA    0x50,0x1,IPORT        Configure port address 50h
                            as an input port
MC    0x50,in.dat,READ      Open file in.dat and
                            connect to port address 50h
```

Assume that this ’C3x instruction is part of your ’C3x program. This reads the data from the file in.dat.

```
LDI   @50h,R0               LDI instruction reads from the file
```

Configuring memory to use serial port simulation

In order to use the serial port simulation, you must configure memory with the MA and MC commands. The following example adds the transmit and receive registers to the memory map and then connects their input and output to a file:

Example 5–3. Adding Serial Port 0 Transmit and Receive Registers; Connecting Their Input and Output to a File

```
ma 0x808020,0x27,RAM    ;Configure all control registers
ma 0x808048,0x1,OPORT   ;Configure DTR as output port
ma 0x80804C,0x1,IPORT   ;Configure DRR as input port
ma 0x808050,0x350,RAM   ;Configure other MMR registers
mc 0x808048,xdat,WRITE  ;Open file xdat and connect to port address
                        ;0x808048h
mc 0x80804C,rdat,READ   ;Open file rdat and connect to port address
                        ;0x80804C
```

The following commands configure the global port control, FSX/DX/CLKX port control register, and FSR/DR/CLKR port control register of serial port 0 for a 8-bit transmit and receive operations:

```
?*0x808040=0x000000C0
?*0x808042=0x00000010
?*0x808043=0x00000010
```

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

```
0x00000000
0xA4450000
0x099F0000
  .
  .
  .
```

Disconnecting an I/O port

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

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

mi *port address,* {**READ** | **WRITE**}

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

5.10 Simulating External Interrupts (SImulator Only)

The ’C3x simulator allows you to simulate and monitor external interrupt signals and to specify at what clock cycle you want an interrupt to occur. To do this, you create a data file and connect it to one of the four interrupt pins, $\overline{\text{INT}}0$–$\overline{\text{INT}}3$.

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

Setting up your input file

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

clock cycle... [*(clock cycle...)* **rpt** {*n* **|** **EOS**}]

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

  You can have two types of CPU clock cycles:

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

    ```
    12 34 56
    ```

    Interrupts are simulated at the 12th, 34th, and 56th CPU clock cycles. Notice that no operation is done to the clock cycle value; the interrupt occurs exactly as the clock cycle value is written.

  - **Relative**. You can also select a clock cycle that is relative to the time at which the last event occurred. For example:

    ```
    12 +34 55
    ```

    In this example, a total of three interrupts are simulated at the 12th, 46th (12+34), and 55th CPU clock cycles. A plus sign (+) before a clock cycle adds that value to the total clock cycles preceding it. Notice that you can mix both relative and absolute values in your input file.

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

  You can have two forms of repetition to simulate interrupts:

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

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

    The values inside of the parenthesis represent the portion that is repeated. Therefore, an interrupt is simulated at the 5th CPU cycle, then the15th (5+10), 35th (15+20), 45th (35+10), and 65th (45+20) CPU clock cycles.

    Note that n is a positive integer value.

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

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

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

Programming the simulator

After you have created your input file, you can use debugger commands to:

- Connect the interrupt pin to your input file
- List the interrupt pins
- Disconnect the interrupt pin from your input file

Use these commands as described below, or use them from the PIN pulldown menu.

---

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

pinc *pinname, filename*

- The *pinname* parameter identifies the pin and must be one of the four external interrupt pins ($\overline{\text{INT}}$0–$\overline{\text{INT}}$3).
- The *filename* parameter is the name of your input file.

Example 5–4 shows you how to connect your input file by using the PINC command.

Example 5–4. Connecting the Input File With the PINC Command

Suppose you want to simulate external interrupts at the 12th, 34th, 56th, and 89th clock cycles.

First, create a input file with an arbitrary name such as myfile that contains the following line:

```
12 34 56 89
```

Then use the PINC command in the pin pulldown menu to connect the input file to the $\overline{\text{INT}}$2 pin.

```
pinc  myfile, int2
```

Connects your data file to the specific interrupt pin

This command connects myfile to the $\overline{\text{INT}}$2 pin. As a result, the debugger simulates an $\overline{\text{INT}}$2 external interrupt at the 12th, 34th, 56th, and 89th clock cycles.

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

pinl

The PINL command displays all of the unconnected pins first, followed by the connected pins. For a connected pin, the simulator displays the name of the pin and the absolute pathname of the file in the COMMAND window.

```
COMMAND
PIN                  FILENAME
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 INT0                 NULL
 INT1                 NULL
 INT3                 NULL
 INT2                 /320hll/myfile
_
>>>
```

When you want to connect another file to an interrupt pin, the PINL command is useful for looking up an unconnected pin.

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

pind *pinname*

The *pinname* parameter identifies the interrupt pin and must be one of the four interrupt pins ($\overline{\text{INT}}0$–$\overline{\text{INT}}3$). The PIND command detaches the file from the interrupt pin. After executing this command, you can connect another file to the same pin.

Chapter 6

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.

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

The debugger has three code-display windows:

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

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

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

Here's a summary of the modes and displays; for a complete description of the three debugging modes, refer to Section 3.1, *Debugging Modes and Default Displays* (page 3-2).

Use this mode	To view	The debugger uses these code-display windows
assembly mode	*assembly language code only* (even if your program is executing C code)	❑ DISASSEMBLY
auto mode	*assembly language code* (when that's what your program is running)	❑ DISASSEMBLY
auto mode	*C code only* (when that's what your program is running)	❑ FILE ❑ CALLS
mixed mode	*both assembly language and C code*	❑ DISASSEMBLY ❑ FILE ❑ CALLS

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

Selecting a debugging mode

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

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

```
MoDe
 C  (auto)
 Asm
 Mixed
```

1) Point to the menu name.

2) Press the left mouse button; do not release the button. Move the mouse down the menu until your choice is highlighted.

3) Release the mouse button.

For more information about the pulldown menus, refer to Section 4.2, *Using the Pulldown Menus,* on page 4-7.

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

auto → assembly → mixed → (back to auto)

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

c Changes from the current mode to auto mode.

asm Changes from the current mode to assembly mode.

mix Changes from the current mode to mixed mode.

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

6.2 Displaying Your Source Programs (or Other Text Files)

The debugger displays two types of code:

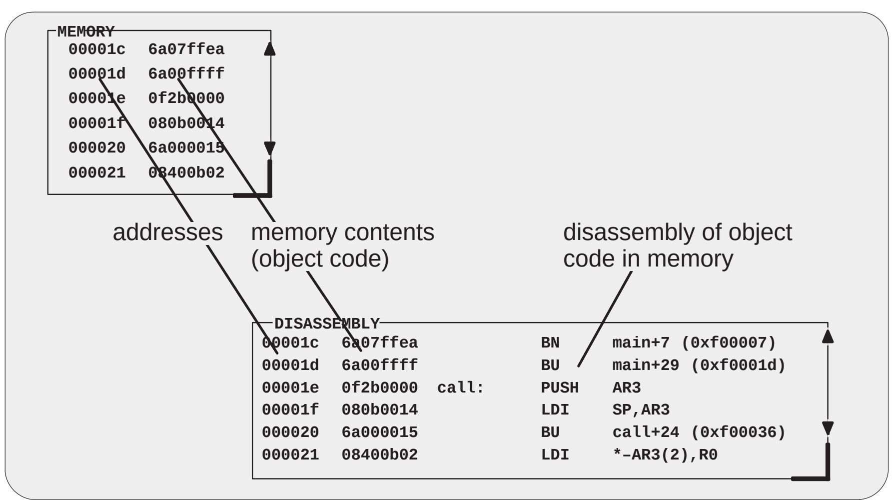
- It displays **assembly language code** in the DISASSEMBLY window in auto, assembly, or mixed mode.
- It displays **C code** in the FILE window in auto and mixed modes.

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

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

Displaying assembly language code

The assembly language code in the DISASSEMBLY window is the reverse assembly of memory contents. (This code doesn't come from any of your text files or from the intermediate assembly files produced by the compiler.)



When you invoke the debugger, it comes up in auto mode. If you load an object file when you invoke the debugger, then the DISASSEMBLY window displays the reverse assembly of the object file that's loaded into memory. If you don't load an object file, the DISASSEMBLY window shows the reverse assembly of whatever happens to be in memory.

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

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

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

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

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

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

In assembly mode, ADDR works like the DASM command, positioning the code starting at *address* or at *function name* as the first line of code in the DISASSEMBLY window. In mixed mode, ADDR affects both the DISASSEMBLY and FILE windows.

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*

For patch assembly, use the **right** mouse button instead of the left. (Clicking the left mouse button sets a software 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 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.

CAUTION

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, recompile or 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 Floating-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 Floating-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 Floating-Point DSP Assembly Language Tools User's Guide* for a detailed list of these messages.

Displaying C code

Unlike assembly language code, C code isn't reconstructed from memory contents—the C code that you view is your original C source. You can display C code explicitly or implicitly:

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

These commands are valid in C and mixed modes.

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

file *filename*

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

(Note that displaying a file *doesn't* load that file's object code. If you want to be able to run the program, you must load the file's associated object code as described in Section 6.3 on page 6-10.)

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

func *function name*
or **func** *address*

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

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

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

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

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

Whenever the CALLS window is open, you can use the mouse or function keys to display a specific C function. This is similar to the FUNC or ADDR command but applies only to the functions listed in the CALLS window.

1) In the CALLS window, point to the name of C function.

2) Click the left mouse button.

(If the CALLS window is active, you can also use the arrow keys and F9 to display the function; see the *CALLS window* discussion on page 3-9 for details.)

Displaying other text files

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

The FILE window is primarily for displaying C code, but you can use the FILE command to display any text file within the FILE window. You may, for example, wish to examine system files such as autoexec.bat or an initialization batch file. You can also view your original assembly language source files in the FILE window.

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

6.3 Loading Object Code

In order to debug a program, you must load the program's object code into memory. You can do this as you're invoking the debugger, or you can do it after you've invoked the debugger. (Note that you create an object file by compiling, assembling, and linking your source files; see Section 1.4, *Preparing Your Program for Debugging*, on page 1-10.)

Loading code while invoking the debugger

You can load an object file when you invoke the debugger (this has the same effect as using the debugger's LOAD command). To do this, enter the appropriate debugger-invocation command along with the name of the object file.

If you want to load a file's symbol table only, use the –s option (this has the same effect as using the debugger's SLOAD command). To do this, enter the appropriate debugger-invocation command along with 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*

If you don't supply an extension, the debugger will look for *filename.out.*

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*]

If you enter the RELOAD command without specifying a filename, the debugger reloads the file that you loaded last.

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.

6.4 Where the Debugger Looks for Source Files

Some commands (FILE, LOAD, RELOAD, and SLOAD) expect a filename as a parameter. If the filename includes path information, the debugger uses the file from the specified directory and does not search for the file in any other directory. If you don't supply path information, though, the debugger must search for the file. The debugger first looks for these files in the current directory. You may, however, have your files in several different directories.

- If you're using LOAD, RELOAD, or SLOAD, you have only two choices for supplying the path information:
  - Specify the path as part of the filename.
  - **cd** Alternatively, you can use the CD command to change the current directory from within the debugger. The format for this command is:

    cd *directory name*
- If you're using the FILE command, you have several options:
  - Within the DOS environment, you can name additional directories with the D_SRC environment variable. The format for doing this is:

    SET D_SRC=*pathname;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 or initdb.bat file. If you do this, then the list of directories is always available when you're using the debugger.
  - When you invoke the debugger, you can use the – i option to name additional source directories for the debugger to search. The format for this option is **–i** *pathname.*

    You can specify multiple pathnames by using several –i options (one pathname per option). The list of source directories that you create with –i options is valid until you quit the debugger.
  - **use** Within the debugger environment, you can use the USE command to name additional source directories. The format for this command is:

    use *directory name*

    You can specify only one directory at a time.

In all cases, you can use relative pathnames such as ..\csource or ..\..\code. The debugger can recognize a cumulative total of 20 paths specified with D_SRC, –i, and USE.

6.5 Running Your Programs

To debug your programs, you must execute them on one of the three 'C3x debugging tools (emulator, evaluation module, or simulator). The debugger provides two basic types of commands to help you run your code:

- Basic **run commands** run your code on the target system without updating the display until you explicitly halt execution. There are several ways to halt execution:
  - Set a breakpoint.
  - When you issue a run command, define a specific stopping point.
  - Press ESC.
  - Press the left mouse button.
- **Single-step** commands execute assembly language or C code, one statement at time, and update the display after each execution.

Defining the starting point for program execution

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

- Finding its entry in the CPU window

  or
- Finding the appropriately highlighted line in the FILE or DISASSEMBLY window. To do this, execute one of these commands:

  dasm PC

  or **addr PC**

Sometimes you may want to modify the PC to point to a different position in your program. 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 6-17).
- 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.

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

go [*address*]

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

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

return
or **ret**

Breakpoints do not affect this command, but you can halt execution by pressing ESC or the left mouse button.

runb Use the RUNB (run benchmark) command to execute a specific section of code and count the number of clock cycles consumed by the execution. The format for this command is:

runb

Using the RUNB command to benchmark code is a multistep process, described in Section 6.7, *Benchmarking*, on page 6-19.

key

F5 Pressing this key runs code from the current PC. This is similar to entering 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.) You can single-step through assembly language code or C code.

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

> **Note:**
>
> The single-stepping debugger commands (STEP, CSTEP, and NEXT) turn off the global interrupt bit GIE and prevent stepping through an interrupt service routine. If you want to step into an interrupt service routine, set a breakpoint in the interrupt service routine and use one of the run commands.

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 6-17).
- ❑ If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger single-steps *count* C or assembly language statements (depending on the type of code you're in).

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

step [*expression*]

If you're in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time.

If you're single-stepping through C code and encounter a function call, the STEP command shows you the single-step execution of the called function (assuming that the function was compiled with the compiler's –g debug option). When function execution completes, single-step execution returns to the caller. If the function wasn't compiled with the debug option, the debugger executes the function but doesn't show single-step execution of the function.

cstep The CSTEP command is similar to STEP, but CSTEP always single-steps in terms of a C statement. If you're in C code, STEP and CSTEP behave identically. In assembly language code, however, CSTEP executes all assembly language statements associated with one C statement before updating the display. The format for this command is:

cstep [*expression*]

next
cnext The NEXT and CNEXT commands are similar to the STEP and CSTEP commands. The only difference is that NEXT/CNEXT never show single-step execution of called functions—they always step to the next consecutive statement. The formats for these commands are:

next [*expression*]
cnext [*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 disconnected from the target

runf Use the RUNF command to disconnect the emulator or EVM from the target system while code is executing. The format for this command is:

runf

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

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

EVM & emulator

halt Use the HALT command to halt the target system after you've entered a RUNF command. The format for this command is:

halt

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

reset The RESET command resets the target system. This is a *software* reset. The format for this command is:

reset

If you are using the simulator and execute the RESET command, the simulator simulates the 'C3x processor and peripheral reset operation, putting the processor in a known state.

Running code conditionally

The RUN, STEP, CSTEP, NEXT, and CNEXT commands all have an optional *expression* parameter that can be a relational or logical expression. This type of expression 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 software breakpoints with conditional runs; the expression is evaluated each time the debugger encounters a breakpoint. Each time the debugger evaluates the conditional expression, it updates the screen. The debugger applies this algorithm:

top:
    if (*expression* = = 0), go to end;
    run or single-step (until breakpoint, ESC, or mouse button halts execution)
    if (halted by breakpoint, *not* by ESC or mouse button), go to top
end:

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

6.6 Halting Program Execution

Whenever you're running or single-stepping code, program execution halts automatically if the debugger encounters a breakpoint or if it reaches a particular point where you told it to stop (by supplying a *count* or an *address*). If you'd like to explicitly halt program execution, there are two ways to accomplish this:

Click the left mouse button.

ESC 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.

6.7 Benchmarking

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

Benchmarking code is a multiple-step process:

Step 1: Set a software breakpoint at the statement that marks the beginning of the section of code you'd like to benchmark.

Step 2: Set a software breakpoint at the statement that marks the end of the section of code you'd like to benchmark.

Step 3: Enter any RUN command to execute code up to the first breakpoint.

Step 4: Now enter the RUNB command:

```
runb
```

When the processor halts at the second breakpoint, the value of CLK is valid. To display it, use the ? command or enter it into the WATCH window with the WA command. This value is valid until you enter another RUN command.

Note:

- The RUNB command counts CPU clock cycles from the current PC to the breakpoint. This count is not cumulative. You cannot add the number of clock cycles from point A to point B to the number of cycles from point B to point C to learn the number of cycles from point A to point C. This error occurs because of pipeline filling and flushing.
- The value in CLK is valid only after using a RUNB command that is terminated by a software breakpoint.

Chapter 7

Managing Data

The debugger allows you to examine and modify many different types of data related to the 'C3x and to your program. You can display and modify the values of:

- Individual memory locations or a range of memory
- 'C3x registers
- Variables, including scalar types (ints, chars, etc.) and aggregate types (arrays, structures, etc.)

Topic **Page**

7.1 Where Data Is Displayed

Four windows are dedicated to displaying the various types of data.

Type of data	Window name and purpose
memory locations	**MEMORY windows** Display the contents of a range of memory
register values	**CPU window** Displays the contents of ’C3x registers
pointer data or selected variables of an aggregate type	**DISP windows** Display the contents of aggregate types and show the values of individual members
selected variables (scalar types or individual members of aggregate types) and specific memory locations or registers	**WATCH window** Displays selected data

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

7.2 Basic Commands for Managing Data

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

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

whatis *symbol*

This lists *symbol*’s data type in the COMMAND window display area. The *symbol* can be any variable (local, global, or static), a function name, structure tag, typedef name, or enumeration constant.

Command	Result displayed in the COMMAND window
`whatis giant`	`struct zzz giant[100];`

`whatis xxx`	`struct xxx {`
	`int a;`
	`int b;`
	`int c;`
	`int f1 : 2;`
	`int f2 : 4;`
	`struct xxx *f3;`
	`int f4[10];`
	`}`

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

? *expression*

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

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

Here are some examples that use the ? command:

Command	Result displayed in the COMMAND window
? giant	giant[0].b1 436547877 giant[0].b2 –791051538 giant[0].b3 1952557575 giant[0].b4 –1555212096 etc.
? j	4194425
? j=0x5a	90
? i	–12635
? i,x	0x000cea5

Note that the DISP command (described in detail on page 7-12) behaves like the ? command when its *expression* parameter does not identify an aggregate type.

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

eval *expression*
or **e** *expression*

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

7.3 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 a MEMORY window
- Elements displayed in a DISP window
- Values displayed in the WATCH window

There are two similar methods for overwriting displayed data:

This method is sometimes referred to as the "click and type" method.

1) Point to the data item that you want to modify.

2) Click the left button. The debugger highlights the selected field. (Note that the window containing this field becomes active when you press the mouse button.)

(ESC) 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 3.4, *The Active Window,* on page 3-19.)

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.

F9 3) When the field you'd like to edit is highlighted, press F9. The debugger highlights the field that the cursor is pointing to.

ESC 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 register R3, you can enter:

? R3 ↵

However, you can also use this type of command to modify R3's contents. Here are some examples of how you might do this:

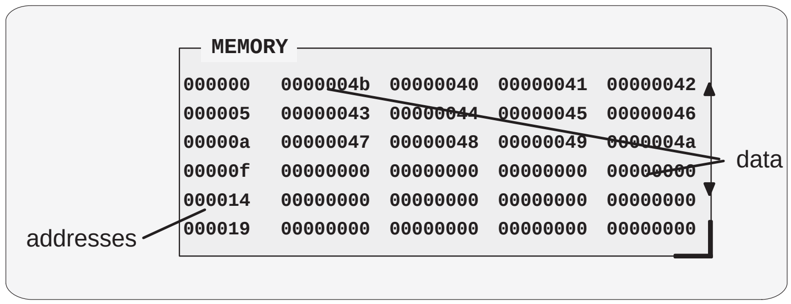
? R3++ ↵	*Side effect: increments the contents of R3 by 1*
eval --R3 ↵	*Side effect: decrements the contents of R3 by 1*
? R3 = 8 ↵	*Side effect: sets R3 to 8*
eval R3/=2 ↵	*Side effect: divides contents of R3 by 2*

Note that not all expressions have side effects. For example, if you enter **? R3+4**, the debugger displays the result of adding 4 to the contents of R3 but does not modify R3's contents. Expressions that have side effects must contain an assignment operator or an operator that implies an assignment. Operators that can cause a side effect are:

=	+=	–=	*=	/=
%=	&=	^=	|=	<<=
	>>=	++	– –	

7.4 Managing Data in Memory

In mixed and assembly modes, the debugger maintains a MEMORY window that displays the contents of memory. For details concerning the MEMORY window, see the *MEMORY windows* discussion (page 3-12).



The debugger has commands that show the data values at a specific location or that display a different range of memory in the MEMORY window. The debugger allows you to change the values at individual locations; refer to Section 7.3 (page 7-4), for more information.

Displaying memory contents

The main way to observe memory contents is to view the display in a MEMORY window. Four MEMORY windows are available: the default window is labeled MEMORY, and the three additional windows are called MEMORY1, MEMORY2, and MEMORY3. Notice that the default window does not have an extension number in its name; this is because MEMORY1, MEMORY2, and MEMORY3 are pop-up windows that can be opened and closed throughout your debugging session. Having four windows allows you to view four different memory ranges.

The amount of memory that you can display is limited by the size of the individual MEMORY windows (which is limited only by the screen size). During a debugging session, you may need to display different areas of memory within a window. You can do this by typing a command or using the mouse.

mem If you want to display a different memory range in the MEMORY window, use the MEM command. You can do this by entering:

mem *expression*

To view different memory locations in an additional MEMORY window, use the MEM command with the appropriate extension number. For example:

To do this. . .	Enter this. . .
View the block of memory starting at address 0x8000 in the MEMORY1 window	**`mem1`** `0x8000`
View the same block of memory (starting at address 0x8000) but in the MEMORY2 window	**`mem2`** `0x8000`

Note:

If you want to view a different block of memory explicitly in the default MEMORY window, you can use the aliased command MEM0. This works *exactly* the same as the MEM command. To use this command, enter:

mem0 *address*

For more information, see the *MEMORY windows* discussion on page 3-12.

The *expression* you type in represents the address of the first entry in the MEMORY window. The end of the range is defined by the size of the window: to show more memory locations, make the window larger; to show fewer locations, make the window smaller. (See *Resizing a window*, page 3-21, for more information.)

Expression can be an absolute address, a symbolic address, or any C expression. Here are several examples:

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

  `mem 0x00`

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

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

  `mem &SYM`

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

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

  ```
  mem SP – R0 + 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 3-26) for more details.

Displaying memory contents while you're debugging C

If you're debugging C code in auto mode, you won't see a MEMORY window—the debugger doesn't show the MEMORY window in the C-only display. However, there are several ways to display memory in this situation.

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

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

  ```
  ? *0x26
  ```

  The debugger displays the memory value in the COMMAND window display area.

- If you want the opportunity to observe a specific memory location over a longer period of time, you can display it in a WATCH window. Use the WA command to do this:

  ```
  wa *0x26
  ```

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

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

Saving memory values 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. The syntax for the MS command is:

ms *address, length, filename*

- ❑ The *address* parameter identifies the first address in the block.
- ❑ The *length* parameter defines the length, in words, of the block. This parameter can be any C expression.
- ❑ The *filename* is a system file. If you don't supply an extension, the debugger adds an .obj extension.

For example, to save the values in data memory locations 0x0–0x10 to a file named memsave, you could enter:

`ms 0x0,0x10,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.obj`

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, length, data*

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

For example, to fill locations 0x0080 0000 to 0x0080 0300 with the value 0x1234 ABCD, you would enter:

`fill 0x800000,0x301,0x1234abcd`

If you want to check to see that memory has been filled correctly, you can enter:

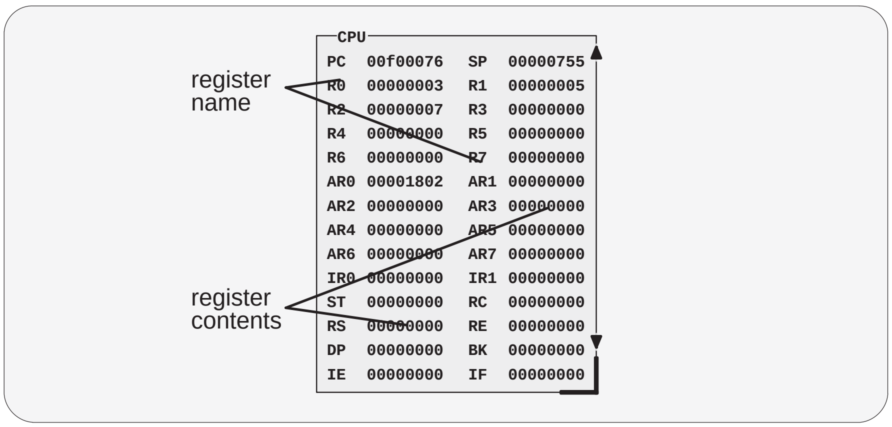
`mem 0x800000`

This changes the MEMORY window display to show the block of memory beginning at address 0x0080 0000.

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

7.5 Managing Register Data

In mixed and assembly modes, the debugger maintains a CPU window that displays the contents of individual registers. For details concerning the CPU window, see the *CPU window* discussion (page 3-15).



The debugger provides commands that allow you to display and modify the contents of specific registers. Remember, 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 7.3, *Basic Methods for Changing Data Values* (page 7-4), 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 two registers, you might want to make the CPU window small and use the extra screen space for the DISASSEMBLY or FILE display. In this type of situation, there are several ways to observe the contents of the selected registers.

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

  ```
  ? SP
  ```

  The debugger displays the SP's current contents in the COMMAND window display area.

- If you want to observe a register over a longer period of time, you can use the WA command to display it 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.

When you're debugging C in auto mode, these methods are also useful because the debugger doesn't show the CPU window in the C-only display.

Accessing extended-precision registers

The simulator represents extended-precision registers in the register file with a set of registers, E*n* and R*n*. The *n* represents the register number. The register ranges are:

Range	Description
E0–E7	Represent the exponent of the floating-point number.
R0–R7	Represent the mantissa of the floating-point number or a 32-bit integer.

For example, if you loaded the 40-bit floating-point number 0x0003 4000 0000 into extended-precision register R1, the simulator will load it as:

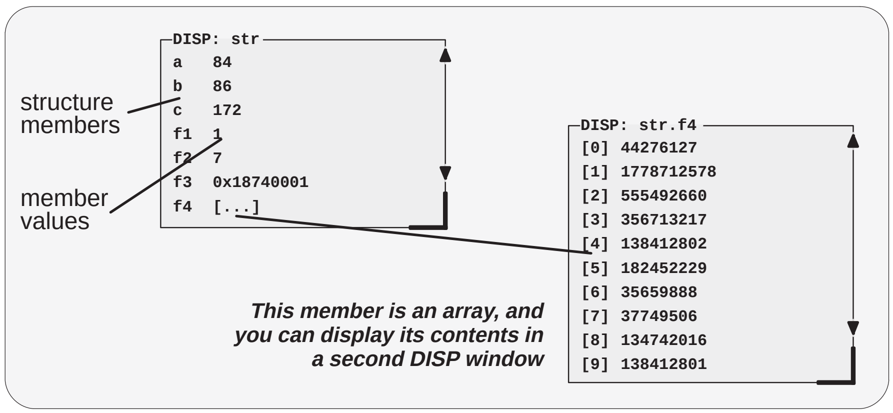
```
E1 = 03h           (exponent)
R1 = 40000000h     (mantissa)
```

Register E1 is essentially a pseudoregister provided by the simulator. Floating-point instructions affect both the exponent and mantissa fields (R*n* and E*n*), but inters instructions affect only the mantissa field (R*n*).

The CPU window displays all of the registers in the primary register and expansion register files; however, the display window displays only the mantissa (R*n*) portion of the register in the extended-precision register file. To display the exponent (E*n*) portion, either use the EVAL command, or add the exponent portion to the WATCH window.

7.6 Managing Data in a DISP (Display) Window

The main purpose of the DISP window is to display the values of members of complex, aggregate data types such as arrays and structures. The debugger shows DISP windows *only when you specifically request to see DISP windows* with the DISP command (described below). Note that you can have up to 120 DISP windows open at once. For additional details about DISP windows, see the *DISP window* discussion (page 3-16).



Remember, you can use the data-management commands or the debugger's overwrite editing capability to modify the contents of any value displayed in a DISP window. Refer to Section 7.3, *Basic Methods for Changing Data Values* (page 7-4), for more information.

Displaying data in a DISP window

disp To open a DISP window, use the DISP command. The basic syntax is:

disp *expression*

If the *expression* is not an array, structure, or pointer (of the form *pointer name), the DISP command behaves like the ? command. However, if *expression* **is** one of these types, the debugger opens a DISP window to display the values of the members.

If a DISP window contains a long list of members, you can use PAGE DOWN, PAGE UP, or arrow keys to scroll through the window. If the window contains an array of structures, you can use CONTROL PAGE DOWN and CONTROL PAGE UP to scroll through the array.

Once you open a DISP window, you may find that a displayed member is another one of these types. This is how you identify the members that are arrays, structures, or pointers:

A member that is an array looks like this:	[. . .]
A member that is a structure looks like this:	{. . .}
A member that is a pointer looks like an address:	0x00000000

You can display the additional data (the data pointed to or the members of the array or structure) in additional DISP windows (these are referred to as *children*). There are three ways to do this.

Use the DISP command again; this time, *expression* must identify the member that has additional data. For example, if the first expression identifies a structure named *str* and one of str's members is an array named *f4*, you can display the contents of the array by entering this command:

```
disp str.f4
```

This opens a new DISP window that shows the contents of the array. If str has a member named *f3* that is a pointer, you could enter:

```
disp *str.f3
```

This opens a window to display what str.f3 points to.

Here's another method of displaying the additional data:

1) Point to the member in the DISP window.

2) Now click the left button.

Here's the third method:

↑ ↓ 1) Use the arrow keys to move the cursor up and down in the list of members.

F9 2) When the cursor is on the desired field, press F9.

When the debugger opens a second DISP window, the new window may at first be displayed on top of the original DISP window; if so, you can move the windows so that you can see both at once. If the new windows also have members that are pointers or aggregate types, you can continue to open new DISP windows.

Closing a DISP window

Closing a DISP window is a simple, two-step process.

Step 1: Make the DISP window that you want to close active (see Section 3.4, *The Active Window*, on page 3-19).

Step 2: Press F4 .

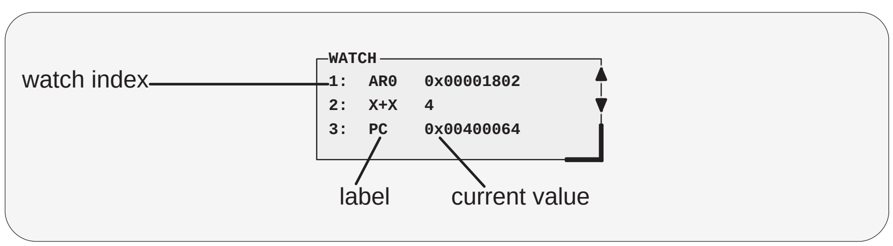
Note that you can close a window and all of its children by closing the original window.

Note:

The debugger automatically closes any DISP windows when you execute a LOAD or SLOAD command.

7.7 Managing Data in a WATCH Window

The debugger doesn't maintain a dedicated window that tells you about the status of all the symbols defined in your program. Such a window might be so large that it wouldn't be useful. Instead, the debugger allows you to create a WATCH window that shows you how program execution affects specific expressions, variables, registers, or memory locations.



The debugger displays a WATCH window *only when you specifically request a WATCH window* with the WA command (described below). Note that there is only one WATCH window. For additional details concerning the WATCH window, see the *WATCH window* discussion (page 3-17).

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 7.3, *Basic Methods for Changing Data Values* (page 7-4), for more information.

Note:

All of the watch commands described can also be accessed from the Watch pulldown menu. For more information about using the the pulldown menus, refer to Section 4.2, *Using the Menu Bar and the Pulldown Menus* (page 4-7).

```
Watch
Add
Delete
Reset
```

Displaying data in the WATCH window

The debugger has one command for adding items to the WATCH window.

wa To open the WATCH window, use the WA (watch add) command. The basic syntax is:

wa *expression* [, *label*]

When you first execute WA, the debugger opens the WATCH window. After that, executing WA adds additional values to the WATCH window.

- The *expression* parameter can be any C expression, including an expression that has side effects. It's most useful to watch an expression whose value will change over time; constant expressions provide no useful function in the watch window.
- 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.

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 7-14 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:

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

7.8 Monitoring the Pipeline (Simulator Only)

The simulator allows you to monitor the pipeline through pseudoregisters that you can query with ? or DISP or add to the WATCH window.

The instruction pipeline consists of four phases: instruction fetch, decode, operand fetch, and execution. During any cycle, one to four instructions can be active, each at a different stage of completion. Instruction operation occurs during the appropriate stages of the pipeline. For example, the instruction AR*n* (*n*=0–7) updates of auxiliary registers occur during the decode phase.

The simulator provides eight pseudoregisters that display the opcode or address of the instructions in each phase of the pipeline. The following table identifies these registers.

Table 7–1. Pipeline Pseudoregisters

Pipeline phase	Opcode pseudoregister	Address pseudoregister
Instruction fetch	fins	faddr
Decode	dins	daddr
Operand fetch	rins	raddr
Execution	xins	xaddr

For example, if you wanted to observe the decode phase during program execution, you could watch the dins and daddr pseudoregisters in the WATCH window:

```
wa dins,Decode–Opcode
wa daddr,Decode–Address
```

This adds dins and daddr to the WATCH window and labels them as Decode-Opcode and Decode-Address, respectively.

7.9 Displaying Data in Alternative Formats

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

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

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

Changing the default format for specific data types

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

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

The *display format* parameter identifies the new display format for any data of type *data type*. Table 7–2 lists the available formats and the corresponding characters that can be used as the *display format* parameter.

Table 7–2. Display Formats for Debugger Data

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

Table 7–3 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 7–3 also shows valid combinations of data types and display formats.

Table 7–3. Data Types for Displaying Debugger Data

	Valid Display Formats									
Data Type	**c**	**d**	**o**	**x**	**e**	**f**	**p**	**s**	**u**	**Default Display Format**
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 unsigned decimals, 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, DISP, and WA

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

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

When you 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`

- To display an array of integers as characters, enter:

  `disp ai,c`

The valid combinations of data types and display formats listed for SETF also apply to the data displayed with DISP, ?, 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 8

Using Software Breakpoints

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

Software breakpoints are especially useful in combination with conditional execution (described on page 6-17) and benchmarking (described on page 6-19).

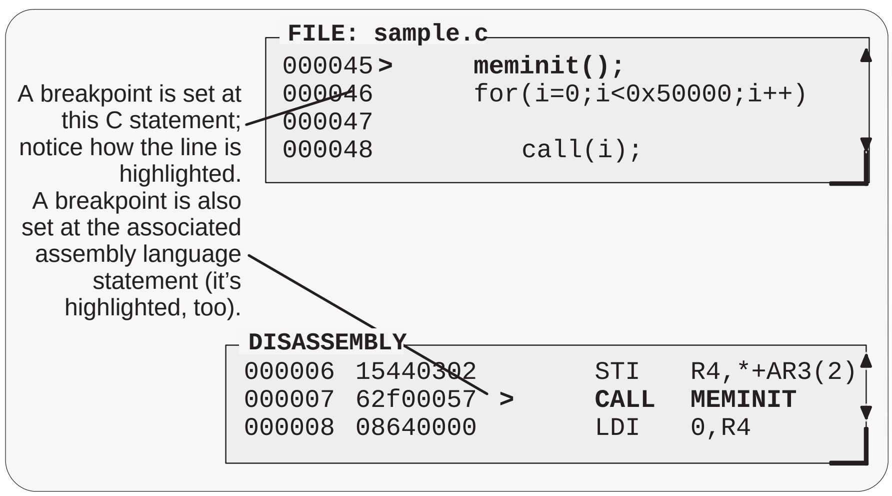
Topic **Page**

8.1 Setting a Software Breakpoint

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

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

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



Note:

- ❑ 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 software breakpoint:

---

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

2) Click the left button.

Repeating this action clears the breakpoint.

---

1) Make the FILE or DISASSEMBLY window the active window.

2) Use the arrow keys to move the cursor to the line of code where you'd like to set a breakpoint.

F9 3) Press the F9 key.

Repeating this action clears the breakpoint.

---

ba If you know the address where you'd like to set a software breakpoint, you can use the BA (breakpoint add) command. This command is useful because it doesn't require you to search through code to find the desired line. The syntax for the BA command is:

ba *address*

This command sets a breakpoint at *address*. This parameter can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label. You cannot set multiple breakpoints at the same statement.

8.2 Clearing a Software Breakpoint

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

↖ 1) Point to a breakpointed assembly language or C statement.

2) Click the left button.

↑ ↓ 1) Use the arrow keys or the DASM command to move the cursor to a breakpointed assembly language or C statement.

F9 2) Press the F9 key.

---

br If you want to clear **all** the software 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 software breakpoint and you know the address of this breakpoint, you can use the BD (breakpoint delete) command. The syntax for the BD command is:

bd *address*

This command clears the breakpoint at *address*. This parameter can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label. If no breakpoint is set at *address*, the debugger ignores the command.

8.3 Finding the Software Breakpoints That Are Set

bl Sometimes you may need to know where software 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 software breakpoints that are currently set in your program. The syntax for this command is:

bl

The BL command displays a table of software breakpoints in the COMMAND window display area. BL lists all the software breakpoints that are set, in the order in which you set them. Here's an example of this type of list:

```
 Address       Symbolic Information
00400065
00400007      in main, at line 45, ”c:\c3xhll\sample.c”
00400066
```

The address is the memory address of the breakpoint. The symbolic information identifies the function, line number, and filename of the breakpointed C statement:

- If the breakpoint was set in assembly language code, you'll see only an address unless the statement defines a symbol.
- If the breakpoint was set in C code, you'll see the address together with symbolic information.

Chapter 9

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.

9.1 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*, *attribute*$_1$ [, *attribute*$_2$ [, *attribute*$_3$ [, *attribute*$_4$]]]
scolor *area name*, *attribute*$_1$ [, *attribute*$_2$ [, *attribute*$_3$ [, *attribute*$_4$]]]

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 9–1 lists the valid values for the *attribute* parameters.

Table 9–1. 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 9–2 lists valid values for the *area name* parameters. This is a long list; the subsections following the table further identify these areas.

Table 9–2. Summary of Area Names for the COLOR and SCOLOR Commands

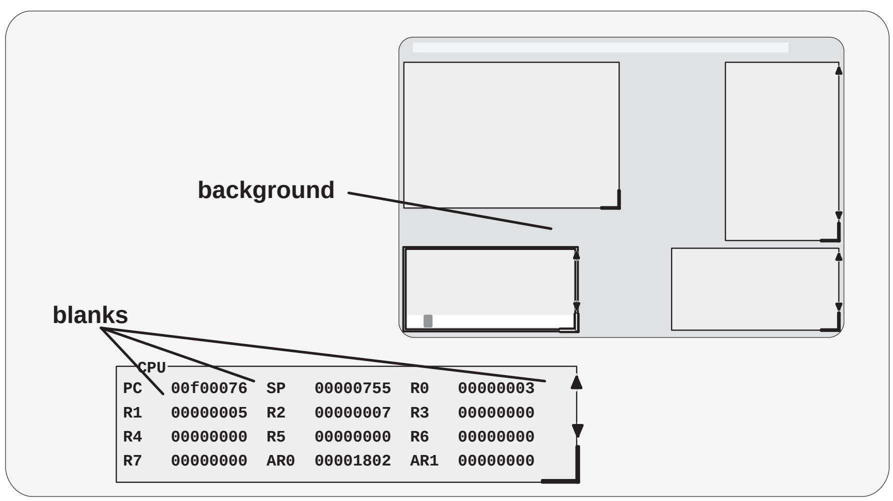
menu_bar	menu_border	menu_entry	menu_cmd
menu_hilite	menu_hicmd	win_border	win_hiborder
win_resize	field_text	field_hilite	field_edit
field_label	field_error	cmd_prompt	cmd_input
cmd_cursor	cmd_echo	asm_data	asm_cdata
asm_label	asm_clabel	background	blanks
error_msg	file_line	file_eof	file_text
file_brk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

Note: Listing order is left to right, top to bottom.

You don't have to type an entire *attribute* or *area name*; you need to 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 9–2 (left to right, top to bottom).

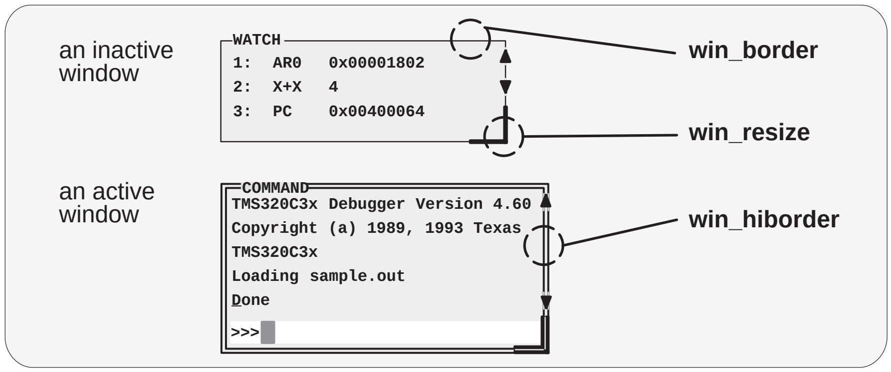
The remainder of this section identifies these areas.

Area names: common display areas



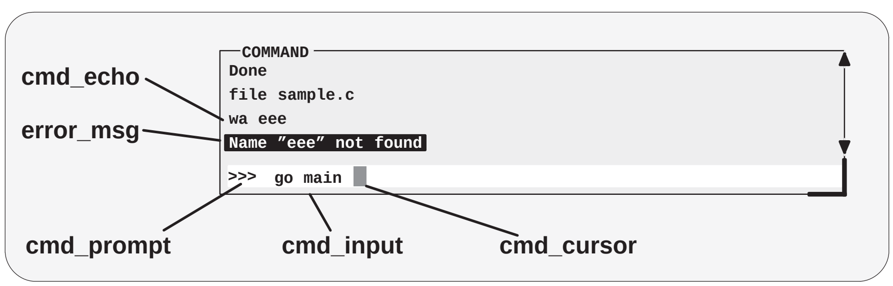
Area identification	Parameter name
Screen background (behind all windows)	background
Window background (inside windows)	blanks

Area names: window borders



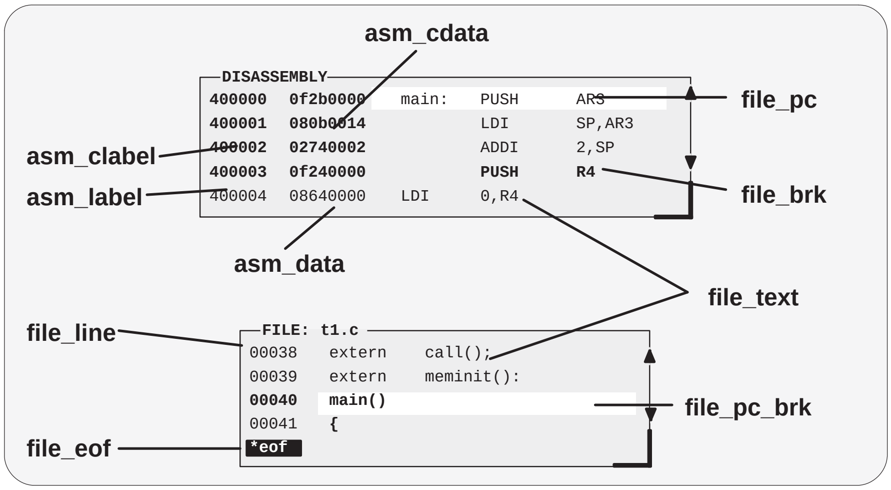
Area identification	Parameter name
Window border for any window that isn't active	win_border
The reversed "L" in the lower right corner of a resizable window	win_resize
Window border of the active window	win_hiborder

Area names: COMMAND window



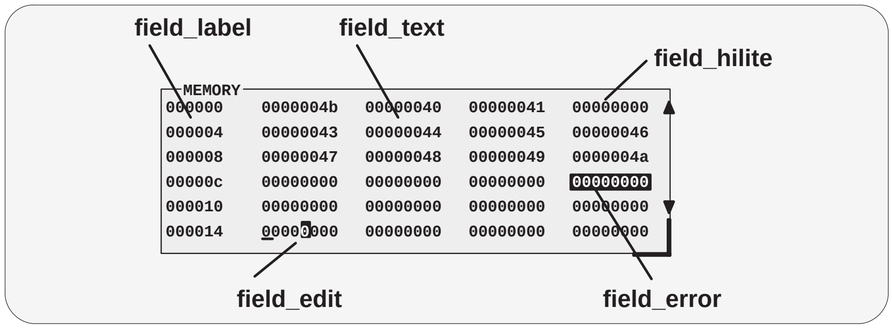
Area identification	Parameter name
Echoed commands in display area	cmd_echo
Errors shown in display area	error_msg
Command-line prompt	cmd_prompt
Text that you enter on the command line	cmd_input
Command-line cursor	cmd_cursor

Area names: DISASSEMBLY and FILE windows



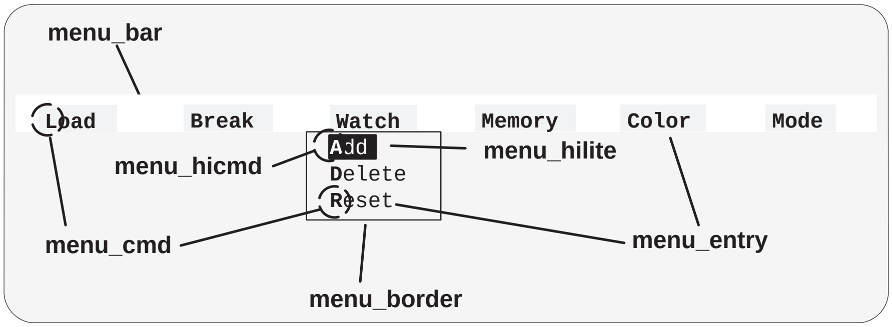
Area identification	Parameter name
Object code in DISASSEMBLY window that is associated with current C statement	asm_cdata
Object code in DISASSEMBLY window	asm_data
Addresses in DISASSEMBLY window	asm_label
Addresses in DISASSEMBLY window that are associated with current C statement	asm_clabel
Line numbers in FILE window	file_line
End-of-file marker in FILE window	file_eof
Text in FILE or DISASSEMBLY window	file_text
Breakpointed text in FILE or DISASSEMBLY window	file_brk
Current PC in FILE or DISASSEMBLY window	file_pc
Breakpoint at current PC in FILE or DISASSEMBLY window	file_pc_brk

Area names: data-display windows



Area identification	Parameter name
Label of a window field (includes register names in CPU window, addresses in MEMORY window, index numbers and labels in WATCH window, member names in DISP window)	field_label
Text of a window field (includes data values for all data-display windows) and of most command output messages in command window	field_text
Text of a highlighted field	field_hilite
Text of a field that has an error (such as an invalid memory location)	field_error
Text of a field being edited (includes data values for all data-display windows)	field_edit

Area names: menu bar and pulldown menus



Area identification	Parameter name
Top line of display screen; background to main menu choices	menu_bar
Border of any pulldown menu	menu_border
Text of a menu entry	menu_entry
Invocation key for a menu or menu entry	menu_cmd
Text for current (selected) menu entry	menu_hilite
Invocation key for current (selected) menu entry	menu_hicmd

9.2 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*.

9.3 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) Next, 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 monitors.

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 resolution, border styles, colors, window positions, window sizes, and (on PCs) video mode (EGA, VGA, CGA, etc.) 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 specify path information, the debugger places the file in the current directory. If you don't supply a filename, 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 resolution, colors, window positions, window sizes, border styles, and (on PCs) video mode (EGA, CGA, MDA, etc.) saved in *filename*. Screen resolution and video mode are restored either by changing the mode (on video cards with switchable modes) or by resizing the debugger screen (on other hosts).

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.

Note:

The file created by the SSAVE command in this version of the debugger saves positional, screen size, and video mode information that was not saved by SSAVE in previous versions of the debugger. The format of this new information is not compatible with the old format. If you attempt to load an earlier version's SCONFIG file, the debugger will issue an error message and stop the load.

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 batch file that the debugger executes at invocation time (init.cmd). This line should use the SCONFIG command to load the custom configuration.

Returning to the default display

If you saved a custom configuration into init.clr 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.

9.4 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 init.cmd file that the debugger executes at invocation time.

You can also execute this command as the Prompt selection on the Color pull-down menu.

Chapter 10

Profiling Code Execution

The profiling environment is a special debugger environment that lets you collect execution statistics for your code. This environment is available on all debugger platforms except for DOS.

Note that the profiling environment is *separate* from the basic debugging environment; the only way to switch between the two environments is by exiting and then reinvoking the debugger.

Topic **Page**

10.1 An Overview of the Profiling Process

Profiling consists of five simple steps:

Step 1	Enter the profiling environment.	See *Entering the Profiling Environment*, page 10-3.
Step 2	Identify the areas of code where you'd like to collect statistics.	See *Defining Areas for Profiling*, page 10-5.
Step 3	Identify the profiling session stopping points.	See *Defining a Stopping Point,* page 10-13.
Step 4	Begin profiling.	See *Running a Profiling Session,* page 10-15.
Step 5	View the profile data.	See *Viewing Profile Data,* page 10-17.

Note:

When you compile a program that will be profiled, you must use the –g and the –as options. The –g option includes symbolic debugging information; the –as option ensures that you will be able to include ranges as profile areas.

A profiling strategy

The profiling environment provides a method for collecting execution statistics about specific areas in your code. This gives you immediate feedback on your application's performance. Here's a suggestion for a basic approach to optimizing the performance of your program.

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

10.2 Entering the Profiling Environment

The profiling environment is available on all debugger platforms except DOS. To enter the profiling environment, invoke the debugger with the **–profile** option. At the system command line, enter the appropriate command:

emulator: **`emu3x –profile`**

simulator: **`sim3x –profile`**

EVM: **`evm30 –profile`**

Use any additional debugger options that you desire (–b, –p, etc.).

Restrictions of the profiling environment

Some restrictions apply to the profiling environment:

- You'll always be in mixed mode.
- COMMAND, DISASSEMBLY, FILE, and PROFILE are the only windows available; additional windows, such as the WATCH window, cannot be opened.
- Breakpoints cannot be set. (However, you can use a similar feature called *stopping points* when you mark sections of code for profiling.)
- The profiling environment supports only a subset of the debugger commands. Table 10–1 lists the debugger commands that can and can't be used in the profiling environment.

Table 10–1. Debugger Commands That Can/Can't Be Used in the Profiling Environment

Can be used		**Can't be used**	
?	ML	ADDR	MIX
ALIAS	MOVE	ASM	MS
CD	MR	BA	NEXT
CLS	PROMPT	BD	PATCH
DASM	QUIT	BL	RETURN
DIR	RELOAD	BORDER	RUN
DLOG	RESET	BR	RUNB
ECHO	RESTART	C	RUNF
EVAL	SCONFIG	CALLS	SCOLOR
FILE	SIZE	CNEXT	SETF
FUNC	SLOAD	COLOR	SOUND
IF/ELSE/ENDIF	SYSTEM	CSTEP	SSAVE
LOAD	TAKE	DISP	STEP
LOOP/ENDLOOP	UNALIAS	FILL	WA
MA	USE	GO	WD
MAP	VERSION	HALT	WHATIS
MC	WIN	MEM	WR
MD	ZOOM		
MI			

Be sure you don't use any of the "can't be used" commands in your initialization batch file.

Using pulldown menus in the profiling environment

The debugger displays a different menu bar in the profiling environment:

```
Load   mAp   Mark  Enable  Disable  Unmark  View   Stop–points  Profile
```

The Load menu corresponds to the Load menu in the basic debugger environment. The mAp menu provides memory map commands available from the basic Memory menu. The other entries provide access to profiling commands and features.

The profiling environment's pulldown menus operate similarly to the basic debugger pulldown menus. However, several of the menus have additional submenus. A submenu is indicated by a > character following a menu item. For example, here's one of the submenus for the Mark menu:

```
Mark
C level     >
Asm level   >   Line areas      >
                Range areas     >   Explicitly
                Function areas  >   in one Function
```

Chapter 11, *Summary of Commands and Special Keys*, shows which debugger commands are associated with the menu items in the basic debugger pulldown menus. Because the profiling environment supports over 100 profile-specific commands, it's not practical to show the commands associated with the menu choices. Here's a tip to help you with the profiling commands: the highlighted menu letters form the name of the corresponding debugger command. For example, if you prefer the function-key approach to using menus, the highlighted letters in **M**ark→ **C** level→**L**ine areas→in one **F**unction show that you could press ALT M, C, L, F. This also shows that the corresponding debugger command is MCLF.

10.3 Defining Areas for Profiling

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

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

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

The mouse is the simplest way to mark, disable, enable, and unmark tasks. The pulldown menus also support these tasks and more complex tasks.

The following subsections explain how to mark, disable, re-enable, and unmark profile areas by using the mouse or the pulldown menus. The individual commands are summarized in *Restrictions of the profiling environment* on page 10-3. *Restrictions on profiling areas* are summarized on page 10-12.

Marking an area

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

Remember, to display C code, use the FILE or FUNC command; to display disassembly, use the DASM command.

Notes:

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

Marking a line. These instructions apply to both C and disassembly.

1) Point to the line you want to mark.

2) Click the left mouse button.

   The beginning of the line will be highlighted with a blinking >>.

3) Click the left mouse button again.

   The beginning of the line will be highlighted with Le> *(line enabled).*

Marking a range. These instructions apply to both C and disassembly.

1) Point to the first line of the range you want to mark.

2) Click the left mouse button.

   The beginning of the line will be highlighted with a blinking >>*.*

3) Point to the last line of the range.

4) Click the left mouse button again.

   The beginning of the line will be highlighted with Re> *(range enabled), marking the beginning of the range. The last line will be highlighted with* <<*, marking the end of the range.*

Marking a function. These instructions apply to C only.

1) Point to the statement that declares the function you want to mark.

2) Click the left mouse button.

   The beginning of the line will be highlighted with Fe> *(function enabled).*

Table 10–2 lists the menu selections for marking areas. The highlighted areas show the keys that you can use if you prefer to use the function-key method of selecting menu choices.

Table 10–2. Menu Selections for Marking Areas

To mark this area	**C only:** **M**ark→**C** level	**Disassembly only:** **M**ark→**A**sm level
Lines	→**L**ine areas	→**L**ine areas
❑ By line number†	→**E**xplicitly	→**E**xplicitly
❑ All lines in a function	→in one **F**unction	→in one **F**unction
Ranges	→**R**ange areas	→**R**ange areas
❑ By line numbers†	→**E**xplicitly	→**E**xplicitly
Functions	→**F**unction areas	
❑ By function name	→**E**xplicitly	not applicable
❑ All functions in a module	→in one **M**odule	
❑ All functions everywhere	→**G**lobally	

† C areas are identified by line number; disassembly areas are identified by address.

Disabling an area

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

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

Note:

If you disable an area after you’ve already collected statistics on it, that information will be lost.

The simplest way to disable an area is to use the mouse, as described below.

Disabling a line area:

1) Point to the marked line.

2) Click the left mouse button once.

   The beginning of the line will be highlighted with Ld> *(line disabled).*

Disabling a range area:

1) Point to the marked line.

2) Click the left mouse button once.

   The beginning of the line will be highlighted with Rd> *(range disabled).*

Disabling a function area:

1) Point to the marked statement that declares the function.

2) Click the left mouse button once.

   The beginning of the line will be highlighted with Fd> *(function disabled).*

Table 10–3 lists the menu selections for disabling areas. The highlighted areas show the keys that you can use if you prefer to use the function-key method of selecting menu choices.

Table 10–3. Menu Selections for Disabling Areas

To disable this area	C only: **D**isable→**C** level	Disassembly only: **D**isable→**A**sm level	C *and* disassembly: **D**isable→**B**oth levels
Lines	→**L**ine areas	→**L**ine areas	→**L**ine areas
❑ By line number†	→**E**xplicitly	→**E**xplicitly	not applicable
❑ All lines in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All lines in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All lines everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Ranges	→**R**ange areas	→**R**ange areas	→**R**ange areas
❑ By line numbers†	→**E**xplicitly	→**E**xplicitly	not applicable
❑ All ranges in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All ranges in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All ranges everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Functions	→**F**unction areas	not applicable	→**F**unction areas
❑ By function name	→**E**xplicitly		not applicable
❑ All functions in a module	→in one **M**odule		→in one **M**odule
❑ All functions everywhere	→**G**lobally		→**G**lobally
All areas	→**A**ll areas	→**A**ll areas	→**A**ll areas
❑ All areas in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All areas in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All areas everywhere	→**G**lobally	→**G**lobally	→**G**lobally

† C areas are identified by line number; disassembly areas are identified by address.

Re-enabling a disabled area

When an area has been disabled and you would like to profile it once again, you must enable the area. To use the mouse, just point to the line, the function, or the first line of a range, and click the left mouse button; the range will once again be highlighted in the same way as a marked area.

In addition to using the mouse, you can enable an area by using one of the commands listed in Table 10–4. However, the easiset way to enter these commands is by accessing them from the Enable menu.

Table 10–4. Menu Selections for Enabling Areas

To enable this area	**C only:** **E**nable→**C** level	**Disassembly only:** **E**nable→**A**sm level	**C *and* disassembly:** **E**nable→**B**oth levels
Lines	→**L**ine areas	→**L**ine areas	→**L**ine areas
❏ By line number†	→**E**xplicitly	→**E**xplicitly	not applicable
❏ All lines in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❏ All lines in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❏ All lines everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Ranges	→**R**ange areas	→**R**ange areas	→**R**ange areas
❏ By line numbers†	→**E**xplicitly	→**E**xplicitly	not applicable
❏ All ranges in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❏ All ranges in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❏ All ranges everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Functions	→**F**unction areas	not applicable	→**F**unction areas
❏ By function name	→**E**xplicitly		not applicable
❏ All functions in a module	→in one **M**odule		→in one **M**odule
❏ All functions everywhere	→**G**lobally		→**G**lobally
All areas	→**A**ll areas	→**A**ll areas	→**A**ll areas
❏ All areas in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❏ All areas in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❏ All areas everywhere	→**G**lobally	→**G**lobally	→**G**lobally

† C areas are identified by line number; disassembly areas are identified by address.

Unmarking an area

If you want to stop collecting information about a specific area, unmark it. You can use the mouse or key method.

Unmarking a line area:

1) Point to the marked line.
2) Click the right mouse button once.

The line will no longer be highlighted.

Unmarking a range area:

1) Point to the marked line.
2) Click the right mouse button once.

The line will no longer be highlighted.

Unmarking a function area:

1) Point to the marked statement that defines the function.
2) Click the right mouse button once.

The line will no longer be highlighted.

Table 10–5 lists the selections on the Unmark menu.

Table 10–5. Menu Selections for Unmarking Areas

To unmark this area	C only: Unmark→C level	Disassembly only: Unmark→Asm level	C *and* disassembly: Unmark→Both levels
Lines	→**L**ine areas	→**L**ine areas	→**L**ine areas
❑ By line number†	→**E**xplicitly	→**E**xplicitly	not applicable
❑ All lines in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All lines in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All lines everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Ranges	→**R**ange areas	→**R**ange areas	→**R**ange areas
❑ By line numbers†	→**E**xplicitly	→**E**xplicitly	not applicable
❑ All ranges in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All ranges in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All ranges everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Functions	→**F**unction areas	not applicable	→**F**unction areas
❑ By function name	→**E**xplicitly		not applicable
❑ All functions in a module	→in one **M**odule		→in one **M**odule
❑ All functions everywhere	→**G**lobally		→**G**lobally
All areas	→**A**ll areas	→**A**ll areas	→**A**ll areas
❑ All areas in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All areas in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All areas everywhere	→**G**lobally	→**G**lobally	→**G**lobally

† C areas are identified by line number; disassembly areas are identified by address.

Restrictions on profiling areas

The following restrictions apply to profiling areas:

- ❑ There must be a minimum of three instructions between a delayed branch and the beginning of an area.
- ❑ An area cannot begin or end on the RPTS instruction or on the instruction to be repeated.
- ❑ An area cannot begin or end on the last instruction of a repeat block.

10.4 Defining a Stopping Point

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

Each stopping point is highlighted in the FILE or DISASSEMBLY window with a ***** character at the beginning of the line. Even though no statistics can be gathered for areas following a stopping point, the areas will be listed in the PROFILE window.

You can use the mouse or commands to add or delete a stopping point; you can also use commands to list or reset all the stopping points.

Note:

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

To set a stopping point:

1) Point to the statement that you want to add as a stopping point.

2) Click the right mouse button.

To remove a stopping point:

1) Point to the statement marking the stopping point that you want to delete.

2) Click the right mouse button.

The debugger supports several commands for adding, deleting, resetting, and listing stopping points (described below); all of these commands can also be entered from the Stop-points menu.

sa To add a stopping point, use the SA (stop add) command. The syntax for this command is:

sa *address*

This adds *address* as a stopping point. The *address* parameter can be a label, a function name, or a memory address.

sd To delete a stopping point, use the SD (stop delete) command. The syntax for this command is:

sd *address*

This deletes *address* as a stopping point. As for SA, the *address* can be a label, a function name, or a memory address.

sr To delete all the stopping points at once, use the SR (stop reset) command. The syntax for this command is:

sr

This deletes all stopping points, including the default *exit* (if it exists).

sl To see a list of all the stopping points that are currently set, use the SL (stop list) command. The syntax for this command is:

sl

10.5 Running a Profiling Session

Once you have defined profile areas and a stopping point, you can run a profiling session. You can run two types of profiling sessions:

- A **full profile** collects a full set of statistics for the defined profile areas.
- A **quick profile** collects a subset of the available statistics (it doesn't collect exclusive or exclusive max data, which are described in Section 10.6). This reduces overhead because the debugger doesn't have to track entering/exiting subroutines within an area.

The debugger supports commands for running both types of sessions. In addition, the debugger supports a command that helps you to resume a profiling session. All of these commands can also be entered from the Profile menu.

pf To run a full profiling session, use the PF (profile full) command. The syntax for this command is:

pf *starting point* [*, update rate*]

pq To run a quick profiling session, use the PQ (profile quick) command. The syntax for this command is:

pq *starting point* [*, update rate*]

The debugger will collect statistics on the defined areas between the *starting point* and the stopping point. The *starting point* parameter can be a label, a function name, or a memory address. There is no default starting point.

The *update rate* is an optional parameter that determines how often the statistics listed in the PROFILE window will be updated. The *update rate* parameter can have one of these values:

0 An *update rate* of 0 means that the statistics listed in the PROFILE window are not updated until the profiling session is halted. A “spinning wheel” character will be shown at the beginning of the PROFILE window label line to indicate that a profiling session is in progress. 0 is the default value.

≥**1** If a number greater than or equal to 1 is supplied, the statistics in the PROFILE window are updated during the profiling session. If a value of **1** is supplied, the data will be updated as often as possible. When larger numbers are supplied, the data is updated less often.

<**0** If a negative number is supplied, the statistics listed in the PROFILE window are not updated until the profiling session is halted. The “spinning wheel” character is not displayed.

No matter which *update rate* you choose, you can force the PROFILE window to be updated during a profiling session by pointing to the window header and clicking a mouse button.

After you enter a PF or PQ command, your program restarts and runs to the defined starting point. Profiling begins when the starting point is reached and continues until a stopping point is reached or until you halt the profiling session by pressing ESC.

pr Use the PR command to resume a profiling session that has halted. The syntax for this command is:

pr [*clear data* [, *update rate*]]

The optional *clear data* parameter tells the debugger whether or not it should clear out the previously collected data. The *clear data* parameter can have one of these values:

0 The profiler will continue to collect data (adding it to the existing data for the profiled areas) and to use the previous internal profile stacks. 0 is the default value.

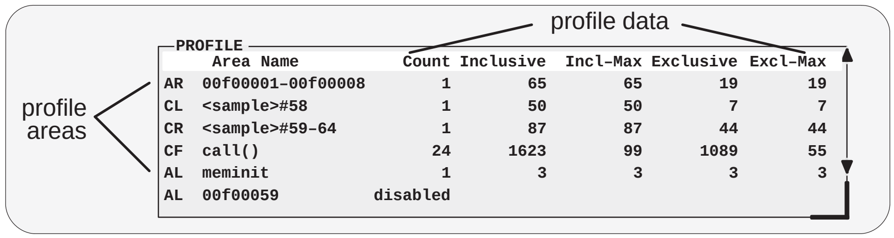
nonzero All previously collected profile data and internal profile stacks are cleared.

The *update rate* parameter is the same as for the PF and PQ commands.

10.6 Viewing Profile Data

The statistics collected during a profiling session are displayed in the PROFILE window. Figure 10–1 shows an example of this window.

Figure 10–1. An Example of the PROFILE Window



The example in Figure 10–1 shows the PROFILE window with some default conditions:

- ❑ Column headings show the labels for the default set of profile data, including *Count, Inclusive, Incl-Max, Exclusive*, and *Excl-Max*.
- ❑ The data is sorted on the address of the first line in each area.
- ❑ All marked areas are listed, including disabled areas.

You can modify the PROFILE window to display selected profile areas or different data; you can also sort the data differently. The following subsections explain how to do these things.

Note:

To reset the PROFILE display back to its default characteristics, use View→Reset.

Viewing different profile data

By default, the PROFILE window shows a set of statistics labeled as Count, Inclusive, Incl-Max, Exclusive, and Excl-Max. The address, which is not part of the default statistics, can also be displayed. Table 10–6 describes the statistic that each field represents.

Table 10–6. Types of Data Shown in the PROFILE Window

Label	**Profile data**
Count	The number of times a profile area is entered during a session.
Inclusive	The total execution time (cycle count) of a profile area, including the execution time of any subroutines called from within the profile area.
Incl-Max (inclusive maximum)	The maximum inclusive time for one iteration of a profile area. If the profiled code contains no flow control (such as conditional processing), inclusive-maximum will equal the inclusive timing divided by the count.
Exclusive	The total execution time (cycle count) of a profile area, excluding the execution time of any subroutines called from within the profile area. In general, the exclusive data provides the best statistics for comparing the execution time of one profile area to another area.
Excl-Max (exclusive maximum)	The maximum exclusive time for one iteration of a profile area.
Address	The memory address of the line. If the area is a function or range, the Address field shows the memory address of the first line in the area.

In addition to viewing this data in the default manner, you can view each of these statistics individually. The benefit of viewing them individually is that in addition to a cycle count, you are also supplied with a percentage indication and a histogram.

In order to view the fields individually, you can use the mouse—just point to the header line in the PROFILE window and click a mouse button. You can also use the View→Data menu to select the field you'd like to display. When you use the left mouse button to click on the header, fields are displayed individually in the order listed below on the left. (Use the right mouse button to go in the opposite direction.) On the right are the corresponding menu selections.

Count	**V**iew→**D**ata	→**C**ount
Inclusive		→**I**nclusive
Incl-max		→I**n**clusive Max
Exclusive		→**E**xclusive
Excl-max		→E**x**clusive Max
Address		→**A**ddress
Default		→A**ll**

One advantage of using the mouse is that you can change the display while you're profiling.

Data accuracy

During a profiling session, the debugger sets many internal breakpoints and issues a series of RUNB commands. As a result, the processor is momentarily halted when entering and exiting profiling areas. This stopping and starting can affect the cycle count information (due to pipeline flushing and the mechanics of software breakpoints) so that it varies from session to session. This method of profiling is referred to as *intrusive profiling*.

Treat the data as *relative*, not absolute. The percentages and histograms are relevant only to the cycle count from the starting point to the stopping point—not to overall performance. Even though the cycle counts may change if you profiled the same area twice, the relationship of that area to other profiled areas should not change.

Sorting profile data

By default, the data displayed in the PROFILE window is sorted on the memory addresses of the displayed areas. The area with the least significant address is listed first, followed by the area with the most significant address, etc. When you view fields individually, the data is automatically sorted from highest cycle count to lowest (instead of by address).

You can sort the data on any of the data fields by using the View→Sort menu. For example, to sort all the data based on the values of the Inclusive field, use View→Sort→Inclusive; the area with the highest Count field will display first, and the area with the lowest Count field will display last. This applies even when you are viewing individual fields.

Viewing different profile areas

By default, all marked areas are listed in the PROFILE window. You can modify the window to display selected areas. To do this, use the selections on the View→Filter pulldown menu; these selections are summarized in Table 10–7.

Table 10–7. Menu Selections for Displaying Areas in the PROFILE Window

To view these areas	**C only:** **V**iew→**F**ilter→**C** level	**Disassembly only:** **V**iew→**F**ilter→**A**sm level	**C *and* disassembly:** **V**iew→**F**ilter→**B**oth levels
Lines	→**L**ine areas	→**L**ine areas	→**L**ine areas
❑ By line number	→**E**xplicitly	→**E**xplicitly	not applicable
❑ All lines in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All lines in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All lines everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Ranges	→**R**ange areas	→**R**ange areas	→**R**ange areas
❑ By line numbers	→**E**xplicitly	→**E**xplicitly	not applicable
❑ All ranges in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All ranges in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All ranges everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Functions	→**F**unction areas	not applicable	→**F**unction areas
❑ By function name	→**E**xplicitly		not applicable
❑ All functions in a module	→in one **M**odule		→in one **M**odule
❑ All functions everywhere	→**G**lobally		→**G**lobally
All areas	→**R**ange areas	→**R**ange areas	→**R**ange areas
❑ All areas in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All areas in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All areas everywhere	→**G**lobally	→**G**lobally	→**G**lobally

Interpreting session data

General information about a profiling session is displayed in the COMMAND window during and after the session. This information identifies the starting and stopping points. It also lists statistics for three important areas:

- ❑ **Run cycles** shows the number of execution cycles consumed by the program from the starting point to the stopping point.
- ❑ **Profile cycles** equals the run cycles minus the cycles consumed by disabled areas.
- ❑ **Hits** shows the number of internal breakpoints encountered during the profiling session.

Viewing code associated with a profile area

You can view the code associated with a displayed profile area. The debugger will update the display so that the associated C or disassembly statements are shown in the FILE or DISASSEMBLY windows.

Use the mouse to select the profile area in the PROFILE window and display the associated code:

1) Point to the appropriate area name in the PROFILE window.

2) Click the left mouse button.

The area name and the associated C or disassembly statement will be highlighted. To view the code associated with another area, point and click again.

If you are attempting to show disassembly, you may have to make several attempts because program memory can be accessed only when the target is not running.

10.7 Saving Profile Data to a File

You may want to run several profiling sessions during a debugging session. Whenever you start a new profiling session,the results of the previous session are lost. However, you can save the results of the current profiling session to a system file.There are two commands that you can use to do this:

vac To save the contents of the PROFILE window to a system file, use the VAC (view save current) command. The syntax for this command is:

vac *filename*

This saves only the current view; if, for example, you are viewing only the Count field, then only that information will be saved.

vaa To save all data for the currently displayed areas, use the VAA (view save all) command. The syntax for this command is:

vaa *filename*

This saves all views of the data—including the individual count, inclusive, etc.—with the percentage indications and histograms.

Both commands write profile data to *filename*. The filename can include path information. There is no default filename. If *filename* already exists, the command will overwrite the file with the new data.

Note that if the PROFILE window displays only a subset of the areas that are marked for profiling, data is saved *only for those areas that are displayed*. (For VAC, the currently displayed data will be saved for the displayed areas. For VAA, all data will be saved for the displayed areas.) If some areas are hidden and you want to save all the data, be sure to select View→Reset before saving the data to a file.

The file contents are in ASCII and are formatted in exactly the same manner as they are displayed (or would be displayed) in the PROFILE window. The general profiling-session information that is displayed in the COMMAND window is also written to the file.

Chapter 11

Summary of Commands and Special Keys

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

Topic **Page**

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 freely between the three debugging modes (auto, mixed, and assembly). You can also select these commands from the Mode pulldown menu.
- **Managing windows.** These commands enable you to select the active window and move or resize the active window. You can also perform these functions with the mouse.
- **Displaying and changing data.** These commands enable you to display and evaluate a variety of data items. Some of these commands are also available on the Watch pulldown menu.
- **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 FILE and DISASSEMBLY windows and to load object files into memory. Several of these commands are available on the Load pulldown menu.
- **Managing breakpoints.** These commands provide you with a command-line method for controlling software breakpoints and are also 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. You can also use the Color pulldown menu to access these commands.
- **Memory mapping.** These commands enable you to define the areas of target memory that the debugger can access. You can also use the Memory pulldown menu to access these commands.
- **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.
- **Profiling commands.** These commands enable you to collect execution statistics for your code. Commands can be entered from the pulldown menus or on the command line.

Changing modes

To do this	Use this command	See page
Put the debugger in assembly mode	asm	11-13
Put the debugger in auto mode for debugging C code	c	11-15
Put the debugger in mixed mode	mix	11-28

Managing windows

To do this	Use this command	See page
Reposition the active window	move	11-29
Resize the active window	size	11-39
Select the active window	win	11-46
Make the active window as large as possible	zoom	11-47

Displaying and changing data

To do this	Use this command	See page
Evaluate and display the result of a C expression	?setf	11-11
Display the values in an array or structure or display the value that a pointer is pointing to	disp	11-18
Evaluate a C expression without displaying the results	eval	11-21
Display a different range of memory in the MEMORY window	mem	11-27
Display an pop-up MEMORY window	mem1,mem2, mem3	11-27
Change the default format for displaying data values	setf	11-38
Continuously display the value of a variable, register, or memory location within the WATCH window	wa	11-45
Delete a data item from the WATCH window	wd	11-46
Show the type of a data item	whatis	11-46
Delete all data items from the WATCH window and close the WATCH window	wr	11-47

Performing system tasks

To do this	Use this command	See page
Define your own command string	alias	11-12
Change the current working directory from within the debugger environment	cd/chdir	11-15
Clear all displayed information from the COMMAND window display area	cls	11-16
List the contents of the current directory or any other directory	dir	11-18
Record the information shown in the COMMAND window display area	dlog	11-20
Display a string to the COMMAND window while executing a batch file	echo	11-21
Conditionally execute debugger commands in a batch file	if/else/endif	11-23
Loop debugger commands in a batch file	loop/endloop	11-24
Exit the debugger	quit	11-34
Reset the target system (emulator only), the simulator, or the EVM.	reset	11-34
Associate a beeping sound with the display of error messages	sound	11-40
Enter any operating-system command or exit to a system shell	system	11-42
Execute commands from a batch file	take	11-43
Delete an alias definition	unalias	11-43
Name additional directories that can be searched when you load source files	use	11-44

Displaying files and loading programs

To do this	Use this command	See page
Display C and/or assembly language code at a specific point	addr	11-12
Reopen the CALLS window	calls	11-15
Display assembly language code at a specific address	dasm	11-18
Display a text file in the FILE window	file	11-22
Display a specific C function	func	11-22
Load an object file	load	11-24
Modify disassembly with the patch assembler	patch	11-31
Load only the object-code portion of an object file	reload	11-34
Load only the symbol-table portion of an object file	sload	11-40

Managing breakpoints

To do this	Use this command	See page
Add a software breakpoint	ba	11-13
Delete a software breakpoint	bd	11-13
Display a list of all the software breakpoints that are set	bl	11-13
Reset (delete) all software breakpoints	br	11-14

Customizing the screen

To do this	Use this command	See page
Change the border style of any window	border	11-14
Change the screen colors, but don't update the screen immediately	color	11-16
Change the command-line prompt	prompt	11-33
Change the screen colors and update the screen immediately	scolor	11-36
Load and use a previously saved custom screen configuration	sconfig	11-37
Save a custom screen configuration	ssave	11-41

Memory mapping

To do this	Use this command	See page
Initialize a block of memory	fill	11-22
Add an address range to the memory map	ma	11-25
Enable or disable memory mapping	map	11-26
Connect a simulated I/O port to an input or output file (simulator only)	mc	11-26
Delete an address range from the memory map	md	11-27
Disconnect a simulated I/O port (simulator only)	mi	11-28
Display a list of the current memory map settings	ml	11-28
Reset the memory map (delete all ranges)	mr	11-30
Save a block of memory to a system file	ms	11-30
Connect an input file to the pin	pinc	11-32
Disconnect the input file from the pin	pind	11-32
List the pins that are connected to the input files	pinl	11-32

Running programs

To do this	Use this command	See page
Single-step through assembly language or C code one C statement at a time; step over function calls	cnext	11-16
Single-step through assembly language or C code, one C statement at a time	cstep	11-17
Run a program up to a certain point	go	11-23
Halt the target system after executing a RUNF command (emulator and EVM only)	halt	11-23
Single-step through assembly language or C code; step over function calls	next	11-30
Reset the target system (emulator only), simulator, or EVM	reset	11-34
Reset the program entry point	restart	11-34
Execute code in a function and return to the function's caller	return	11-35
Run a program	run	11-35
Run a program with benchmarking (count the number of CPU clock cycles consumed by the executing portion of code)	runb	11-35
Disconnect the emulator from the target system and run free (emulator and EVM only)	runf	11-36
Single-step through assembly language or C code	step	11-41
Execute commands from a batch file	take	11-43

Profiling commands

All of the profiling commands can be entered from the pulldown menus. In many cases, using the pulldown menus is the easiest way to enter some of these commands. For this reason and also because there are over 100 profiling commands, most of these commands are not described individually in this chapter (as the basic debugger commands are).

Listed below are some of the profiling commands that you might choose to enter from the command line instead of from a menu; these commands are also described in the alphabetical command summary. The remaining profiling commands are summarized in Section 11.4 on page 11-48.

To do this	Use this command	See page
Run a full profiling session	pf	11-31
Run a quick profiling session	pq	11-33
Resume a profiling session	pr	11-33
Add a stopping point	sa	11-36
Delete a stopping point	sd	11-37
List all the stopping points	sl	11-39
Delete all the stopping points	sr	11-40
Save all the profile data to a file	vaa	11-44
Save currently displayed profile data to a file	vac	11-44
Reset the display in the PROFILE window to show all areas and the default set of data	vr	11-45

11.2 How the Menu Selections Correspond to Commands

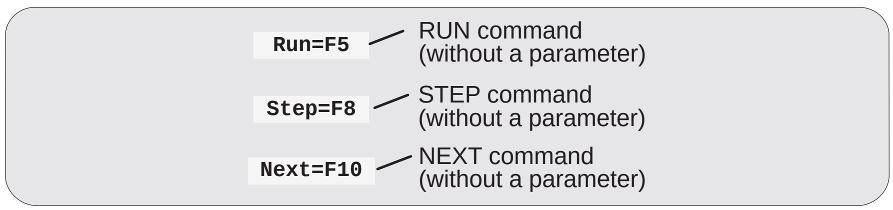
The following sample screens illustrate the relationship of the basic debugger commands to the menu bar and pulldown menus.

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 SC to close the menu.

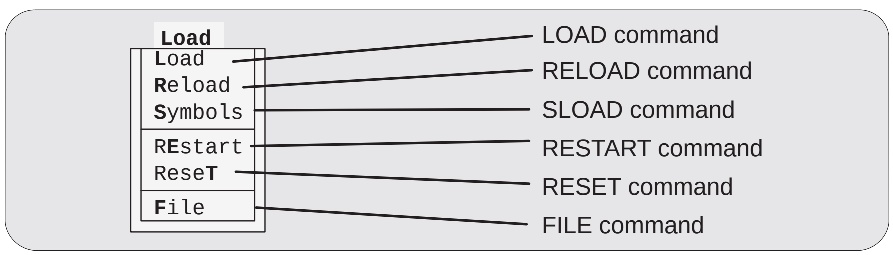
Note:

Because the profiling environment supports over 100 profile-specific commands, it's not practical to show the commands associated with the profile menu choices.

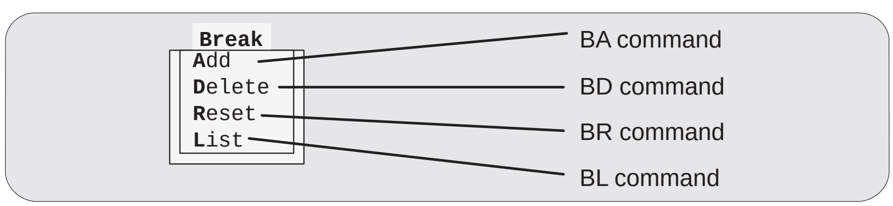
Program-execution commands



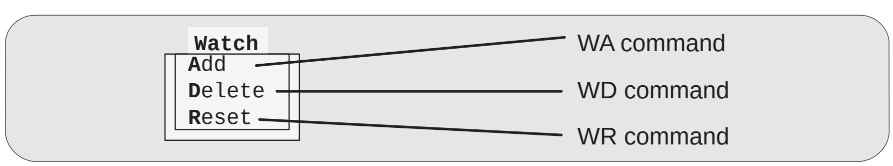
File/load commands



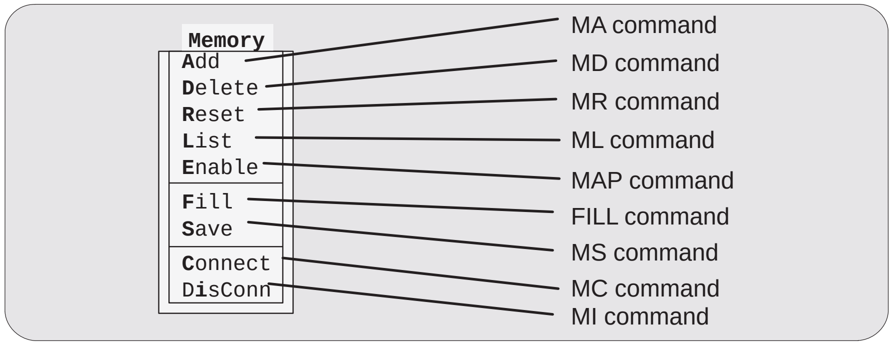
Breakpoint commands



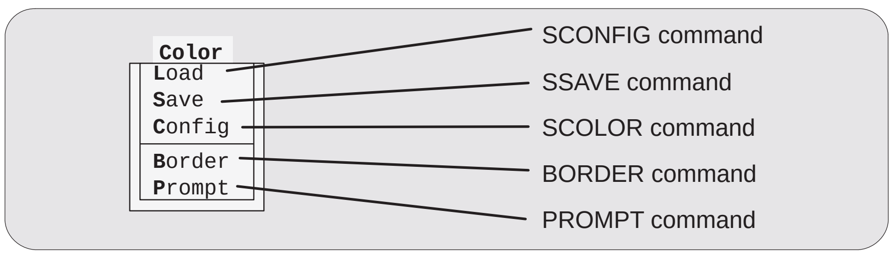
Watch commands



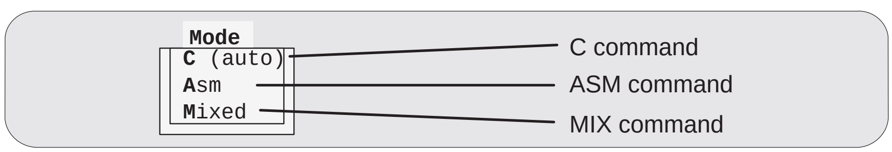
Memory commands



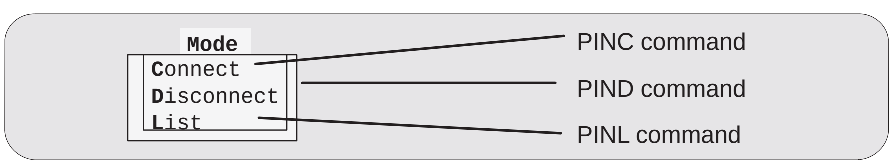
Screen-configuration commands



Mode commands



Interrupt-simulation commands



11.3 Alphabetical Summary of Debugger Commands

There are two debugger environments: the basic debugger environment and the profiling environment. Some debugger commands can be used in both environments; some can be used in only one of the environments. Each command description identifies the applicable environments for the command.

Commands are not case sensitive; to emphasize this, command names are shown in both uppercase and lowercase throughout this book.

?

Evaluate Expression

Syntax **?** *expression* [, *display format*]

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The ? (evaluate expression) command evaluates an expression and shows the result in the COMMAND window display area. The *expression* can be any C expression, including an expression with side effects. However, you cannot use a string constant or function call in the *expression.* If the result of *expression* is not an array or structure, then the debugger displays the results in the COMMAND window. If *expression* is a structure or array, ? displays the entire contents of the structure or array; you can halt long listings by pressing ESC .

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	**o**	Octal
c	ASCII character (bytes)	**p**	Valid address
d	Decimal	**s**	ASCII string
e	Exponential floating point	**u**	Unsigned decimal
f	Decimal floating point	**x**	Hexadecimal

addr

Display Code at Selected Address

Syntax

addr *address*
addr *function name*

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description

Use the ADDR command to display C code or the disassembly at a specific point. ADDR's behavior changes, depending on the current debugging mode:

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

> **Note:**
>
> ADDR affects the FILE window only if the specified *address* is in a C function.

alias

Define Custom Command String

Syntax

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

Menu selection none

Environments ☑ basic debugger ☑ profiling

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 Mode

Syntax **asm**

Menu selection Mo**D**e→**A**sm

Environments ☑ basic debugger ☐ profiling

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

Add Software Breakpoint

Syntax **ba** *address*

Menu selection **B**reak→**A**dd

Environments ☑ basic debugger ☐ profiling

Description The BA command sets a software breakpoint at a specific *address*. This command is useful because it doesn't require you to search through code to find the desired line. The *address* can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label.

bd

Delete Software Breakpoint

Syntax **bd** *address*

Menu selection **B**reak→ **D**elete

Environments ☑ basic debugger ☐ profiling

Description The BD command clears a software breakpoint at a specific *address*. The *address* can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label.

bl

List Software Breakpoint

Syntax **bl**

Menu selection **B**reak→**L**ist

Environments ☑ basic debugger ☐ profiling

Description The BL command provides an easy way to get a complete listing of all the software 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 **C**olor→**B**order

Environments ☑ basic debugger ☐ profiling

Description The BORDER command changes the border style of the active window, the inactive windows, and the border style of 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 — *Reset Software Breakpoints*

Syntax **br**

Menu selection **B**reak→**R**eset

Environments ☑ basic debugger ☐ profiling

Description The BR command clears all software breakpoints that are set.

c — *Enter Auto Mode*

Syntax **c**

Menu selection Mo**D**e→**C** (auto)

Environments ☑ basic debugger ☐ profiling

Description The C command changes from the current debugging mode to auto mode. If you're already in auto mode, then the C command has no effect.

calls — *Open CALLS Window*

Syntax **calls**

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The CALLS command displays the CALLS window. The debugger displays this window automatically when you are in auto/C or mixed mode. However, you can close the CALLS window; the CALLS command opens the window again.

cd, chdir — *Change Directory*

Syntax
cd [*directory name*]
chdir [*directory name*]

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The CD or CHDIR command changes the current working directory from within the debugger. You can use relative pathnames as part of the *directory name*. If you don't use a *pathname,* 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 it is used with the USE command. You can also use the CD command to change the current drive. For example,

```
cd c:
cd d:\csource
cd c:\c3xhll
```

cls — *Clear Screen*

Syntax **cls**

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The CLS command clears all displayed information from the COMMAND window display area.

cnext — *Single-Step C, Next Statement*

Syntax **cnext** [*expression*]

Menu selection Next=**F10** (in C code)

Environments ☑ basic debugger ☐ profiling

Description The CNEXT command is similar to the CSTEP command. It runs a program one C statement at a time, updating the display after executing each statement. If you're using CNEXT to step through assembly language code, the debugger won't update the display until it has executed all assembly language statements associated with a single C statement. Unlike CSTEP, CNEXT steps over function calls rather than stepping into them—you don't see the single-step execution of the function call.

The *expression* parameter specifies the number statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (the *Running code conditionally* discussion, page 6-17, discusses this in detail).

color — *Change Screen Colors*

Syntax **color** *area name, attribute*$_1$ [*,attribute*$_2$ [*,attribute*$_3$ [*,attribute*$_4$]]]

Menu selection none

Environments ☑ basic debugger ☐ profiling

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 area of the display that is affected. The *attributes* identify how the area is affected. The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Valid values for the *attribute* parameters include:

black	blue	green	cyan
red	magenta	yellow	white
bright		blink	

Valid values for the *area name* parameters include:

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

You don't have to type an entire *attribute* or *area name*; you need to type only enough letters to uniquely identify the attribute. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they're listed above (left to right, top to bottom).

cstep — *Single-Step C*

Syntax **cstep** [*expression*]]

Menu selection Step=**F8** (in C code)

Environments ☑ basic debugger ☐ profiling

Description The CSTEP single-steps through a program one C statement at a time, updating the display after executing each statement. If you're using CSTEP to step through assembly language code, the debugger won't update the display until it has executed all assembly language statements associated with a single C statement.

If you're single-stepping through C code and encounter a function call, the STEP command shows you the single-step execution of the called function (assuming that the function was compiled with the compiler's –g debug option). When function execution completes, single-step execution returns to the caller. If the function wasn't compiled with the debug option, the debugger executes the function but doesn't show single-step execution of the function.

The *expression* parameter specifies the number statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (the *Running code conditionally* discussion, page 6-17, discusses this in detail).

dasm — *Display Disassembly at Specified Address*

Syntax
dasm *address*
dasm *function name*

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The DASM command displays code beginning at a specific point within the DISASSEMBLY window.

dir — *List Directory Contents*

Syntax **dir** [*directory name*]

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The DIR command displays a directory listing in the display area of the COMMAND window. If you use the optional *directory name* parameter, the debugger displays a list of the specified directory's contents. If you don't use the parameter, the debugger lists the contents of the current directory.

disp — *Open DISP Window*

Syntax **disp** *expression* [, *display format*]

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The DISP command opens a DISP window to display the contents of an array, structure, or pointer expression to a scalar type (of the form **pointer*). If the *expression* is not one of these types, then DISP acts like a ? command.

Once you open a DISP window, you may find that a displayed member is itself an array, structure, or pointer:

A member that is an array looks like this:	[. . .]
A member that is a structure looks like this:	{. . .}
A member that is a pointer looks like an address:	0x00000000

You can display the additional data (the data pointed to or the members of the array or structure) in another DISP window by using the DISP command again, using the arrow keys to select the field and then pressing F9, or pointing the mouse cursor to the field and pressing the left mouse button. You can have up to 120 DISP windows open at the same time.

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	**o**	Octal
c	ASCII character (bytes)	**p**	Valid address
d	Decimal	**s**	ASCII string
e	Exponential floating point	**u**	Unsigned decimal
f	Decimal floating point	**x**	Hexadecimal

The *display format* parameter can be used only when you are displaying a scalar type, an array of scalar type, or an individual member of an aggregate type.

You can also use the DISP command with a typecast expression to display memory contents in any format. Here are some examples:

```
disp *0
disp *(float *)123
disp *(char *)0x111
```

This shows memory in the DISP window as an array of locations; the location that you specify with the *expression* parameter is member [0], and all other locations are offset from that location.

dlog

Record COMMAND Window Display

Syntax

dlog *filename* [**,**{**a** | **w**}]
or
dlog close

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description

The DLOG command allows you to record the information displayed in the command window into a log file.

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

  dlog *filename*

  Log files can be executed by using the TAKE command. When you use DLOG to record the information from the COMMAND window display area into a log file called *filename*, 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 end the recording session, enter:

  dlog close ↵

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:

- **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.

echo

Echo String to COMMAND Window

Syntax **echo** *string*

Menu selection none

Environments ☑ basic debugger ☑ profiling

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

else

Execute Alternative Debugger Commands

Description ELSE provides an alternative list of debugger commands in the IF/ELSE/ENDIF command sequence. See page 11-23 for more information about the IF/ELSE/ENDIF commands.

endif

Terminate Conditional Sequence

Description ENDIF identifies the end of the IF/ELSE/ENDIF command sequence. See page 11-23 for more information about the IF/ELSE/ENDIF commands.

endloop

Terminate Looping Sequence

Description ENDLOOP identifies the end of the LOOP/ENDLOOP command sequence. See page 11-24 for more information about the LOOP/ENDLOOP commands.

eval

Evaluate Expression

Syntax **eval** *expression*
e *expression*

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The EVAL command evaluates an expression like the ? command does *but does not show the result* in the COMMAND window display area. EVAL is useful for assigning values to registers or memory locations in a batch file (where it's not necessary to display the result).

file

Display Text File

Syntax **file** *filename*

Menu selection **L**oad→**F**ile

Environments ☑ basic debugger ☑ profiling

Description The FILE command displays the contents of any text file in the FILE window. The debugger continues to display this file until you run a program and halt in a C function. This command is primarily intended for displaying C source code. You can view only one text file at a time.

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

fill

Fill Memory

Syntax **fill** *address, length, data*

Menu selection **M**emory→**F**ill

Environments ☑ basic debugger ☐ profiling

Description The FILL command fills a block of memory with a specified value. This command has three parameters:

- The *address* parameter identifies the beginning of the block.
- The *length* parameter defines the number of 32-bit words that will be filled.
- The *data* parameter is the value that is placed in each word in the block.

func

Display Function

Syntax **func** *function name*
func *address*

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The FUNC command displays a specified C function in the FILE window. You can identify the function by its name or its address. Note that FUNC works the same way FILE works, but with FUNC you don't need to identify the name of the file that contains the function.

go — *Run to Specified Address*

Syntax **go** [*address*]

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The GO command executes code up to a specific point in your program. If you don't supply an *address* parameter, then GO acts like a RUN command without an *expression* parameter.

halt — *Halt Target System* — **EVM & Emulator Only**

Syntax **halt**

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The HALT command halts the target system after you've entered a RUNF command. When you invoke the debugger, it automatically executes a HALT command. Thus, if you enter a RUNF, quit the debugger, and later reinvoke the debugger, you will effectively reconnect the emulator to the target system and run the debugger in its normal mode of operation.

if/else/endif — *Conditionally Execute Debugger Commands*

Syntax

if *expression*
debugger commands
[**else**
debugger commands]
endif

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description These commands allow you to execute debugger commands conditionally in a batch file. If the *expression* is nonzero, the debugger executes the commands between the IF and the ELSE or ENDIF. Note that the ELSE portion of the command is optional.

You can substitute a keyword for the expression. Keywords evaluate to true (1) or false (0). You can use the following keywords with the IF command:

- **$$EMU$$** (tests for the emulator version of the debugger)
- **$$SIM$$** (tests for the simulator version of the debugger)
- **$$EVM$$** (tests for the EVM version of the debugger)

The conditional commands work with the following provisions:

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

load — *Load Executable Object File*

Syntax **load** *object filename*

Menu selection **L**oad→ **L**oad

Environments ☑ basic debugger ☑ profiling

Description The LOAD command loads both an object file and its associated symbol table into memory. In effect, the LOAD command performs both a RELOAD and an SLOAD. If you don't supply an extension, the debugger looks for *filename*.out. Note that the LOAD command clears the old symbol table and closes the WATCH and DISP windows.

loop/endloop — *Loop Through Debugger Commands*

Syntax

loop *expression*
debugger commands
endloop

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description 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:

- If you use an *expression* that is not Boolean, the debugger evaluates the expression as a loop count.
- If you use a Boolean *expression*, the debugger executes the command repeatedly as long as the expression is true.

The LOOP/ENDLOOP commands work under the following conditions:

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

ma — *Add Block to Memory Map*

Syntax **ma** *address, length, type*

Menu selection **M**emory→**A**dd

Environments ☑ basic debugger ☑ profiling

Description The MA command identifies valid ranges of target memory. A new memory map must not overlap an existing entry. If you define a range that overlaps an existing range, the debugger ignores the new range.

- The *address* parameter defines the starting address of a range. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.
- The *length* parameter defines the length of the range. This parameter can be any C expression.
- The *type* parameter identifies the read/write characteristics of the memory range. The *type* must be one of these keywords:

To identify this kind of memory	Use this keyword as the *type* parameter
read-only memory	**R**, **ROM**, or **READONLY**
write-only memory	**W**, **WOM**, or **WRITEONLY**
read/write memory	**WR** or **RAM**
no-access memory	**PROTECT**
input port	**IPORT**
output port	**OPORT**
input/output port	**IOPORT**

map — *Enable Memory Mapping*

Syntax **map** {**on** | **off**}

Menu selection **M**emory→**E**nable

Environments ☑ basic debugger ☑ profiling

Description The MAP command enables or disables memory mapping. In some instances, you may want to explicitly enable or disable memory. Note that disabling memory mapping can cause bus fault problems in the target because the debugger may attempt to access nonexistent memory.

mc — *Connect a Simulated I/O Port to a File* **Simulator Only**

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

Menu selection **M**emory→**C**onnect

Environments ☑ basic debugger ☑ profiling

Description The MC command connects IPORT, OPORT, or IOPORT to an input or output file. Before you can connect the port, you must add it to the memory map with the MA command.

- The *port address* parameter defines the address of the I/O port. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.
- The *filename* parameter can be any filename. If you connect a port to read from a file, the file must exist, or the MC command will fail.
- The final parameter is specified as **READ** or **WRITE** and defines how the file will be used (for input or output, respectively).

The file is accessed during an LDI or STI instruction to the associated port address. A maximum of one input and one output file can be connected to a single port; multiple ports can be connected to a single file.

This port-connect feature can also be used for simulation of serial ports. The data transmit and data receive registers of serial port 0 and serial port 1 can be connected to files.

md

Delete Block From Memory Map

Syntax **md** *address*

Menu selection **M**emory→**D**elete

Environments ☑ basic debugger ☑ profiling

Description The MD command deletes a range of memory from the debugger's memory map. The *address* parameter identifies the starting address of the range of memory. Note that if you are attempting to delete a simulated I/O port, you must first disconnect it.

mem

Modify MEMORY Window Display

Syntax **mem**[*#*] *expression* [*, display format*]

Menu selection none

Description The MEM command identifies a new starting address for the block of memory displayed in a MEMORY window. The optional extension number (#) opens an additional MEMORY window, allowing you to view a separate block of memory. The debugger displays the contents of memory at *expression* in the first data position in the MEMORY window. The end of the range is defined by the size of the window. The *expression* can be an absolute address, a symbolic address, or any C expression.

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

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

mi — *Disconnecting I/O Port* — Simulator Only

Syntax **mi** *port address,* {**READ** | **WRITE**}

Menu selection **M**emory→D**i**sConn

Environments ☑ basic debugger ☑ profiling

Description The MI command disconnects a simulated I/O port from its associated system file.

The *port address* parameter identifies the address of the I/O port, which must have been previously defined with the MC command.

mix — *Enter Mixed Mode*

Syntax **mix**

Menu selection Mo**D**e→**M**ixed

Environments ☑ basic debugger ☐ profiling

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 — *List Memory Map*

Syntax **ml**

Menu selection **M**emory→**L**ist

Environments ☑ basic debugger ☑ profiling

Description The ML command lists the memory ranges that are defined for the debugger's memory map. The ML command lists the starting address, ending address, and read/write characteristics of each defined memory range.

move

Move Active Window

Syntax **move** [*X position*, *Y position* [, *width*, *length*]]

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The MOVE command moves the active window to the specified XY position. If you choose, you can resize the window while you move it (see the SIZE command for valid *width* and *length* values). You can use the MOVE command in one of two ways:

- By supplying a specific *X position* and *Y position* or
- By omitting the *X position* and *Y position* parameters and using function keys to interactively move the window.

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. X positions are valid if the X position plus the window width in characters is less than or equal to the screen width in characters. Y positions are valid if the Y position plus the widow height is less than or equal to the screen height in lines.

For example, if the window is 10 characters wide and 5 lines high and the screen size is 80 x 25, the command **move 70, 20** would put the lower right-hand corner of the window in the lower right-hand corner of the screen. No X value greater than 70 or Y value greater than 20 would be valid in this example.

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 ESC or ↵.

mr — *Reset Memory Map*

Syntax **mr**

Menu selection **M**emory→**R**eset

Environments ☑ basic debugger ☑ profiling

Description The MR command resets the debugger's memory map by deleting all defined memory ranges from the map.

ms — *Save Memory Block to File*

Syntax **ms** *address, length, filename*

Menu selection **M**emory→**S**ave

Environments ☑ basic debugger ☑ profiling

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 beginning of the block.
- The *length* parameter defines the length, in words, of the block. This parameter can be any C expression.
- The *filename* is a system file. If you don't supply an extension, the debugger adds an .obj extension.

next — *Single-Step, Next Statement*

Syntax **next** [*expression*]

Menu selection Next=**F10** (in disassembly)

Environments ☑ basic debugger ☐ profiling

Description The NEXT command is similar to the STEP command. If you're in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time. Unlike STEP, NEXT never updates the display when executing called functions; NEXT always steps to the next consecutive statement. Unlike STEP, NEXT steps over function calls rather than stepping into them—you don't see the single-step execution of the function call.

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (the *Running code conditionally* discussion, page 6-17, discusses this in detail).

patch — *Patch Assemble*

Syntax **patch** *address, assembly language instruction*

Menu selection none

Environments ☑ basic debugger ☐ profiling

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*.

pf — *Profile, Full*

Syntax **pf** *starting point* [, *update rate*]

Menu selection **P**rofile→**F**ull

Environments ☐ basic debugger ☑ profiling

Description The PF command initiates a RUN and collects a full set of statistics on the defined areas between the *starting point* and the first-encountered stopping point. The *starting point* parameter can be a label, a function name, or a memory address.

The optional *update rate* parameter determines how often the PROFILE window will be updated. The *update rate* parameter can have one of these values:

Value	Description
0	This is the default. Statistics are not updated until the session is halted (although you can force an update by clicking the mouse in the window header). A “spinning wheel” character is shown to indicate that a profiling session is in progress.
≥**1**	Statistics are updated during the session. A value of **1** means that data is updated as often as possible.
<**0**	Statistics are not updated until the profiling session is halted, and the “spinning wheel” character is not displayed.

pinc — *Connect Pin* — **Simulator Only**

Syntax **pinc** *pinname, filename*

Menu selection **P**in→**C**onnect

Environments ☑ basic debugger ☐ profiling

Description The PINC command connects an input file to interrupt pin.

- The *pinname* parameter identifies the interrupt pin and must be one of the four interrupt pins ($\overline{\text{INT}}0$–$\overline{\text{INT}}3$).
- The *filename* parameter is the name of your input file.

pind — *Disconnect Pin* — **Simulator Only**

Syntax **pind** *pinname*

Menu selection **P**in→**D**isconnect

Environments ☑ basic debugger ☐ profiling

Description The PIND command disconnects an input file from an interrupt pin. The *pinname* parameter identifies the interrupt pin and must be one of the four interrupt pins, ($\overline{\text{INT}}0$–$\overline{\text{INT}}3$).

pinl — *List the Interrupt Pins* — **Simulator Only**

Syntax **pinl**

Menu selection **P**in→**L**ist

Environments ☑ basic debugger ☐ profiling

Description The PINL command displays all of the pins—unconnected pins first, followed by the connected pins. For a connected pin, the simulator displays the name of the pin and the absolute pathname of the file in the COMMAND window.

pq

Profile, Quick

Syntax

pq *starting point* [, *update rate*]

Menu selection

Profile→**Q**uick

Environments

☐ basic debugger ☑ profiling

Description

The PQ command initiates a RUN command and collects a subset of the available statistics on the defined areas between the *starting point* and the first-encountered stopping point. PQ is similar to PF, except that PQ doesn't collect exclusive or exclusive max data.

The *update rate* parameter is the same as for the PF command.

pr

Resume Profile Session

Syntax

pr [*clear data* [, *update rate*]]

Menu selection

Profile→**R**esume

Environments

☐ basic debugger ☑ profiling

Description

The PR command resumes the last profiling session (initiated by PF or PQ), starting from the current program counter.

The optional *clear data* parameter tells the debugger whether or not it should clear out the previously collected data. The *clear data* parameter can have one of these values:

Value	Description
0	This is the default. The profiler will continue to collect data, adding it to the existing data for the profiled areas, and to use the previous internal profile stacks.
nonzero	All previously collected profile data and internal profile stacks are cleared.

The *update rate* parameter is the same as for the PF and PQ commands.

prompt

Change Command-Line Prompt

Syntax

prompt *new prompt*

Menu selection

Color→**P**rompt

Environments

☑ basic debugger ☑ profiling

Description

The PROMPT command changes the command-line prompt. The *new prompt* can be any string of characters (note that a semicolon or comma ends the string).

quit — *Exit Debugger*

Syntax **quit**

Menu selection none

Environments √ basic debugger √ profiling

Description The QUIT command exits the debugger and returns to the operating system.

reload — *Reload Object Code*

Syntax **reload** [*object filename*]

Menu selection **L**oad→**R**eload

Environments √ basic debugger √ profiling

Description The RELOAD command loads only an object file *without* loading its associated symbol table. This is useful for reloading a program when target memory has been corrupted. If you enter the RELOAD command without specifying a filename, the debugger reloads the file that you loaded last.

reset — *Reset Target System*

Syntax **reset**

Menu selection **L**oad→**R**eseT

Environments √ basic debugger √ profiling

Description The RESET command resets the target system (emulator only), simulator, or EVM and reloads the monitor. Note that this is a *software* reset.

restart — *Reset PC to Program Entry Point*

Syntax **restart**
rest

Menu selection **L**oad→R**E**start

Environments √ basic debugger √ profiling

Description The RESTART or REST command resets the program to its entry point. (This assumes that you have already used one of the load commands to load a program into memory.)

return

Return to Function's Caller

Syntax
return
ret

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The RETURN or RET command executes the code in the current C function and halts when execution reaches the caller. Breakpoints do not affect this command, but you can halt execution by pressing the left mouse button or pressing (ESC).

run

Run Code

Syntax **run** [*expression*]

Menu selection Run=**F5**

Environments ☑ basic debugger ☐ profiling

Description The RUN command is the basic command for running an entire program. The command's behavior depends on the type of parameter you supply:

- If you don't supply an *expression*, the program executes until it encounters a breakpoint or until you press the left mouse button or press (ESC).
- If you supply a logical or relational *expression*, this becomes a conditional run (described in detail on page 6-17).
- If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger executes *count* instructions, halts, and updates the display.

runb

Benchmark Code

Syntax **runb**

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The RUNB command executes a specific section of code and counts the number of CPU clock cycles consumed by the execution. In order to operate correctly, *execution must be halted by a software breakpoint*. After RUNB execution halts, the debugger stores the number of cycles into the CLK pseudoregister. For a complete explanation of the RUNB command and the benchmarking process, read Section 6.7, *Benchmarking,* on page 6-19.

runf — *Run Free* — **EVM & Emulator Only**

Syntax **runf**

Menu selection none

Environments ☑ basic debugger ☐ profiling

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

The HALT command stops a RUNF; note that the debugger automatically executes a HALT when the debugger is invoked.

sa — *Add Stopping Point*

Syntax **sa** *address*

Menu selection **S**top-points→**A**dd

Environments ☐ basic debugger ☑ profiling

Description The SA command adds a stopping point at *address*. The *address* can be a label, a function name, or a memory address.

scolor — *Change Screen Colors*

Syntax **scolor** *area name*, $attribute_1$ [, $attribute_2$ [, $attribute_3$ [, $attribute_4$]]]

Menu selection **C**olor→**C**onfig

Environments ☑ basic debugger ☐ profiling

Description The SCOLOR command changes the color of specified areas of the debugger display and updates the display immediately. The *area name* parameter identifies the area of the display that is affected. The *attributes* identify how the area is affected. The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Valid values for the *attribute* parameters include:

black	blue	green	cyan
red	magenta	yellow	white
bright		blink	

Valid values for the *area name* parameters include:

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

You don't have to type an entire *attribute* or *area name*; you need to 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 **C**olor→**L**oad

Environments ☑ basic debugger ☑ profiling

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 the init.clr file. The debugger searches for the specified file in the current directory and then in directories named with the D_DIR environment variable.

sd — *Delete Stopping Point*

Syntax **sd** *address*

Menu selection **S**top-points→**D**elete

Environments ☐ basic debugger ☑ profiling

Description The SD command deletes the stopping point at *address*.

setf — *Set Default Data-Display Format*

Syntax

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

Menu selection

none

Environments

☑ basic debugger ☐ profiling

Description

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

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

char	short	uint	ulong	double
uchar	int	long	float	ptr

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

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

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

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

```
setf *
```

size — *Size Active Window*

Syntax **size** [*width*, *length*]

Menu selection none

Environments ☑ basic debugger ☑ profiling

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. If the window is in the upper left corner of the screen, the maximum size of the window is the same as the screen size minus one line. (The extra line is needed for the menu bar.) For example, if the screen size is 80 characters by 25 lines, the largest window size is 80 characters by 24 lines.

If a window is in the middle of the display, 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 3-23.

If you enter the SIZE command without *width* and *length* parameters, you can use arrow keys to size the window.

↓	Makes the active window one line longer.
↑	Makes the active window one line shorter.
←	Makes the active window one character narrower.
→	Makes the active window one character wider.

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

sl — *List Stopping Point*

Syntax **sl**

Menu selection **S**top-points→**L**ist

Environments ☐ basic debugger ☑ profiling

Description The SL command lists all of the currently set stopping points.

sload — *Load Symbol Table*

Syntax **sload** *object filename*

Menu selection **L**oad→**S**ymbols

Environments ☑ basic debugger ☑ profiling

Description The SLOAD command loads the symbol table of the specified object file. SLOAD is useful in a debugging environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). SLOAD clears the existing symbol table before loading the new one but does not modify memory or set the program entry point. Note that SLOAD closes the WATCH and DISP windows.

sound — *Enable Error Beep*

Syntax **sound** {**on** | **off**}

Menu selection none

Environments ☑ basic debugger ☐ profiling

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

sr — *Reset Stopping Point*

Syntax **sr**

Menu selection **S**top-points→**R**eset

Environments ☐ basic debugger ☑ profiling

Description The SR command resets (deletes) *all* currently set stopping points.

ssave — *Save Screen Configuration*

Syntax **ssave** [*filename*]

Menu selection **C**olor→**S**ave

Environments ☑ basic debugger ☐ profiling

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 — *Single-Step*

Syntax **step** [*expression*]

Menu selection Step=**F8** (in disassembly)

Environments ☑ basic debugger ☐ profiling

Description The STEP command single-steps through assembly language or C code. If you're in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time.

If you're single-stepping through C code and encounter a function call, the STEP command shows you the single-step execution of the called function (assuming that the function was compiled with the compiler's –g debug option). When function execution completes, single-step execution returns to the caller. If the function wasn't compiled with the debug option, the debugger executes the function but doesn't show single-step execution of the function.

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (the *Running code conditionally* discussion, page 6-17, discusses this in detail).

system

Enter DOS Command

Syntax

system [*DOS command* [, *flag*]]

Menu selection

none

Environments

☑ basic debugger ☑ profiling

Description

The SYSTEM command allows you to enter DOS commands without explicitly exiting the debugger environment.

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 DOS command. (In MS-DOS, available memory may limit the commands that you can enter.) When you finish, enter:

```
exit
```

If you prefer, you can supply the DOS 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 *flag* parameter to tell the debugger whether or not it should hesitate after displaying the information. *Flag* may be a 0 or a 1.

0 If you supply a value of 0 for *flag*, the debugger immediately returns to the debugger environment after the last item of information is displayed.

1 If you supply a value of 1 for *flag*, the debugger does not return to the debugger environment until you press . (This is the default.)

take — *Execute Batch File*

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

Menu selection none

Environments ☑ basic debugger ☑ profiling

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 — *Delete Alias Definition*

Syntax **unalias** *alias name*
unalias *****

Menu selection none

Environments ☑ basic debugger ☑ profiling

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 New Directory*

Syntax **use** [*directory name*]

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The USE command allows you to name an additional directory that the debugger can search when looking for source files. You can specify only one directory at a time.

If you enter the USE command without specifying a directory name, the debugger lists all of the current directories.

vaa — *Save All Profile Data to a File*

Syntax **vaa** *filename*

Menu selection **V**iew→S**a**ve→**A**ll views

Environments ☐ basic debugger ☑ profiling

Description The VAA command saves all statistics collected during the current profiling session. The data is stored in a system file.

vac — *Save Currently Displayed Profile Data to a File*

Syntax **vac** *filename*

Menu selection **V**iew→S**a**ve→**C**urrent view

Environments ☐ basic debugger ☑ profiling

Description The VAC command saves all statistics currently displayed in the PROFILE window. (Statistics that aren't displayed aren't saved.) The data is stored in a system file.

version — *Display the Current Debugger Version*

Syntax **version**

Menu selection none

Environments ☑ basic debugger ☑ profiler

Description The VERSION command displays the debugger's copyright date and the current version number of the debugger, silicon, etc.

vr

Reset PROFILE Window Display

Syntax **vr**

Menu selection **V**iew→**R**eset

Environments ☐ basic debugger ☑ profiling

Description The VR command resets the display in the PROFILE window so that all marked areas are listed and statistics are displayed with default labels and in the default sort order.

wa

Add Item to WATCH Window

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

Menu selection **W**atch→**A**dd

Environments ☑ basic debugger ☐ profiling

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. It's most useful to watch an expression whose value changes over time; constant expressions serve no useful function in the watch window. The *label* parameter is optional. When used, it provides a label for the watched entry. If you 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	**o**	Octal
c	ASCII character (bytes)	**p**	Valid address
d	Decimal	**s**	ASCII string
e	Exponential floating point	**u**	Unsigned decimal
f	Decimal floating point	**x**	Hexadecimal

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

```
wa PC,,d
```

wd

Delete Item From WATCH Window

Syntax **wd** *index number*

Menu selection **W**atch→**D**elete

Environments ☑ basic debugger ☐ profiling

Description The WD command deletes a specific item from the WATCH window. The WD command's *index number* parameter must correspond to one of the watch indexes listed in the WATCH window.

whatis

Find Data Type

Syntax **whatis** *symbol*

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The WHATIS command shows the data type of *symbol* in the COMMAND window display area. The *symbol* can be any variable (local, global, or static), a function name, structure tag, typedef name, or enumeration constant.

win

Select Active Window

Syntax **win** *WINDOW NAME*

Menu selection none

Environments ☑ basic debugger ☑ profiling

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 to specify only enough letters to identify the window.

If several of the same types of window are visible on the screen, don't use the WIN command to select one of them. If you supply an ambiguous name (such as C, which could stand for CPU or CALLS), the debugger selects the first window it finds whose name matches the name you supplied. If the debugger doesn't find the window you asked for (because you closed the window or misspelled the name), then the WIN command has no effect.

wr — *Reset WATCH Window*

Syntax **wr**

Menu selection **W**atch→**R**eset

Environments ☑ basic debugger ☐ profiling

Description The WR command deletes all items from the WATCH window and closes the window.

zoom — *Zoom Active Window*

Syntax **zoom**

Menu selection none

Environments ☑ basic debugger ☑ profiling

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 Profiling Commands

The following tables summarize the profiling commands that are used for marking, enabling, disabling, and unmarking areas and for changing the display in the PROFILE window. These commands are easiest to use from the pulldown menus, so they are not included in the alphabetical command summary. The syntaxes for these commands are provided here so that you can include them in batch files.

Table 11–1. Marking Areas

To mark this area	C only	Disassembly only
Lines		
❑ By line number, address	**MCLE** *filename*, *line number*	**MALE** *address*
❑ All lines in a function	**MCLF** *function*	**MALF** *function*
Ranges		
❑ By line numbers	**MCRE** *filename*, *line number*, *line number*	**MARE** *address*, *address*
Functions		
❑ By function name	**MCFE** *function*	not applicable
❑ All functions in a module	**MCFM** *filename*	
❑ All functions everywhere	**MCFG**	

Table 11–2. Disabling Marked Areas

To disable this area	C only	Disassembly only	C *and* disassembly
Lines			
❑ By line number, address	**DCLE** *filename*, *line number*	**DALE** *address*	not applicable
❑ All lines in a function	**DCLF** *function*	**DALF** *function*	**DBLF** *function*
❑ All lines in a module	**DCLM** *filename*	**DALM** *filename*	**DBLM** *filename*
❑ All lines everywhere	**DCLG**	**DALG**	**DBLG**
Ranges			
❑ By line number, address	**DCRE** *filename*, *line number*	**DARE** *address*	not applicable
❑ All ranges in a function	**DCRF** *function*	**DARF** *function*	**DBRF** *function*
❑ All ranges in a module	**DCRM** *filename*	**DARM** *filename*	**DBRM** *filename*
❑ All ranges everywhere	**DCRG**	**DARG**	**DBRG**

Table 11–2. Disabling Marked Areas (Continued)

To disable this area	C only	Disassembly only	C *and* disassembly
Functions			
❑ By function name	**DCFE** *function*	not applicable	not applicable
❑ All functions in a module	**DCFM** *filename*		**DBFM** *filename*
❑ All functions everywhere	**DCFG**		**DBFG**
All areas			
❑ All areas in a function	**DCAF** *function*	**DAAF** *function*	**DBAF** *function*
❑ All areas in a module	**DCAM** *filename*	**DAAM** *filename*	**DBAM** *filename*
❑ All areas everywhere	**DCAG**	**DAAG**	**DBAG**

Table 11–3. Enabling Disabled Areas

To enable this area	C only	Disassembly only	C *and* disassembly
Lines			
❑ By line number, address	**ECLE** *filename*, *line number*	**EALE** *address*	not applicable
❑ All lines in a function	**ECLF** *function*	**EALF** *function*	**EBLF** *function*
❑ All lines in a module	**ECLM** *filename*	**EALM** *filename*	**EBLM** *filename*
❑ All lines everywhere	**ECLG**	**EALG**	**EBLG**
Ranges			
❑ By line number, address	**ECRE** *filename*, *line number*	**EARE** *address*	not applicable
❑ All ranges in a function	**ECRF** *function*	**EARF** *function*	**EBRF** *function*
❑ All ranges in a module	**ECRM** *filename*	**EARM** *filename*	**EBRM** *filename*
❑ All ranges everywhere	**ECRG**	**EARG**	**EBRG**
Functions			
❑ By function name	**ECFE** *function*	not applicable	not applicable
❑ All functions in a module	**ECFM** *filename*		**EBFM** *filename*
❑ All functions everywhere	**ECFG**		**EBFG**
All areas			
❑ All areas in a function	**ECAF** *function*	**EAAF** *function*	**EBAF** *function*
❑ All areas in a module	**ECAM** *filename*	**EAAM** *filename*	**EBAM** *filename*
❑ All areas everywhere	**ECAG**	**EAAG**	**EBAG**

Table 11–4. Unmarking Areas

To unmark this area	C only	Disassembly only	C *and* disassembly
Lines			
❑ By line number, address	**UCLE** *filename*, *line number*	**UALE** *address*	not applicable
❑ All lines in a function	**UCLF** *function*	**UALF** *function*	**UBLF** *function*
❑ All lines in a module	**UCLM** *filename*	**UALM** *filename*	**UBLM** *filename*
❑ All lines everywhere	**UCLG**	**UALG**	**UBLG**
Ranges			
❑ By line number, address	**UCRE** *filename*, *line number*	**UARE** *address*	not applicable
❑ All ranges in a function	**UCRF** *function*	**UARF** *function*	**UBRF** *function*
❑ All ranges in a module	**UCRM** *filename*	**UARM** *filename*	**UBRM** *filename*
❑ All ranges everywhere	**UCRG**	**UARG**	**UBRG**
Functions			
❑ By function name	**UCFE** *function*	not applicable	not applicable
❑ All functions in a module	**UCFM** *filename*		**UBFM** *filename*
❑ All functions everywhere	**UCFG**		**UBFG**
All areas			
❑ All areas in a function	**UCAF** *function*	**UAAF** *function*	**UBAF** *function*
❑ All areas in a module	**UCAM** *filename*	**UAAM** *filename*	**UBAM** *filename*
❑ All areas everywhere	**UCAG**	**UAAG**	**UBAG**

Table 11–5. Changing the PROFILE Window Display

(a) Viewing specific areas

To view this area	C only	Disassembly only	C *and* disassembly
Lines			
❑ By line number, address	**VFCLE** *filename*, *line number*	**VFALE** *address*	not applicable
❑ All lines in a function	**VFCLF** *function*	**VFALF** *function*	**VFBLF** *function*
❑ All lines in a module	**VFCLM** *filename*	**VFALM** *filename*	**VFBLM** *filename*
❑ All lines everywhere	**VFCLG**	**VFALG**	**VFBLG**
Ranges			
❑ By line number, address	**VFCRE** *filename*, *line number*	**VFARE** *address*	not applicable
❑ All ranges in a function	**VFCRF** *function*	**VFARF** *function*	**VFBRF** *function*
❑ All ranges in a module	**VFCRM** *filename*	**VFARM** *filename*	**VFBRM** *filename*
❑ All ranges everywhere	**VFCRG**	**VFARG**	**VFBRG**

Table 11–5. Changing the PROFILE Window Display (Continued)

To view this area	C only	Disassembly only	C *and* disassembly
Functions			
❑ By function name	**VFCFE** *function*	not applicable	not applicable
❑ All functions in a module	**VFCFM** *filename*		**VFBFM** *filename*
❑ All functions everywhere	**VFCFG**		**VFBFG**
All areas			
❑ All areas in a function	**VFCAF** *function*	**VFAAF** *function*	**VFBAF** *function*
❑ All areas in a module	**VFCAM** *filename*	**VFAAM** *filename*	**VFBAM** *filename*
❑ All areas everywhere	**VFCAG**	**VFAAG**	**VFBAG**

(b) Viewing different data

To view this information	Use this command
Count	**VDC**
Inclusive	**VDI**
Inclusive, maximum	**VDN**
Exclusive	**VDE**
Exclusive, maximum	**VDX**
Address	**VDA**
All	**VDL**

(c) Sorting the data

To sort on this data	Use this command
Count	**VSC**
Inclusive	**VSI**
Inclusive, maximum	**VSN**
Exclusive	**VSE**
Exclusive, maximum	**VSX**
Address	**VSA**
Data	**VSD**

11.5 Summary of Special Keys

The debugger provides function key, cursor key, and command key sequences for performing a variety of actions:

- Editing text on the command line
- Using the command history
- Switching modes
- Halting or escaping from an action
- Displaying the pulldown menus
- Running code
- Selecting or closing a window
- Moving or sizing a window
- Scrolling through a window's contents
- Editing data or selecting the active field

Editing text on the command line

To do this	Use these function keys
Enter the current command (note that if you press the return key in the middle of text, the debugger truncates the input text at the point where you press this key)	
Move back over text without erasing characters	CONTROL H or BACK SPACE
Move forward through text without erasing characters	CONTROL L
Move back over text while erasing characters	DELETE
Move forward through text while erasing characters	SPACE
Insert text into the characters that are already on the command line	INSERT

Using the command history

To do this	Use these function keys
Repeat the last command that you entered	F2
Move backward, one command at a time, through the command history	TAB
Move forward, one command at a time, through the command history	SHIFT TAB

Switching modes

To do this	Use this function key
Switch debugging modes in this order: → auto → assembly → mixed →	F3

Halting or escaping from an action

The escape key acts as an end or undo key in several situations.

To do this	Use this function key
❑ Halt program execution ❑ Close a pulldown menu ❑ Undo an edit of the active field in a data-display window (pressing this key leaves the field unchanged) ❑ Halt the display of a long list of data in the display area of the COMMAND window	ESC

Displaying pulldown menus

To do this	Use these function keys
Display the Load menu	ALT L
Display the Break menu	ALT B
Display the Watch menu	ALT W
Display the Memory menu	ALT M
Display the Color menu	ALT C
Display the MoDe menu	ALT D
Display the Pin menu	ALT P
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 *expression* parameter)	F5
Single-step code from the current PC (equivalent to the STEP command without an *expression* parameter)	F8
Single-step code from the current PC; step over function calls (equivalent to the NEXT command without an *expression* parameter)	F10

Selecting or closing a window

To do this	Use these function keys
Select the active window (pressing this key makes each window active in turn; stop pressing the key when the desired window becomes active)	F6
Close the CALLS, WATCH, DISP, or additional MEMORY window (the window must be active before you can close it)	F4

Moving or sizing a window

You can use the arrow keys to interactively move a window after entering the MOVE or SIZE command without parameters.

To do this	Use these function keys
❑ Move the window down one line ❑ Make the window one line longer	↓
❑ Move the window up one line ❑ Make the window one line shorter	↑
❑ Move the window left one character position ❑ Make the window one character narrower	←
❑ Move the window right one character position ❑ Make the window one character wider	→

Scrolling 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	PAGE UP
Scroll down through the window contents, one window length at a time	PAGE DOWN
Move the field cursor up, one line at a time	↑
Move the field cursor down, one line at a time	↓
❑ *FILE window only:* Scroll left 8 characters at a time ❑ *Other windows:* 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	←
❑ *FILE window only:* Scroll right 8 characters at a time ❑ *Other windows:* Move the field cursor right 1 field; at the last field on a line, wrap around to the first field on the next line	→
FILE window only: Adjust the window's contents so that the first line of the text file is at the top of the window	HOME
FILE window only: Adjust the window's contents so that the last line of the text file is at the bottom of the window	END
DISP windows only: Scroll up through an array of structures	CONTROL PAGE UP
DISP windows only: Scroll down through an array of structures	CONTROL PAGE DOWN

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 these function keys
❑ *FILE or DISASSEMBLY window:* Set or clear a breakpoint ❑ *CALLS window:* Display the source to a listed function ❑ *Any data-display window:* Edit the contents of the current field ❑ *DISP window:* Open an additional DISP window to display a member that is an array, structure, or pointer	F9

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.

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 be familiar with the rules governing C expressions. You should obtain a copy of ***The C Programming Language*** (first or second edition) by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey. This book is referred to in the C community, and in Texas Instruments documentation, as **K&R.**

Note:

A single value or symbol is a legal C expression.

K&R contains a complete description of C expressions; to get you started, here'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 Using Expression Analysis in the Debugger

The debugger's expression analysis is based on C expression analysis. This includes all mathematical, relational, pointer, and assignment operators. However, there are a few limitations, as well as a few additional features not described in K&R C.

Restrictions

The following restrictions apply to the debugger's expression analysis features.

- The size of operator is not supported.
- The comma operator (,) is not supported (commas are used to separate parameter values for the debugger commands).
- Function calls and string constants are currently not supported in expressions.
- The debugger supports a limited number of type casts; the following forms are allowed:

  (*basic type* **)**
  (*basic type* ***** *...***)**
  ([*structure/union/enum***]** *structure/union/enum tag* **)**
  ([*structure/union/enum***]** *structure/union/enum tag* ***** *...* **)**

  Note that you can use up to six *****s in a cast.

Additional features

- All floating-point operations are performed in double precision using standard widening. (This is transparent.) Floats are represented in IEEE floating-point format.
- All registers can be referenced by name. The 'C3x's extended-precision registers (R0–R7) are treated as integers. You can use the names F0–F7 to access the registers as floating-point values.
- Void expressions are legal (treated like integers).
- The specification of variables and functions can be qualified with context information. Local variables (including local statics) can be referenced with the expression form:

  *function name***.***local name*

This expression format is useful for examining the automatic variables of a function that is not currently being executed. Unless the variable is static, however, the function must be somewhere in the current call stack. If you want to see local variables from the currently executing function, you need not use this form; you can simply specify the variable name (just as in your C source).

File-scoped variables (such as statics or functions) can be referenced with the following expression form:

*filename***.***function name*
or *filename***.***variable name*

This expression format is useful for accessing a file-scoped static variable (or function) that may share its name with variables in other files.

In this expression, *filename* **does not include** the file extension; the debugger searches the object symbol table for any source filename that matches the input name, disregarding any extension. Thus, if the variable *ABC* is in file *source.c*, you can specify it as *source.ABC*.

These expression forms can be combined into an expression of the form:

*filename***.***function name***.***variable name*

- Any integral or void expression can be treated as a pointer and used with the indirection operator (*). Here are several examples of valid use of a pointer in an expression:

  ```
  *123
  *R5
  *(R2 + 123)
  *(I*J)
  ```

  By default, the values are treated as integers (that is, these expressions point to integer values).

- Any expression can be typecast to a pointer to a specific type (overriding the default of pointing to an integer, as described above).

  Hint: You can use casting with the WA and DISP commands to display data in a desired format.

  For example, the expression:

  ```
  *(float *)10
  ```

  treats 10 as a pointer to a floating-point value at location 10 in memory. In this case, the debugger fetches the contents of memory location 10 and treats the contents as a floating-point value. If you use this expression as a parameter for the DISP command, the debugger displays memory contents as an array of floating-point values within the DISP window, beginning with memory location 10 as array member [0].

Note how the first expression differs from the expression:

```
(float)*10
```

In this case, the debugger fetches an integer from address 10 and converts the integer to a floating-point value.

You can also typecast to user-defined types such as structures. For example, in the expression:

```
((struct STR *)10)–>field
```

the debugger treats memory location 10 as a pointer to a structure of type STR (assuming that a structure is at address 10) and accesses a field from that structure.

Appendix A

Specifications for Your Target System’s Connection to the Emulator

This appendix contains information about connecting your target system with the emulator.

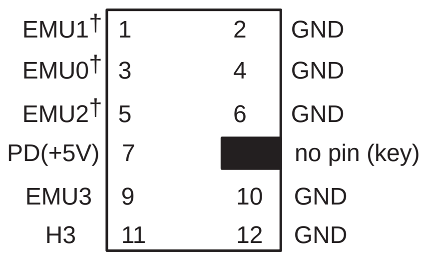
Topic **Page**

A.1 Designing Your Target System's Emulator Connector (12-Pin Header)

The 'C3x uses a revolutionary technology to allow complete emulation via a serial scan path of the 'C3x. To perform realtime emulation, **your target system must have a 12-pin header** (2 rows of 6 pins) with the connections that are shown in Figure A–1.

To use the target cable, supply the signals shown in Figure A–1 to a 12-pin header (two rows of six pins) with pin 8 cut out to provide keying.

Figure A–1. 12-Pin Header Signals and Header Dimensions



Header Dimensions:
Pin-to-pin spacing, 0.100 in. (X,Y)
Pin width, 0.025-in. square post
Pin length, 0.235-in. nominal

Use a BergStik II header or equivalent.

† These signals should always be pulled up with separate 20-kΩ resistors to +5 volts on the 'C3x.

Table A–1. 12-Pin Header Signal Description and Pin Numbers

Signal	Description	'C30 Pin Number	'C31 Pin Number
EMU0	Emulation pin 0	F14	124
EMU1	Emulation pin 1	E15	125
EMU2	Emulation pin 2	F13	126
EMU3	Emulation pin 3	E14	123
H3	'C3x H3	A1	82
PD	Presence detect. Indicates that the cable is connected and target system is powered up. PD should be tied to +5 volts in the target system.		

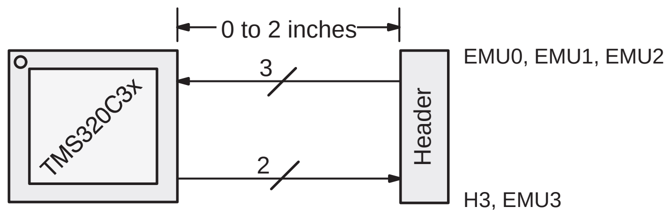
Although you can use other headers, recommended parts include:

straight header, unshrouded	DuPont Connector Systems part number 67996–112
right-angle header, unshrouded	DuPont Connector Systems part number 68405–112
right-angle header, 4-wall shrouded	AMP, Incorporated part number 103167–3 or part number 103166–4

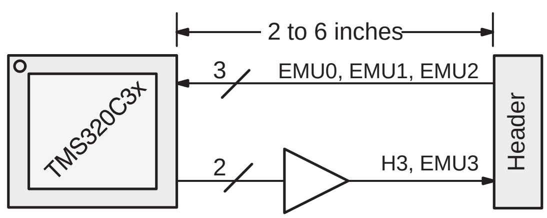
A.2 Buffering Signals Between the Emulator and the Target System

It is extremely important to provide high-quality signals between the emulator and the ’C3x on the target system. In many cases, the signal must be buffered to produce a high-quality signal. The need for signal buffering and placement of the emulation header can be divided into 3 categories:

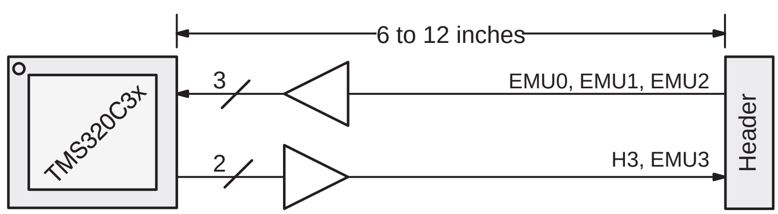
- **No signal buffering.** In this situation, the distance between the header and the ’C3x should be no more than 2 inches.



- **Buffered transmission signals.** In this situation, the distance between the emulation header and the ’C3x is greater than 2 inches but less than 6 inches. The transmission signals—H3 and EMU3—are buffered through the same package.



- **All signals buffered.** The distance between the emulation header and the ’C3x is greater than 6 inches but less than 12 inches. All ’C3x emulation signals—EMU0, EMU1, EMU2, and EMU3—are buffered through the same package.



A.3 Buffer Delays

The emulator is designed to support a TMS320C3x with H3 clock periods down to 40 ns. Table A–2 lists the maximum buffer delay for various H3 periods. The buffer is noninverting.

Table A–2. Maximum Buffer Delays

H3 Period	Maximum Buffer Delay
60 ns	8 ns
50 ns	6 ns
40 ns	4 ns

The distance between the ’C3x and the buffers depends on the printed-wire-board layout and loading on H3. However, Texas Instruments suggests that the distance be as short as possible and less than 4 inches.

When you buffer H3, don’t place another device between the buffer output and the header (see Figure A–2). Connecting another device to this signal could cause false triggering of the device due to cable reflections.

Figure A–2. H3 Buffer Restrictions

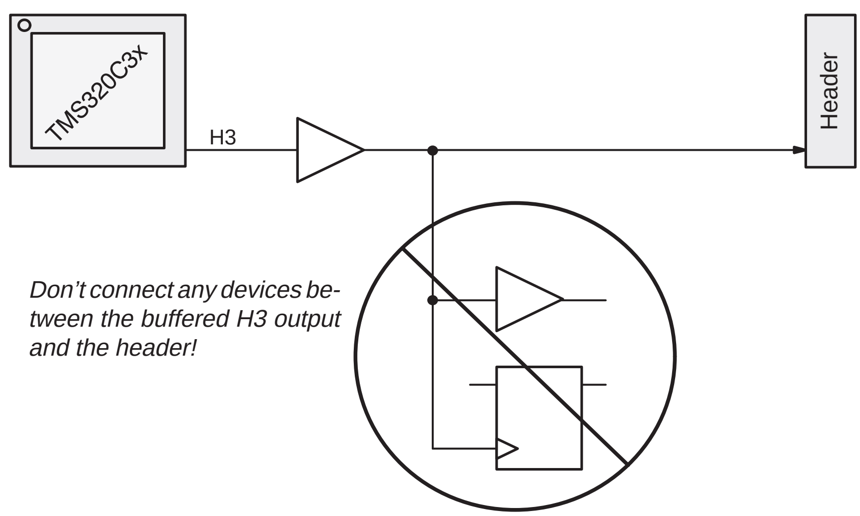
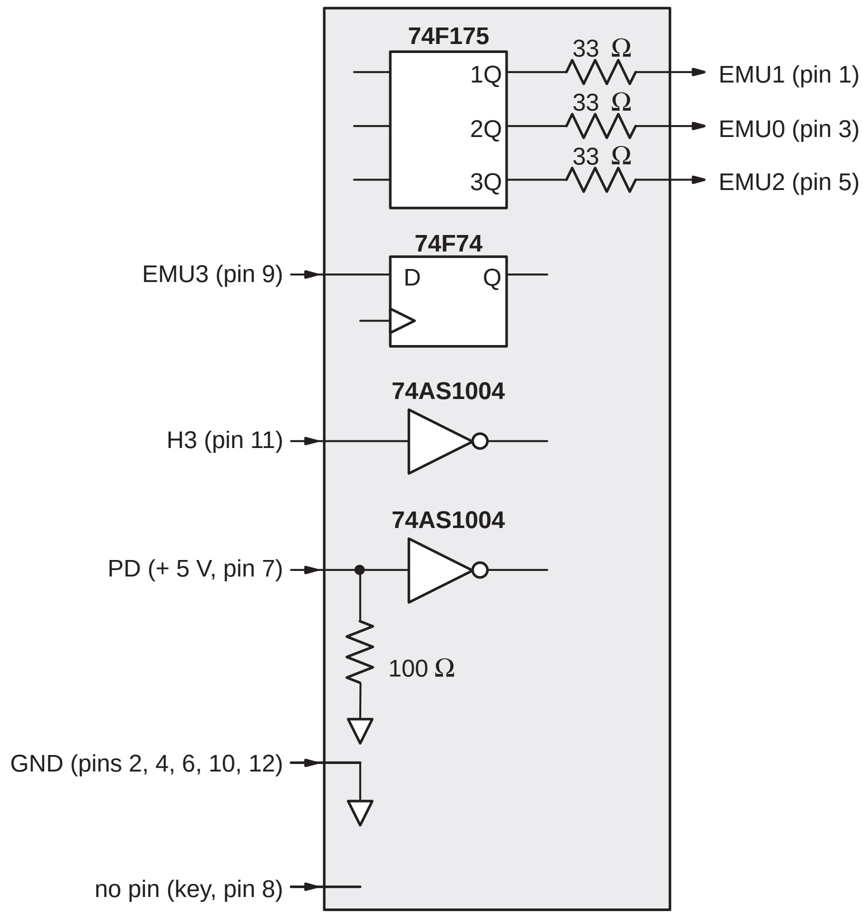


Figure A–3 shows a portion of logic in the emulator pod. Note that 33-Ω resistors are added to EMU0, EMU1, and EMU2; this minimizes cable reflections.

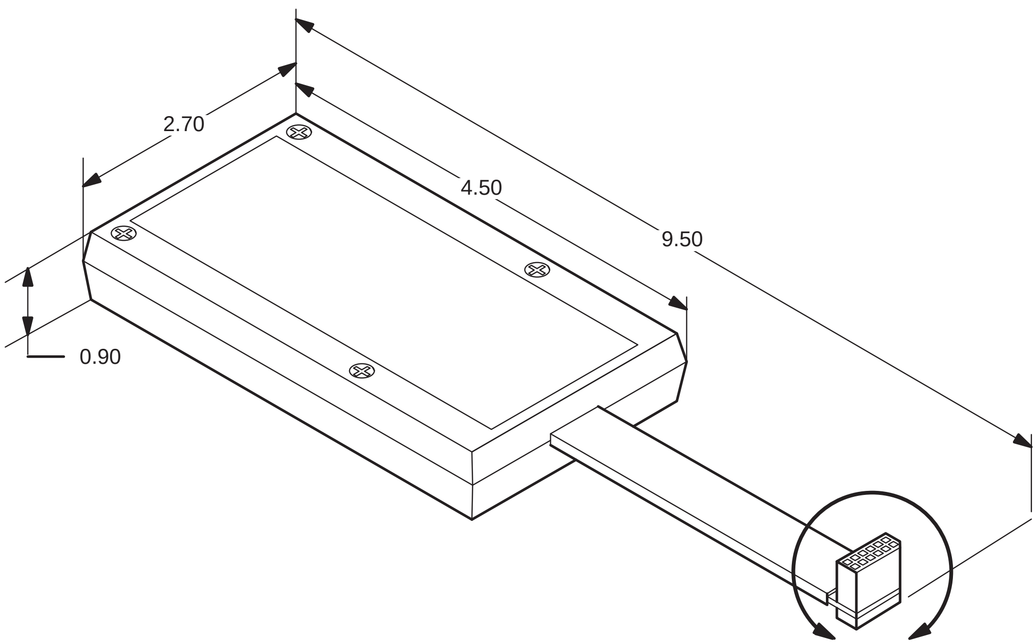
Figure A–3. Emulator Pod Interface



A.4 Mechanical Dimensions for the 12-Pin Emulator Connector

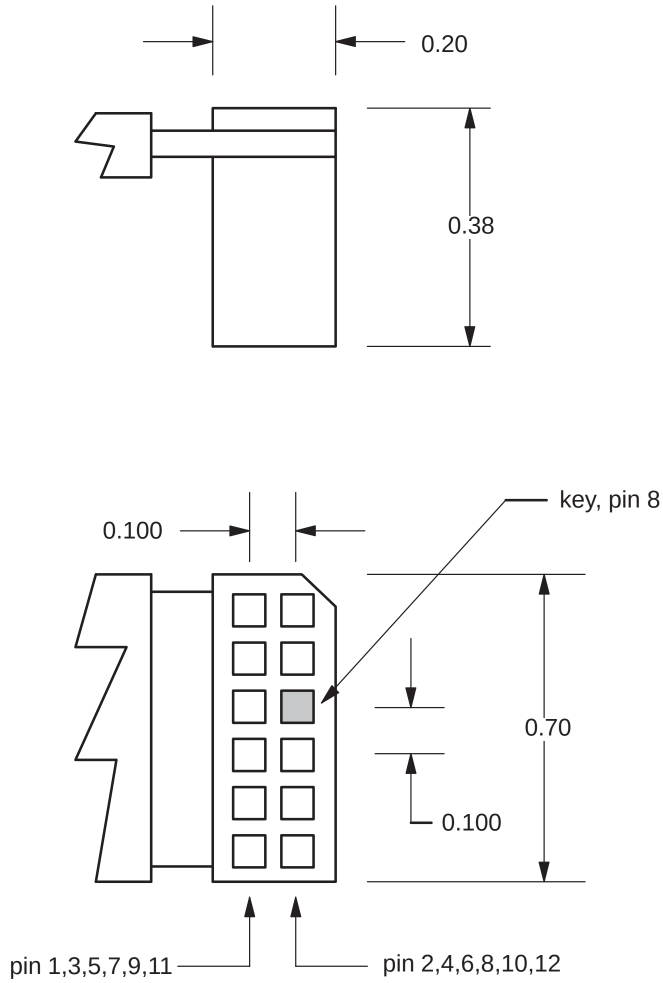
The ’C3x emulator target cable consists of a 3-foot section of jacketed cable, an active cable pod, and a short section of jacketed cable that connects to the target system. The overall cable length is approximately 3 feet,10 inches. Figure A–4 and Figure A–5 show the mechanical dimensions for the target cable pod and short cable. Note that the pin-to-pin spacing on the connector is 0.100 inches in both the X and Y planes. The cable pod box is nonconductive plastic with 4 recessed metal screws.

Figure A–4. Pod/Connector Dimensions



Note: All dimensions are in inches and are nominal dimensions, unless otherwise specified.

Figure A–5. 12-Pin Connector Dimensions



Note: All dimensions are in inches and are nominal dimensions, unless otherwise specified.

Appendix B

Constraints When Using the Emulator

This appendix covers constraints concerning cache control when you are using certain commands and restraints for software breakpoint and execution commands. This information applies only when you are using the debugger with the emulator.

B.1 Cache Interaction With Software Breakpoint Commands

This section explains how cache control works with the software breakpoint commands as discussed in Chapter 8.

- **BA command**

  When the breakpoint address equals a cache instruction address, the cache p-flags are modified according to the following conditions:

Cache Control		
Cache Enable	**Cache Freeze**	**Description**
0	0	Place SWI in memory, no cache modification.
0	1	Place SWI in memory, no cache modification.
1	0	Place SWI in memory. If breakpoint address equals cache address and p-flag is set, the p-flag clears at the corresponding cache address.
1	1	Place SWI in memory.

  Clearing the flag in the third case ensures that the SWI will always be executed, whether from cache or memory.

- **BD and BR commands**

  When the breakpoint address equals a cache instruction address, the cache p-flags are modified according to the following conditions:

Cache Control		
Cache Enable	**Cache Freeze**	**Description**
0	0	Restores instruction to memory, no cache modification.
0	1	Restores instruction to memory, no cache modification.
1	0	Restores instruction to memory. If breakpoint address equals cache address and p-flag is set, the p-flag clears at the corresponding cache address.
1	1	Restores instruction in memory.

B.2 Cache Control for Memory Commands

This section explains how the cache control works with the memory modification commands discussed in Chapter 5.

When a memory modify address is equal to a cache control address, the cache p-flags are modified according to the following conditions:

Cache Control		
Cache Enable	**Cache Freeze**	**Description**
0	0	No cache modification.
0	1	No cache modification.
1	0	Clears p-flag.
1	1	No cache modification.

Clearing the p-flag in the third case ensures that the emulator executes the most current instruction.

B.3 Command Constraints

The following section discusses constraints that apply to software breakpoint and run commands and gives a correct (valid) and an incorrect (not valid) programming example for each rule.

This section also describes constraints imposed when the target system is in a reset condition.

Software breakpoint constraints

- There must be a minimum of three instructions between a **delayed branch** and a breakpoint.

```
                Valid                                  Not Valid

          BRD     TEST                           BRD     TEST
          LDI     0,R0                           LDI     0,R0
          LDI     1,R1               >           LDI     1,R1
          LDI     2,R2                           LDI     2,R2
>         LDI     3,R3                           LDI     3,R3
```

- Do not place a breakpoint on the **repeat single instruction** or the instruction to be repeated.

```
                Valid                                  Not Valid

          RPTS    5                  >           RPTS    5
          LDI     0,R0               >           LDI     0,R0
>         LDI     1,R1                           LDI     1,R1
```

- Do not place a breakpoint on the last instruction of a **repeat block**.

```
                Valid                                  Not Valid

          RPTB    TEST                           RPTB    TEST
          LDI     0,R0                           LDI     0,R0
>         LDI     1,R1                           LDI     1,R1
TEST:     LDI     2,R2               > TEST:     LDI     2,R2
>         LDI     3,R3                           LDI     3,R3
```

Single-step constraints with repeated instructions

The repeat single (RPTS) instruction is an indivisible instruction and cannot be single-stepped. However, the RPTS instruction can be replaced with the repeat block (RPTB) instruction with a block size of one.

Example 1:

```
        RPTS    10
        STI     R0,*AR0++
```

Example 2:

```
        LDI     10,RC
        RTPB    ONE
ONE:    STI     R0,*AR0++
```

Both instruction sequence examples perform the same function. However, the second example can be single-stepped to trace the execution.

Constraints imposed when emulator is reset

When the target system is in the reset condition or when the ’C3x $\overline{\text{RESET}}$ pin is held low, the emulator can still read and write to target memory. Under this condition, the ’C3x memory interface signals will become active. This may cause problems in systems that use the ’C3x $\overline{\text{RESET}}$ signal to put the memory interface in a 3-state condition.

The ’C3x $\overline{\text{HOLD}}$ signal should be used to put the primary bus in a 3-state condition. If the expansion bus is required to remain in the 3-state condition, it cannot be put in a 3-state condition with the $\overline{\text{HOLD}}$ signal and should not be accessed when the ’C3x is in the reset state.

Appendix C

Troubleshooting When Using the Emulator

This appendix answers frequently asked questions about the ’C3x emulator. For other questions about the emulator, call the DSP hotline at (713) 274-2320.

Q *Why does the CLK register on my emulator always read 0?*

A The CLK register is updated only by the RUNB (run benchmark) command (described on page 6-19). Other run commands set the CLK register to 0.

Note: CLK Register Operation

The ’C3x emulator CLK register operates differently than the CLK register for the ’C3x simulator.

Q *Can I get a pipeline status with my ’C3x emulator?*

A No. The emulator does not provide a pipeline status, because it halts only on instruction boundaries with the pipeline flushed. All instructions in the pipeline are guaranteed to be executed when the emulator issues a halt command to the ’C3x.

Q *I have executed the RESET command on my ’C3x emulator and attempted to run code. Why does the PC remain unchanged and still contain my RESET vector?*

A The ’C3x device $\overline{\text{RESET}}$ signal is still at a logic 0. If you attempt to execute code, the PC register remains unchanged, and the SP register increments. If you are using the application board, you must execute the emurst.exe file in order to take the ’C3x device $\overline{\text{RESET}}$ signal to a logic 1. If you are operating the emulator with your own target system, you must set the ’C3x device’s reset signal to a logic 1 to run code.

Q *Does the 'C3x emulator show the last instruction executed or the next instruction to be executed?*

A The emulator always shows the next instruction to be executed. All previous instructions have completed before the emulator halts.

Q *Can I display or directly modify the 'C3x cache?*

A No. The cache is not accessible. However, the emulator keeps the program memory and cache coherent by manipulating the appropriate p-flags.

Q *Does DMA continue to operate when the 'C3x is halted?*

A No. The DMA finishes its current memory cycle and halts. The DMA picks up where it left off when the processor starts running again.

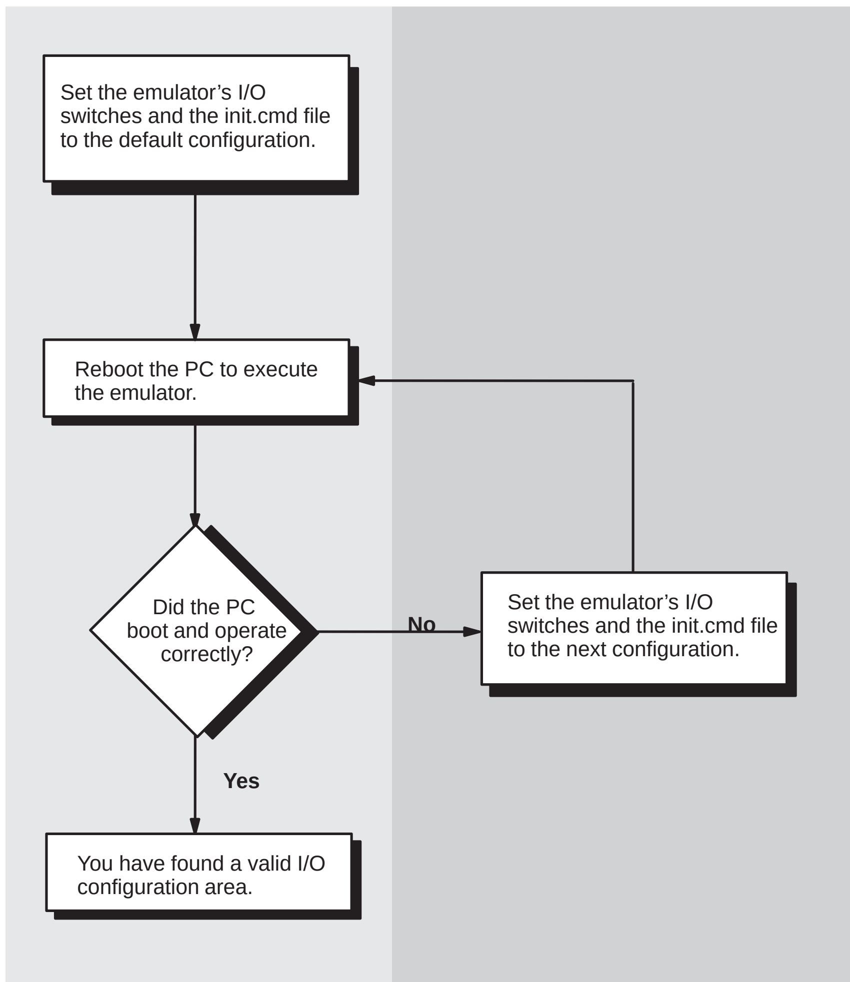
Q *When the 'C3x halts, can other devices gain access to the parallel bus?*

A Yes. When the 'C3x is halted, the $\overline{\text{HOLD}}$ and $\overline{\text{HOLDA}}$ signals continue to function. If you attempt to perform an external memory access via the emulator while the 'C3x is in the $\overline{\text{HOLD}}$ state, you may get a memory error or reduced emulator performance. The emulator always attempts to gain access to the external memory bus. When an attempt fails, the emulator begins a retry and time-out sequence.

Q *I cannot determine or find the I/O address requirements of my PC in any of my product or PC documentation. How can I figure out where to map my 'C3x emulator?*

A The following procedure works but should be used only as a last resort because it may cause I/O bus conflicts if the emulator and another card are mapped to the same I/O address.

Find an open location in the PC I/O map:



The following questions and answers pertain to the interfacing of the ’C3x emulator and the ’C3x application board or to the modification of the ’C3x application board.

Q *I have purchased the ’C3x XDS1000 Development Environment. Must the ’C3x emulator and the ’C3x application board be installed in the same host system?*

A No. The emulator and the application board may be in different host systems. In fact, if you are trying to debug code on the application board and the host at the same time, it is preferable to use two systems because DOS is not a multitasking environment.

Q *I have written a small loader program for the ’C3x application board to load data from the host through the dual port RAM. How can I start to execute the program on the application board and test my host program?*

A There are two methods;

- Perform an xreset command followed by an emurst command to disable the emulator and reset the application board. If your loader program is initiated from reset, this method will work.
- The second and preferred method is to load the debugger, enter a RESET command to initiate the debugger, and then enter a RUNF command. The RUNF command will start to execute the ’C3x at the current address of the program counter.

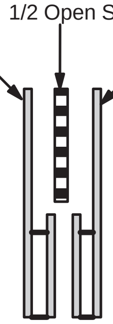
  To suspend the debugger interface, use the SYSTEM command. The SYSTEM command allows you to enter operating-system commands. To re-enter the debugger, type exit. Now that you are back in the debugger interface, you can use the HALT command to resume the program from the point of suspension, or you can type RESET to start over.

Q *When accessing the dual port RAM on the ’C3x application board via the ’C3x emulator, I read* trash *on the upper bits of the data bus. Why?*

A The dual port RAM on the application board is only 8 bits wide. The upper 24 bits of the data bus are left floating. Thus, their value is undetermined on read cycles. The emulator does not mask off the unused data bits before displaying memory.

Q *I have purchased the 'C3x XDS1000 Development Environment. Does this system require 1.5 or 3 slots in my PC?*

A The development environment requires 3 slots in your PC. However, if the emulator is installed in front of the application board, then half of a slot is open between the two boards.



Top View

Q *How can I change the 'C3x vectors on the 'C3x application board?*

A There are two methods:

- One way is to replace the supplied EPROMs with your own EPROMs.
- The second way is to set the applications board MSWAP bit to a logic 1. This causes the EPROM and SRAM to swap address ranges. Modify the SRAM to set up a different set of vectors.

Note:

- ❑ The MSWAP bit is located at address 805FF7h, bit 7.
- ❑ The MSWAP bit is cleared to a logic 0 when the application board is reset via the emurst.exe.
- ❑ The emulator does not clear the MSWAP bit when executing the RESET command.

Q *I want to write my own reset/initialization routine for the 'C3x application board. Are there any special requirements?*

A Yes. Set the memory ports to use external wait states and set the block size to the default of 256 words. The file c30exam.asm, included with the application board software package, contains an example setup.

Q *I am using the examples in the TMS320 Family Floating-Point DSP Optimizing C Compiler User's Guide to write C code for the 'C3x application board. When I try to load my program using the 'C3x emulator, I get a reserved peripheral error message.*

A The most likely problem is that the memory map used in the *TMS320 Family Floating-Point DSP Optimizing C Compiler User's Guide* is not compatible with the applications board. Included with the application board is a memory map template for the program c30exam.asm. The template filename is c30exam.cmd; use it as an example and modify it to meet your needs.

Q *I get the error message CANNOT INITIALIZE TARGET SYSTEM when I try to execute emu3x or evm3x.*

A The port address of the emulator or EVM was not been specified correctly when you started the task. Verify the switch settings in your hardware (refer to your installation guide) and then verify that you have used the correct port address option when you invoked the debugger. (Also check to see whether you specified a different address with the environment variable D_OPTIONS.)

If you continue to have the error, verify that your target system is powered up and that the emulator connector is properly connected. If you are bringing up the target hardware for the first time, verify that the correct signals are active from the 'C3x to the emulator connector. You should check to see if the EMU4/$\overline{\text{SHZ}}$ pin on the 'C3x device is pulled high.

Appendix D

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. (For more information on the environment variables mentioned below, refer to the appropriate installation guide.)

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 called *init.cmd*. If the debugger finds this file, it reads and executes the commands.

7) Loads any object filenames specified with D_OPTIONS or specified on the command line during invocation.

8) Determines the initial mode (auto, assembly, or mixed) and displays the appropriate windows on the screen.

At this point, the debugger is ready to process any commands that you enter.

Appendix E

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.

E.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.

E.2 Alphabetical Summary of Debugger Messages

Symbols

']' expected

Description This is an expression error—it means that the parameter contained an opening "**[**" but didn't contain a closing "**]**".

Action See Section E.3 (page E-20).

')' expected

Description This is an expression error—it means that the parameter contained an opening "**(**" but didn't contain a closing "**)**".

Action See Section E.3 (page E-20).

A

Aborted by user

Description The debugger halted a long COMMAND display listing (from WHATIS, DIR, ML, or BL) because you pressed the ESC key.

Action None required; this is normal debugger behavior.

B

Breakpoint already exists at *address*

Description During single-step execution, the debugger attempted to set a breakpoint where one already existed. (This isn't necessarily a breakpoint that you set—it may have been an internal breakpoint that was used for single-stepping).

Action None should be required; you may want to reset the program entry point (RESTART) and re-enter the single-step command.

Breakpoint table full

Description 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.

Action Enter a BL command to see where software breakpoints are set in your program. Use the BR command to delete all software breakpoints, or use the BD command to delete individual software breakpoints.

C

Cannot allocate host memory

Description This is a fatal error—it means that the debugger is running out of memory.

Action You might try invoking the debugger with the –v 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 allocate system memory

Description This is a fatal error—it means that the debugger is running out of memory.

Action You might try invoking the debugger with the –v option so that fewer symbols may be loaded. Or you might want to relink your program and link in fewer modules at a time.

Corrupt call stack

Description The debugger tried to update the CALLS window and couldn't. This may be because a function was called that didn't return. Or it could be that the program stack was overwritten in target memory. Another reason you may have this message is that you are debugging code that has optimization enabled (for example, you did not use the –g compile switch); if this is the case, ignore this message—code execution is not affected.

Action If your program called a function that didn't return, then this is normal behavior (as long as you intended for the function not to return). Otherwise, you may be overwriting program memory.

Cannot change directory

Description The directory name specified with the CD command either doesn't exist or is not in the current or auxiliary directories.

Action 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 C:\c3xhll, not just c3xhll).

Cannot edit field

Description Expressions that are displayed in the WATCH window cannot be edited.

Action If you attempted to edit an expression in the WATCH window, you may have actually wanted to change the value of a symbol or register used in the expression. Use the ? or EVAL command to edit the actual symbol or register. The expression value will automatically be updated.

Cannot find/open initialization file

Description The debugger can't find the init.cmd file.

Action 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 *Setting Up the Debugger Environment* in the appropriate installation guide.

Cannot halt the processor

Description This is a fatal error—for some reason, pressing ESC didn't halt program execution.

Action Exit the debugger. Invoke emurst (emulator only), then invoke the debugger again.

Cannot map into reserved memory: ?

Description The debugger tried to access unconfigured/reserved/nonexistent memory.

Action Remap the reserved memory accesses.

Cannot map port address

Description You attempted to do a connect/disconnect on an illegal port address.

Action Verify that the address you specified is a valid primary bus, expansion bus, or serial port address.

Cannot open config file

Description The SCONFIG command can't find the screen-customization file that you specified.

Action Be sure that the filename was typed correctly. If it wasn't, re-enter the command with the correct name. If it was, re-enter the command and specify full path information with the filename.

Cannot open "*filename*"

Description The debugger attempted to show *filename* in the FILE window but could not find the file.

Action Be sure that the file exists as named. If it does, enter the USE command to identify the file's directory.

Cannot open object file: "*filename*"

Description The file specified with the LOAD, SLOAD, or RELOAD command is not an object file that the debugger can load.

Action Be sure that you're loading an actual object file. Be sure that the file was linked (you may want to run cl30 again to create an executable object file).

Cannot open new window

Description A maximum of 127 windows can be open at once. The last request to open a window would have made 128, which isn't possible.

Action Close any unnecessary windows. Windows that can be closed include WATCH, CALLS, DISP, and additional MEMORY windows. To close the WATCH window, enter WD. To close the CALLS, DISP, or a MEMORY window, make the desired window active and press F4 .

Cannot read processor status

Description This is a fatal error—for some reason, pressing ESC didn't halt program execution.

Action Exit the debugger. Invoke emurst (emulator only), then invoke the debugger again.

Cannot reset the processor

Description This is a fatal error—for some reason, pressing ESC didn't halt program execution.

Action Exit the debugger. Invoke emurst (emulator only), then invoke the debugger again.

Cannot restart processor

Description If a program doesn't have an entry point, then RESTART won't reset the PC to the program entry point.

Action Don't use RESTART if your program doesn't have an explicit entry point.

Cannot set/verify breakpoint at *address*

Description Either you attempted to set a breakpoint in read-only or protected memory, or there are hardware problems with the target system or EVM. This may also happen when you enable or disable on-chip memory while using breakpoints.

Action Check your memory map. If the address that you wanted to breakpoint wasn't in ROM, see Section E.4 (page E-20).

Cannot step

Description There is a problem with the target system.

Action See Section E.4 (page E-20).

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 E.3 (page E-20).

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.

Command timed out, emulator busy

Description There is a problem with the target system.

Action See Section E.4 (page E-20).

Conflicting map range

Description A block of memory specified with the MA command overlaps an existing memory map entry. Blocks cannot overlap.

Action Use the ML command to list the existing memory map; this will help you find that existing block that the new block would overlap. If the existing block is not necessary, delete it with the MD command and re-enter the MA command. If the existing block is necessary, re-enter the MA command with parameters that will not overlap the existing block.

E

Emulator I/O address is invalid

Description The debugger was invoked with the –p option, and an invalid *port address* was used.

Action For valid *port address* values, refer to the *TMS320C3x Emulator Installation Guide*.

Error in expression

Description This is an expression error.

Action See Section E.3 (page E-20).

Execution error

Description There is a problem with the target system.

Action See Section E.4 (page E-20).

F

File already tied to port

Description You attempted to connect to an address that already has a file connected to it.

Action Connect the file to a mapped port that is not connected to a file.

File already tied to this pin

Description You attempted to connect an input file to an interrupt pin that already has a file connected to it.

Action Use the PINC command to connect the file to another interrupt pin that is not connected to a file.

File does not exist

Description The port file could not be opened for reading.

Action Be sure that the file exists as named. If it does, enter the USE command to identify the file's directory.

Files must be disconnected from ports

Description You attempted to delete a memory map that has files connected to it.

Action You must disconnect a port with the MI command before you can delete it from the memory map.

File not found

Description The filename specified for the FILE command was not found in the current directory or any of the directories identified with D_SRC.

Action Be sure that the filename was typed correctly. If it was, re-enter the FILE command and specify full path information with the filename.

File not found : "*filename*"

Description The filename specified for the LOAD, RELOAD, SLOAD, or TAKE command was not found in the current directory or any of the directories identified with D_SRC.

Action Be sure that the filename was typed correctly. If it was, re-enter the command and specify full path information with the filename.

File too large (*filename*)

Description You attempted to load a file that was more than 65,518 bytes long.

Action Try loading the file without the symbol table (SLOAD), or use cl30 to relink the program with fewer modules.

Float not allowed

Description This is an expression error—a floating-point value was used incorrectly.

Action See Section E.3 (page E-20).

Function required

Description The parameter for the FUNC command must be the name of a function in the program that is loaded.

Action Re-enter the FUNC command with a valid function name.

I

Illegal addressing mode

Description An illegal 'C3x addressing mode was encountered.

Action Refer to the *TMS320C3x User's Guide* for valid addressing modes.

Illegal cast

Description This is an expression error—the expression parameter uses a cast that doesn't meet the C language rules for casts.

Action See Section E.3 (page E-20).

Illegal control transfer instruction

Description The instruction following a delayed branch/call instruction was modifying the program counter.

Action Modify your source code.

Illegal left hand side of assignment

Description This is an expression error—the lefthand side of an assignment expression doesn't meet C language assignment rules.

Action See Section E.3 (page E-20).

Illegal memory access

Description Your program tried to access unmapped memory.

Action Modify your source code.

Illegal opcode

Description An invalid 'C3x instruction was encountered.

Action Modify your source code.

Illegal operand of &

Description This is an expression error—the expression attempts to take the address of an item that doesn't have an address.

Action See Section E.3 (page E-20).

Illegal pointer math

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

Action See Section E.3 (page E-20).

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 E.3 (page E-20).

Illegal structure reference

Description This is an expression error—either the item being referenced as a structure is not a structure, or you are attempting to reference a nonexistent portion of a structure.

Action See Section E.3 (page E-20).

Illegal use of structures

Description This is an expression error—the expression parameter is not using structures according to the C language rules.

Action See Section E.3 (page E-20).

Illegal use of void expression

Description This is an expression error—the expression parameter does not meet the C language rules.

Action See Section E.3 (page E-20).

Integer not allowed

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

Action See Section E.3 (page E-20).

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 9–2 (page 9-3).

Invalid color name

Description The COLOR and SCOLOR commands accept a specific set of color attributes as parameters. The parameter entered did not match one of the valid attributes.

Action Re-enter the COLOR or SCOLOR command with a valid color parameter. Valid color attributes are listed in Table 9–1 (page 9-2).

Invalid memory attribute

Description The third parameter of the MA command specifies the type, or attribute, of the block of memory that MA adds to the memory map. The parameter entered did not match one of the valid attributes.

Action Re-enter the MA command. Use one of the following valid parameters to identify the memory type:

R, ROM, READONLY	(read-only memory)
W, WOM, WRITEONLY	(write-only memory)
R|W, RAM	(read/write memory)
PROTECT	(no-access memory)
OPORT	(I/O memory)
IPORT	(I/O memory)
IOPORT	(I/O memory)

Invalid object file

Description Either the file specified with the LOAD, SLOAD, or RELOAD command is not an object file that the debugger can load, or it has been corrupted.

Action Be sure that you're loading an actual object file. Be sure that the file was linked (you may want to run cl30 again to create an executable object file). If the file you attempted to load was a valid executable object file, then it was probably corrupted; recompile, assemble, and link with cl30.

Invalid watch delete

Description The debugger can't delete the parameter supplied with the WD command. Usually, this is because the watch index doesn't exist or because a symbol name was typed in instead of a watch index.

Action Re-enter the WD command. Be sure to specify the watch index that matches the item you'd like to delete (this is the number in the left column of the WATCH window). Remember, you can't delete items symbolically—you must delete them by number.

Invalid window position

Description The debugger can't move the active window to the XY position specified with the MOVE command. Either the XY parameters are not within the screen limits, or the active window may be too large to move to the desired position.

Action

- You can use the mouse to move the window.
- If you don't have a mouse, enter the MOVE command without parameters; then use the arrow keys to move the window. When you're finished, you *must* press ESC or ↵.
- If you prefer to use the MOVE command with parameters, 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, the minimum size is 4 by 3; the maximum size depends on which screen size you're using.

L

Load aborted

Description This message always follows another message.

Action Refer to the message that preceded *Load aborted*.

Lost power (or cable disconnected)

Description Either the target cable is disconnected, or the target system is faulty.

Action Check the target cable connections. If the target seems to be connected correctly, see Section E.4 (page E-20).

Lost processor clock

Description Either the target cable is disconnected, or the target system is faulty.

Action Check the target cable connections. If the target seems to be connected correctly, see Section E.4 (page E-20).

Lval required

Description This is an expression error—an assignment expression was entered that requires a legal left-hand side.

Action See Section E.3 (page E-20).

M

Memory access error at *address*

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

Action See Section E.4 (page E-20).

Memory map table full

Description Too many blocks have been added to the memory map. This rarely happens unless blocks are added word by word (which is inadvisable).

Action 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.

Nesting of repeats cannot exceed 100

Description The debugger cannot simulate more than 100 levels of repeat nesting in an input data file. If this happens, the debugger disconnects the input file from the pin.

Action Correct the input file so that the data does not include nesting repetition exceeding 100 levels. Use the PINC command to reconnect the input file to the desired pin.

No file connected to this pin

Description You tried to disconnect the input file from a pin that was not previously connected to that pin.

Action Use the PINL command to list all of the pins and the files connected to them. Use the PIND command to re-enter the correct pinname and filename.

Nonrepeatable instruction

Description The instruction following the RPT instruction is not a repeatable instruction.

Action Modify your code.

P

Pinname not valid for this chip

Description You attempted to connect or disconnect an input file to an invalid interrupt pin.

Action Either reconnect the input file to an unused interrupt pin ($\overline{\text{INT}}$0, $\overline{\text{INT}}$1, $\overline{\text{INT}}$2, or $\overline{\text{INT}}$3), or disconnect the input file from the interrupt pin.

Pointer not allowed

Description This is an expression error.

Action See Section E.3 (page E-20).

Processor is already running

Description One of the RUN commands was entered while the debugger was running free from the target system.

Action Enter the HALT command to stop the free run, then re-enter the desired RUN command.

R

Read not allowed for port

Description You attempted to connect a file for input operation to an address that is not configured for read.

Action Remap the port of correct the access in your source code.

Register access error

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

Action See Section E.4 (page E-20).

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, you can specify only the first address of a defined block.

Structure member not found

Description This is an expression error—an expression references a nonexistent structure member.

Action See Section E.3 (page E-20).

Structure member name required

Description This is an expression error—a symbol name followed by a period but no member name.

Action See Section E.3 (page E-20).

Structure not allowed

Description This is an expression error—the expression is attempting an operation that cannot be performed on a structure.

Action See Section E.3 (page E-20).

Syntax error at line number

Description The debugger will not simulate interrupts from the input data file and disconnects the input file.

Action Correct the syntax in the input data file. Reconnect the input file to the pin using the PINC command.

T

Take file stack too deep

Description Batch files can be nested up to 10 levels deep. Batch files can call other batch files, which can call other batch files, and so on. Apparently, the batch file that you are TAKEing calls batch files that are nested more than 10 levels deep.

Action Edit the batch file that caused the error. Instead of calling another batch file from within the offending file, you may want to copy the contents of the second file into the first. This will remove a level of nesting.

Too few instruction words in RPTB

Description The length of the repeat block was less than three instruction words.

Action Modify your code.

Too many breakpoints

Description 200 breakpoints are already set, and there was an attempt to set another. Note that the maximum limit of 200 breakpoints includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.

Action Enter a BL command to see where you have breakpoints set in your program. Use the BR command to delete all breakpoints, or use the BD command to delete individual software breakpoints.

Too many paths

Description More than 20 paths have been specified cumulatively with the USE command, D_SRC environment variable, and –i debugger option.

Action Don’t enter the USE command before entering another command that has a *filename* parameter. Instead, enter the second command and specify full path information for the *filename*.

U

Undeclared port address

Description You attempted to do a connect/disconnect on an address that isn't declared as a port.

Action Verify the address of the port to be connected or disconnected.

User halt

Description The debugger halted program execution because you pressed the ESC key.

Action 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:

CALLS	**CP**U	**DISP**
COMMAND	**DISA**SSEMBLY	**F**ILE
MEMORY	**P**ROFILE	**W**ATCH

Write not allowed for port

Description You attempted to connect a file for output operation to an address that is not configured for write.

Action Either change the 'C3x software to write to a port that is configured for write, or change the attributes of the port.

E.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.

E.4 Additional Instructions for Hardware Errors

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

- ❑ If a bus fault occurs, the emulator may not be able to access memory.
- ❑ The ’C3x must be reset before you can use the emulator. Most target systems reset the ’C3x at power-up; your target system may not be doing this.

Appendix F

Glossary

A

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

aggregate type: A C data type such as a structure or array in which a variable is composed of multiple variables, called members.

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

ANSI C: A version of the C programming language that conforms to the C standards defined by the *American National Standards Institute.*

assembly mode: A debugging mode that shows assembly language code in the DISASSEMBLY and doesn't show the FILE window, no matter what type of code is currently running.

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

auto mode: A context-sensitive debugging mode that automatically switches between showing assembly language code in the DISASSEMBLY window and C code in the FILE window, depending on what type of code is currently running.

B

batch file: One of two different types of files. One type contains DOS commands for the PC to execute. A second type of batch file contains debugger commands for the debugger to execute. The PC 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.

CALLS window: A window that lists the functions called by your program.

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

children: Additional windows opened for aggregate types that are members of a parent aggregate type displayed in an existing DISP window.

cl30: A shell utility that invokes the TMS320 floating-point DSP compiler, assembler, and linker to create an executable object file version of your program.

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

CLK: A pseudoregister that shows the number of CPU cycles consumed during benchmarking. The value in CLK is valid only after you enter a RUNB command but before you enter another RUN command.

code-display windows: Windows that show code, text files, or code-specific information. This category includes the DISASSEMBLY, FILES, and CALLS windows.

COFF: *Common Object File Format*. An implementation of the object file format of the same name developed by AT&T. The TMS320 floating-point DSP compiler, assembler, and linker use and generate COFF files.

command line: The portion of the COMMAND window where you can enter commands.

command-line cursor: A block-shaped cursor that identifies the current character position on the command line.

COMMAND window: A window that provides an area for you to enter commands and for the debugger to echo command entry, show command output, and list progress or error messages.

CPU window: A window that displays the contents of ’C3x on-chip registers, including the program counter, status register, A-file registers, and B-file registers.

current-field cursor: A screen icon that identifies the current field in the active window.

cursor: An icon on the screen (such as a rectangle or a horizontal line) that is used as a pointing device. The cursor is usually under mouse or keyboard control.

D

data-display windows: Windows for observing and modifying various types of data. This category includes the MEMORY, CPU, DISP, and WATCH windows.

D_DIR: An environment variable that identifies the directory containing the commands and files necessary for running the debugger.

debugger: A window-oriented software interface that helps you to debug ’C3x programs running on a ’C3x emulator, EVM, or simulator.

disassembly: Assembly language code formed from the reverse-assembly of the contents of memory.

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

DISP window: A window that displays the members of an aggregate data type.

display area: The portion of the COMMAND window where the debugger echoes command entry, shows command output, and lists progress or error messages.

D_OPTIONS: An environment variable that you can use for identifying often-used debugger options.

drag: To move the mouse while pressing one of the mouse buttons.

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.

emulator: A debugging tool that is external to the target system and provides direct control over the ’C3x processor that is on the target system.

emurst: A utility that resets the emulator.

environment variable: A special system symbol that the debugger uses for finding directories or obtaining debugger options.

EVM: *Evaluation Module.* A development tool that lets you execute and debug applications programs by using the ’C3x debugger.

evmrst: A utility that resets the EVM.

F

FILE window: A window that displays the contents of the current C code. The FILE window is intended primarily for displaying C code but can be used to display any text file.

I

init.cmd: A batch file that contains debugger-initialization commands. If this file isn’t present when you first invoke the debugger, then all memory is invalid.

initdb.bat: A batch file created to contain DOS commands to set up the debugger environment.

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

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

M

memory map: A map of memory space that tells the debugger which areas of memory can and can’t be accessed.

MEMORY window: A window that displays the contents of memory.

menu bar: A row of pulldown menu selections found at the top of the debugger display.

mixed mode: A debugging mode that simultaneously shows both assembly language code in the DISASSEMBLY window and C code in the FILE window.

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

P

PC: Personal computer or program counter, depending on the context and where it's used in this book: 1) In installation instructions or information relating to hardware and boards, *PC* means *Personal Computer* (as in IBM PC). 2) In general debugger and program-related information, *PC* means *Program Counter*, which is the register that identifies the current statement in your program.

point: To move the mouse cursor until it overlays the desired object on the screen.

port address: The PC I/O memory space that the debugger uses for communicating with the emulator or EVM. The port address is selected via switches on the emulator or 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

scalar type: A C type in which the variable is a single variable, not composed of other variables.

scrolling: A method of moving the contents of a window up, down, left, or right to view contents that weren't originally shown.

side effects: A feature of C expressions in which using an assignment operator in an expression affects the value of one of the components used in the expression.

simulator: A development tool that simulates the operation of the 'C3x and lets you execute and debug applications programs by using the 'C3x debugger.

single-step: A form of program execution that allows you to see the effects of each statement. The program is executed statement by statement; the debugger pauses after each statement to update the data-display windows.

symbol table: A file that contains the names of all variables and functions in your 'C3x program.

system shell: A utility invoked with the SYSTEM command, which makes it possible for the debugger to blank the debugger display and temporarily exit to the DOS prompt. This allows you to enter DOS commands *or* allows the debugger to display information resulting from a DOS command.

T

target system: A 'C3x board that works with the emulator; the emulator doesn't contain a 'C3x device, so it must use a 'C3x target board. Usually, the target system is a board that you have designed; you use the emulator and debugger to help you debug your design.

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

Note: All page numbers preceded by the word *EMU* refer to the *TMS320C3x Emulator Installation Guide*; page numbers preceded by *SIM* refer to the *TMS320C3x Simulator Getting Started Guide,* and page numbers preceded by *EVM* refer to the *TMS320C3x EVM Installation Guide*. All other references refer to this user's guide.

A

C

E

F

G

H

I

K

O

P

X

Z