TEXAS INSTRUMENTS

TMS320C5xx C Source Debugger

User's Guide

1994 Microprocessor Development Systems

Texas Instruments

Printed in U.S.A., June 1994, reprinted October 1997
SDS

SPRU099A

User's Guide

TMS320C5xx C Source Debugger

1994

TMS320C5xx
C Source Debugger
User's Guide

SPRU099
June 1994

Printed on Recycled Paper

TMS320C5x 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 *An Introductory Tutorial to the C Source Debugger* (Chapter 3)?

☐ Yes ☐ No

Which additional subjects should be included in future versions of *An Introductory Tutorial to the C Source Debugger*?

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 _____

June 1994

TEXAS INSTRUMENTS

TMS320C5xx 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 profiling session
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 stopping 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 system file
vaa *filename*	Save all data for the current view

Phone Numbers

DSP Hotline: (713) 274–2320

Entering the Profiling Environment

The profiling environment is supported under all platforms except DOS.

Emulator:	**emu5xx –profile**
EVM:	**evm5xx –profile**
Simulator:	**sim5xx –profile**

Debugger Commands That Can Be Used in the Profiling Environment

?	MA	PROMPT	SR
ALIAS	MAP	QUIT	SYSTEM
CD	MC	RELOAD	TAKE
CLS	MD	RESET	UNALIAS
DASM	MI	RESTART	USE
DIR	ML	SA	VAA
DLOG	MOVE	SCONFIG	VAC
ECHO	MR	SD	VERSION
EVAL	MS	SIZE	VR
FILE	PF	SL	WIN
FUNC	PQ	SLOAD	ZOOM
LOAD	PR		

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

!	CSTEP	PAUSE	SET
@	DISP	PESC	SETF
ADDR	FILL	PHALT	SOUND
ASM	GO	PRUN	SPAWN
BA	HALT	PRUNF	SSAVE
BD	HELP	PSTEP	STAT
BL	HISTORY	RETURN	STEP
BORDER	MC	RUN	UNSET
BR	MEM	RUNB	WA
C	MIX	RUNF	WD
CALLS	NEXT	SCOLOR	WHATIS
CNEXT	PATCH	SEND	WR
COLOR			

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, addresses†	**MCRE**	*filename*, *line number*‡	**MARE**	*address*‡
Functions				
❑ By function name	**MCFE**	*function*	not applicable	
❑ All functions in a module	**MCFM**	*filename*		
❑ All functions everywhere	**MCFG**			

† C areas are identified by line number; disassembly areas are identified by address.
‡ Multiple line numbers or addresses, separated by commas, are allowed.

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 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**	

† C areas are identified by line number; disassembly areas are identified by address.
‡ Multiple line numbers or addresses, separated by commas, are allowed.

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 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 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**	

† C areas are identified by line number; disassembly areas are identified by address.
‡ Multiple line numbers or addresses, separated by commas, are allowed.

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 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 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**	

† C areas are identified by line number; disassembly areas are identified by address.
‡ Multiple line numbers or addresses, separated by commas, are allowed.

Changing the PROFILE Window Display

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 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 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**	

† C areas are identified by line number; disassembly areas are identified by address.
‡ Multiple line numbers or addresses, separated by commas, are allowed.

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**

TEXAS INSTRUMENTS

TMS320C5xx C Source Debugger Reference Card

Phone Number

DSP Hotline: (713) 274–2320

Invoking the Debugger

Emulator:	**emu5xx**	[*filename*]	[*–options*]
EVM:	**evm5xx**	[*filename*]	[*–options*]
Simulator:	**sim5xx**	[*filename*]	[*–options]*

Debugger Options

Option	Description
–b[b]	**Select the screen size.** Option / Chars. / Comment none / 80 × 25 / (default) –b / 80 × 43 / (EGA or VGA) –bb / 80 × 50 / (VGA only)
–d *machine-name*	**Display the debugger on a different machine.** (emulator and simulator) If using X Windows, displays the debugger on a different machine than the one the program is running on.
–f *filename*	**Identify a new board configuration file.** Specifies a board configuration file that will be used instead of board.dat.
–i *pathname*	**Identify additional directories.** Identifies directories that contain source files.
–n *processor name*	**Identify processor for debugging.** (emulator under OS/2) Specifies which particular 'C5xx to debug.
–p *port address*	**Identify the port address.** (emulator, SWDS, 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 the symbol table only.** Tells the debugger to load *filename*'s symbol table only.
–t *filename*	**Identify a new initialization file.** Allows you to specify an initialization file.
–v	**Load without the 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	***Tool***
? *expression* [, *display format*]	***ALL***
!{*prompt number*	*string*}	***PDM***
@ variable name = expression	***PDM***
addr *address*[**@prog** | **@data** | **@io**] **addr** *function name*	***DBG***
alias [*alias name* [, "*command string*"]]	***ALL***
asm	***DBG***
ba *address*	***DBG***
bd *address*	***DBG***
bl	***DBG***
border [*active*] [, [*inactive*] [,*resize*]]	***DBG***
br	***DBG***
c	***DBG***
calls	***DBG***
cd [*directory name*] **chdir** [*directory name*]	***D/P***
cls	***D/P***
cnext [*expression*]	***DBG***
color *area, attr*$_1$ [,*attr*$_2$ [,*attr*$_3$ [,*attr*$_4$]]]	***DBG***
cstep [*expression*]	***DBG***
dasm *address*[**@prog** | **@data**] **dasm** *function name*	***D/P***
dir [*directory*]	***D/P***
disp *expression* [, *format*] [**@prog** | **@data**]	***DBG***
dlog *filename*[**,**{**a** | **w**}]	***ALL***
echo *string*	***ALL***
eval *expression*[**@prog** | **@data**] **e** *expression*[**@prog** | **@data**]	***D/P***
eval *expression*[**–g** {*group* | *processor name*}] *variable name=expression*[*, format*]	***D/P***
file *filename*	***D/P***
fill *address* , *page, length, data*	***DBG***
func *function name* **func** *address*	***D/P***
go [*address*]	***DBG***
halt	***DBG***‡
if *expression* *PDM command list* [**elif** *expression* *PDM command list*] [**else** *PDM command list*] **endif**	***PDM***

ALL = debugger, profiler and PDM ***DBG*** = basic debugger only
D/P = profiler and basic debugger ***PDM*** = PDM only
PRO = profiler only † simulator only ‡ emulator only

Summary of Debugger Commands	*Tool*
if *Boolean expression* *debugger command list* [**else** *debugger command list*] **endif**	***D/P***
load *object filename*	***D/P***
loop *Boolean expression* *PDM command list* [**break**] [**continue**] **endloop**	***PDM***
loop *expression* *debugger command list* **endloop**	***D/P***
ma *address, page, length, type*	***D/P***
map {**on** | **off**}	***D/P***
mc *port address, page, length*, *filename, fileaccess*	***DBG***†
md *address, page*	***D/P***
mem[*#*] *expression* [**@prog** | **@data**] [, *format*]	***DBG***
mi *port address, page, fileaccess*	***D/P***†
mix	***DBG***
ml	***D/P***
move [*X*, *Y* [, *width*, *length*]]	***D/P***
mr	***D/P***
ms *address, page, length, filename*	***D/P***
next [*expression*]	***DBG***
patch *address*, *assembly language instruction*	***DBG***
pause	***PDM***
pesc [**–g** {*group* | *processor name*}	***PDM***
pf *starting point* [, *update rate*]	***PRO***
phalt [{**–g** *group* | *processor name*}	***PDM***
pq *starting point* [, *update rate*]	***PRO***
pr [*clear data* [, *update rate*]]	***PRO***
prompt *new prompt*	***D/P***
prun [**–r**] [**–g** {*group* | *processor name*}]	***PDM***
prunf [**–g** {*group* | *processor name*}]	***PDM***
pstep [**–g** {*group* | *processor name*}] [*count*]	***PDM***
quit	***ALL***
reload *object filename*	***D/P***
reset	***D/P***
restart **rest**	***D/P***
return **ret**	***DBG***

ALL = debugger, profiler and PDM ***DBG*** = basic debugger only
D/P = profiler and basic debugger ***PDM*** = PDM only
PRO = profiler only † simulator only ‡ emulator only

Summary of Debugger Commands	*Tool*
run [*expression*]	***DBG***
runb	***DBG***
runf	***DBG***
sa *address*	***PRO***
scolor *area*, $attr_1$ [, $attr_2$ [, $attr_3$ [, $attr_4$]]]	***DBG***
sconfig [*filename*]	***D/P***
sd *address*	***PRO***
send [**–r**] [**–g** {*group* | *processor name*}] *debugger command*	***PDM***
set [*group* [= *list of processor names*]] **set** [*variable* [= *string value*]]	***PDM***
setf [*data type, display format*]	***DBG***
size [*width*, *length*]	***D/P***
sl	***PRO***
sload *object filename*	***D/P***
sound on | off	***DBG***
spawn emu5xx –n] *processor name* [*options*]	***PDM***
sr	***PRO***
ssave [*filename*]	***DBG***
stat [{**–g** *group* | *processor name*}]	***PDM***
step [*expression*]	***DBG***
system [*DOS command* [, *flag*]] **system** *operating-system command*	***D/P*** ***PDM***
take *batch filename* [*, suppress echo flag*] **take** *batch filename*	***D/P*** ***PDM***
unalias *alias name* **unalias ***	***ALL***
unset *group name* **unset ***	***PDM***
use *directory name*	***D/P***
vaa *filename*	***PRO***
vac *filename*	***PRO***
version	***D/P***
vr	***PRO***
wa *expression* [**@prog** | **@data**] [, [*label*], *format*]	***DBG***
wd *index number*	***DBG***
whatis *symbol*	***DBG***
win *WINDOW NAME*	***D/P***
wr	***DBG***
zoom	***D/P***

ALL = debugger, profiler and PDM ***DBG*** = basic debugger only
D/P = profiler and basic debugger ***PDM*** = PDM only
PRO = profiler only † simulator only ‡ emulator only

Memory Types

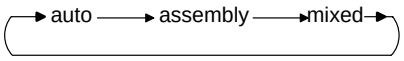
To identify this kind of memory	Use this keyword as the *type* parameter
read-only memory	**R** or **ROM**
write-only memory	**W** or **WOM**
read/write memory	**R|W** or **RAM**
no-access memory	**PROTECT**
input port	**IPORT**
output port	**OPORT**
input/output port	**IOPORT**

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:	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	F8

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 PG DN 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 these function keys
FILE or DISASSEMBLY window: Set or clear a breakpoint	F9
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	

Halting or Escaping From an Action

To do this	Use these function keys
Halt program execution	ESC
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	

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 Analysis menu	ALT A
Display an adjacent menu	← or →
Execute any of the choice from a displayed pulldown menu	Press the highlighted letter corresponding to your choice

Moving or Sizing a Window

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

To do this	Use these function keys
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 the Active Window's Contents

To do this	Use these function keys
Scroll up through the window contents, one window length at a time	PG UP
Scroll down through the window contents, one window length at a time	PG DN
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 PG UP
DISP windows only: Scroll down through an array of structures	CTRL PG DN

IMPORTANT NOTICE

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

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

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

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

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

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

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

Preface

Read This First

What Is This Book About?

This book tells you how to use the TMS320C5xx C source debugger with these debugging tools:

- Emulator
- Evaluation Module (EVM)
- Simulator

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

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

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

Note:

The Evaluation Module described in this book is not currently available.

How to Use This Manual

The goal of this book is to help you learn the Texas Instruments standard 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 'C5xx software tools, tells you how to prepare a 'C5xx program for debugging, and provides instructions and options for invoking the debugger and the parallel debug manager (PDM).
  - Chapter 2 is a tutorial that introduces you to many of the debugger features.
  - Chapter 3 is an advanced tutorial that introduces you to the features of the PDM.
- **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 4 describes all of the debugger's windows and tells you how to move them and size them; Chapter 5 describes everything you need to know about entering commands.
- **Part III: Reference Material** provides supplementary information.
  - Chapter 13 gives 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 14 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.
  - If you plan to debug an application for a multiprocessing system. read Chapter 2.
  - Complete the tutorial in Chapters 3.
  - Read the alphabetical command reference in Chapter 14.
- 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.
  - If you plan to debug an application for a multiprocessing system. read Chapter 2.
  - Complete the tutorial in Chapter 3.
  - Read all of the chapters in Part II.

Notational Conventions

This document uses the following conventions.

- 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. However, parameters for certain commands *are* case sensitive; the book describes these instances.

- 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.

Please read each caution statement carefully.

Related Documentation From Texas Instruments

The following books describe the TMS320C5xx devices 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.

TMS320C5xx Assembly Language Tools User's Guide (literature number SPRU102) 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 'C5xx generation of devices.

TMS320C5xx Optimizing C Compiler User's Guide (literature number SPRU103) describes the 'C5xx C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C5xx generation of devices.

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	Call the TI Customer Response Center: **(800) 336–5236** Or write to: Texas Instruments Incorporated Market Communications Manager, MS 736 P.O. Box 1443 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call the TI Literature Response Center: **(800) 477–8924**
Ask questions about product operation or report suspected problems	Call the DSP hotline: **(713) 274–2320**
Report mistakes in this document or any other TI documentation Please mention the full title of the book and the date of publication (from the spine and/or front cover) in your correspondence.	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

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

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

SPARC is a trademark of SPARC International, Inc.

SunOS and SunView are trademarks of Sun Microsystems, Inc.

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

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

Contents

Part I: Hands-On Information

Figures

Tables

Chapter 1

Overview of a Code Development and Debugging System

The TMS320C5xx C source debugger is an advanced programmer's interface that helps you to develop, test, and refine 'C5xx programs (compiled with the 'C5xx optimizing ANSI C compiler) and assembly language programs. The debugger is the interface to the TMS320C5xx EVM, simulator, and unique scan-based, realtime emulator.

This chapter gives an overview of the programmer's interface, describes the 'C5xx code development environment, and provides instructions and options for invoking the debugger.

Topic **Page**

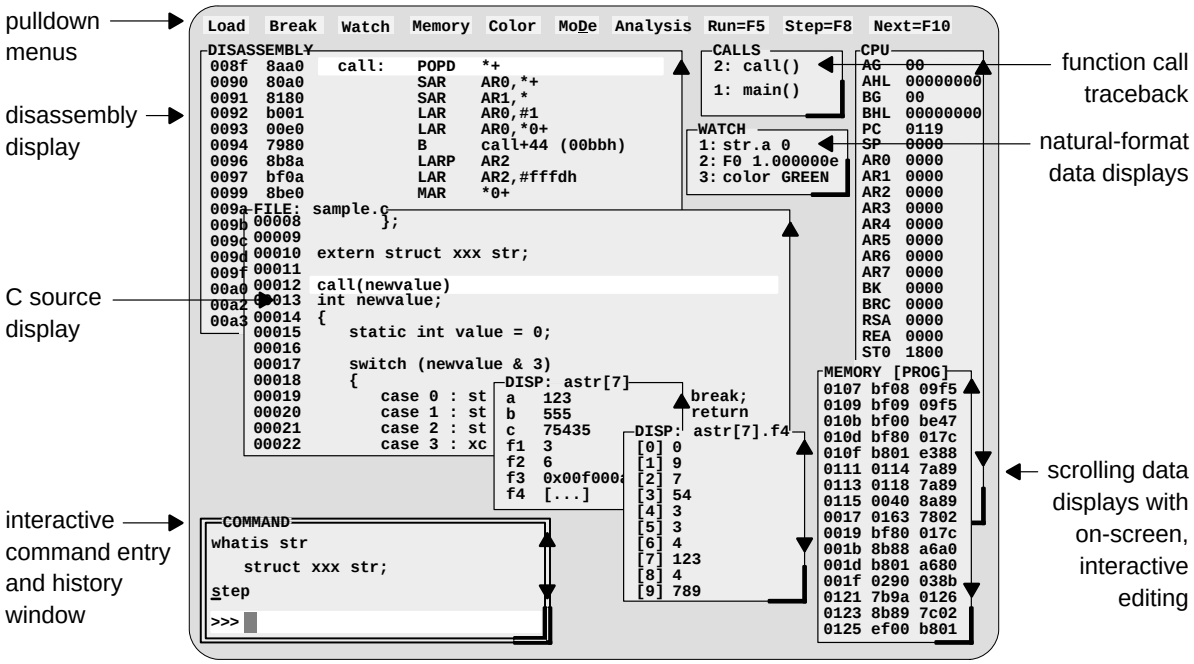
1.1 Description of the TMS320C5xx C Source Debugger

The ’C5xx 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. Additionally, unlike many other debuggers, the ’C5xx debugger’s higher-level features are available even when you’re debugging assembly language code.

The Texas Instruments standard 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

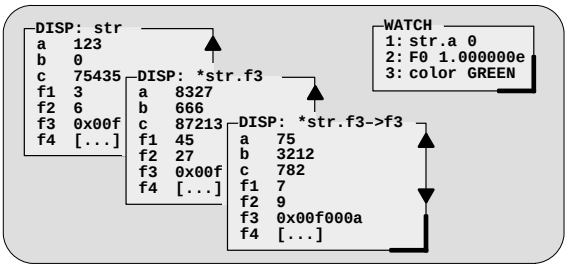


Key features of the debugger

- **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 'C5xx 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 Analysis Interface

In addition to the basic debugger features, the ’C5xx has an analysis module on the chip that allows the simulator and emulator to monitor the operations of your target system. This expands your debugging capabilities beyond simple software breakpoints.

The simulator and emulator interfaces to the analysis module provide you with easy-to-use windows, dialog boxes, and commands that give you a detailed look into the operations of your target system. The analysis interface captures ’C5xx bus cycle information in real time and reacts to this information through actions such as hardware breakpoints. Such features give you the ability to stop the processor and track the path your program took before reaching the breakpoint or event.

Key features of the simulator analysis interface

- **Hardware breakpoints.** You can set up the simulator analysis interface to halt the processor during execution of your program. The events that cause the processor to stop are called *break events*. A break event can define a variety of conditions, including:
  - Bus accesses
  - Instruction fetches
  - Data breakpoints between two instruction addresses.

Key features of the emulator analysis interface

- **Event Counting.** The emulator analysis interface can count nine types of *events*. You have the option of counting the number of times a defined event occurred during execution of your program or stopping processing after a certain number of events are detected. You can count only one event at a time, including:
  - Bus accesses
  - CPU clock cycles
  - Calls/branches taken
  - Interrupts or traps taken
  - Returns from interrupts, traps, or calls
  - Instruction fetches
- **Hardware breakpoints.** You can also set up the emulator analysis interface to halt the processor during execution of your program. The events that cause the processor to stop are called *break events*. A break event can define a variety of conditions, including:
  - Bus accesses
  - CPU clock cycles
  - Calls/branches taken
  - Interrupts or traps taken
  - Returns from interrupts, traps, or calls
  - Instruction fetches
  - Low levels on EMU0/1 pins (EMU0 and EMU1)

- **EMU0/1 pins.** In a system of multiple ’C5xx processors connected by EMU0/1 (emulation event) pins, setting up the EMU0/1 pins allows you to create global breakpoints. Whenever one processor in your system reaches a breakpoint (software or hardware), *all* processors in the system can be halted.

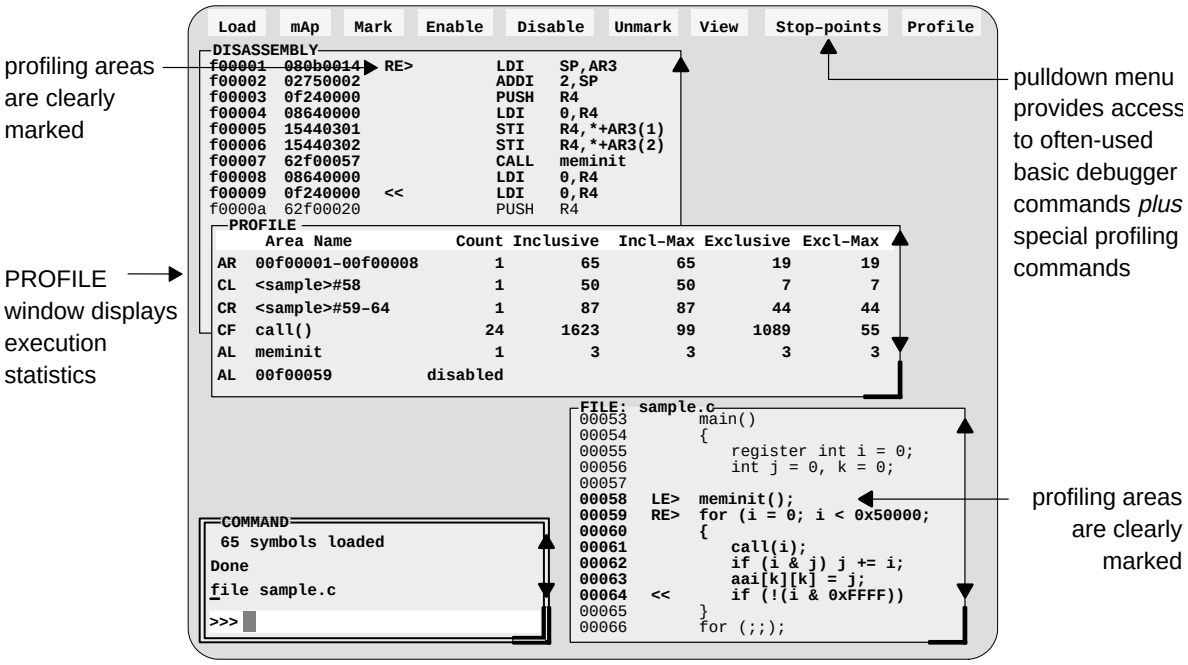
- **The PC discontinuity stack.** *Discontinuity* occurs when the addresses fetched by the debugger become nonsequential as a result of loading the PC (through branches, calls, return instructions, for example) with new values. You can view these values through the PC discontinuity stack and easily track the progress of your program to see exactly how the debugger reached its current state.

1.3 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 has these additional features:

- **More efficient code.** Within the profiling environment, you can quickly identify busy sections in your programs. This helps you to direct valuable development time toward 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.

1.4 Description of the Parallel Debug Manager (PDM)

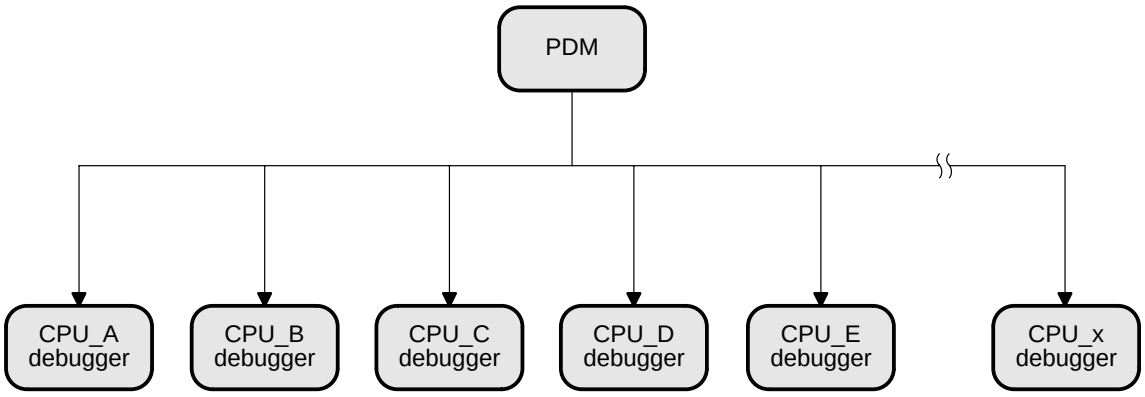
The TMS320C5xx emulation system is a true multiprocessing debugging system. It allows you to debug your entire application by using the parallel debug manager (PDM). The PDM is a command shell that controls and coordinates multiple debuggers, providing you with the ability to:

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

You can operate the PDM only on PCs running OS/2 or Sun workstations running OpenWindows. The PDM is invoked and PDM commands are executed from a command shell window under the host windowing system. From the PDM, you can invoke and control debuggers for each of the processors in your multiprocessing system.

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

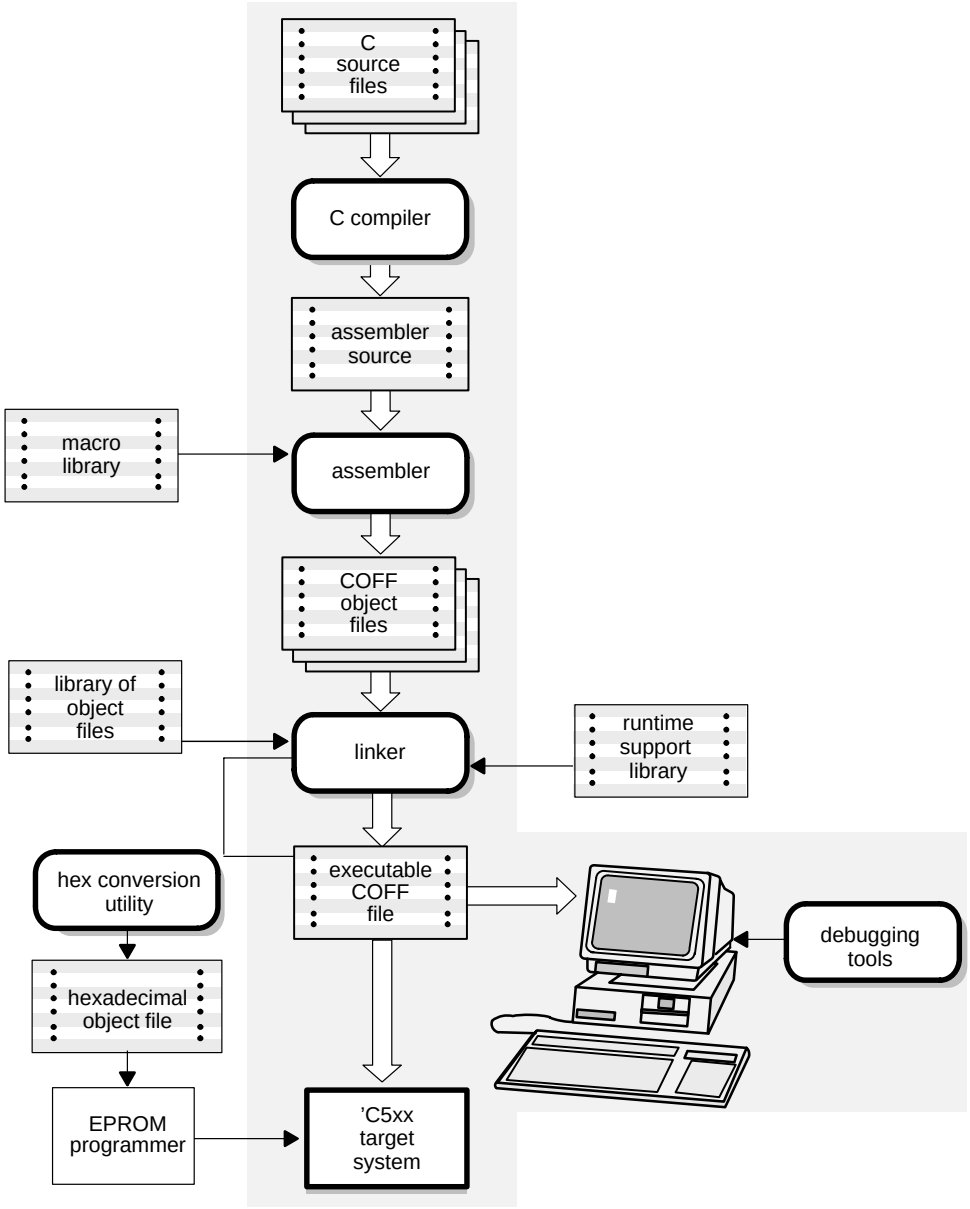
Figure 1–3. The PDM Environment



1.5 Developing Code for the TMS320C5xx

The ’C5xx is well supported by a complete set of hardware and software development tools, including a C compiler, assembler, and linker. Figure 1–4 illustrates the ’C5xx code development flow. The most common paths of software development are highlighted in grey; the other portions are optional.

Figure 1–4. TMS320C5xx 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 1–4.

C compiler

The 'C5xx **optimizing ANSI C compiler** is a full-featured optimizing compiler that translates standard ANSI C programs into 'C5xx 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 operations. Functions for I/O and signal handling are not included, because they are application specific.
- **Flexible assembly language interface.** The compiler has straightforward calling conventions, allowing you to easily write assembly and C functions that call each other.
- **Shell program.** The compiler package includes a shell program that enables you to compile, assemble, and link programs in a single step.
- **Source interlist utility.** The compiler package includes a utility that interlists your original C source statements into the assembly language output of the compiler. This utility provides you with an easy method for inspecting the assembly code generated for each C statement.

assembler

The **assembler** translates ’C5xx 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 **TMS320C5xx target system.** You can use one of several **debugging tools** to refine and correct your code:

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

Each of these tools uses the ’C5xx debugger as a software interface.

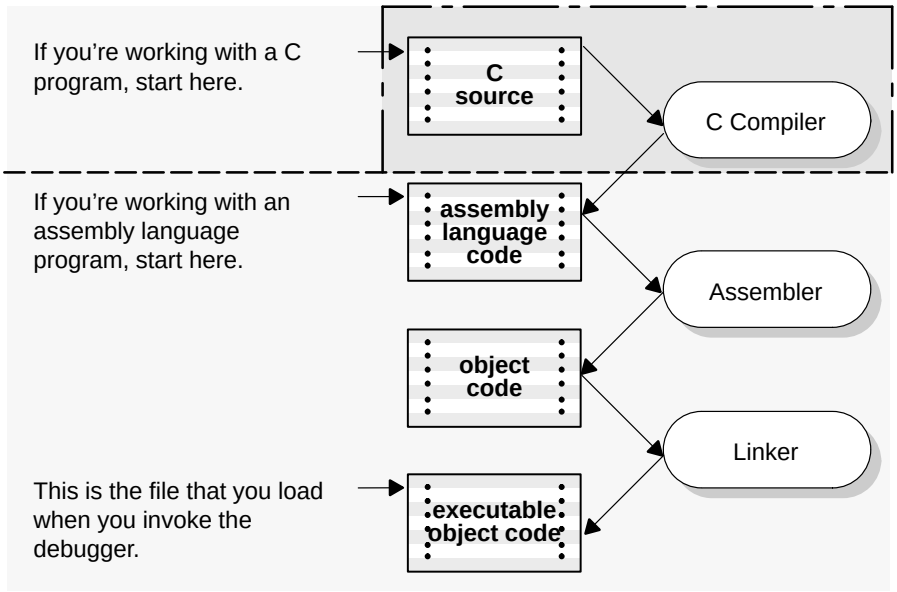
hex conversion utility

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

1.6 Preparing Your Program for Debugging

Figure 1–5 illustrates the steps you must go through to prepare a program for debugging.

Figure 1–5. 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 CL500 shell program. The *TMS320C5xx Assembly Language Tools User's Guide* and *TMS320C5xx 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:

cl500 [*–options*] **–g** [*filenames*] [**–z** [*link options*]]

cl500 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.

–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.

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

–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* (any extension that begins with o)	object file	linked
none (.c assumed)	C source	compiled, assembled, and linked

1.7 Invoking the Debuggers and the PDM

If you are using an XDS510 emulator, there are two ways to invoke the debugger:

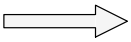
- You can invoke a standalone debugger that is *not* controlled by the parallel debug manager (PDM), or
- You can invoke several debuggers that are under the control of the PDM.

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

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

Invoking a standalone debugger

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

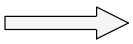


```
emulator:  emu5xx[filename]  [–options]
EVM:       evm5xx[filename]  [–options]
simulator: sim5xx [filename]  [–options]
```

- **emu5xx, evm5xx,** and **sim5xx** are the commands that invoke the debugger. Enter one of these commands from the operating-system command line.
- *filename* is an optional parameter that names an object file that the debugger loads into memory during invocation. The debugger looks for the file in the current directory; if the file 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. See Section 1.8, page 1-18, for a complete list of debugger options.

Invoking multiple debuggers (emulator only)

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



pdm [**–t** *filename*]

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

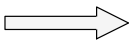
When you invoke the PDM, it looks for a file called init.pdm. This file contains initialization commands for the PDM. The PDM searches for the init.pdm file in the current directory and in the directories you specify with the D_DIR environment variable. If the PDM can't find the initialization file,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

Note:

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

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

To invoke a debugger, use the SPAWN command. Here's the basic format for this command:



spawn emu5xx –n *processor name* [*filename*] [*options*]

- **spawn emu5xx** is the executable command that invokes the debugger. To invoke a debugger, the PDM must be able to find the executable file for that debugger. The PDM will first search the current directory and then search the directories listed with the PATH statement or path environment variable.

- **–n** *processor name* supplies a processor name. You *must* use the –n option because the PDM uses processor names to identify the various debuggers that are running. The processor name can consist of up to eight alphanumeric characters or underscore characters and must begin with an alphanumeric character. Note that the name is not case sensitive.

  The processor name must match one of the names defined in your board configuration file (see Appendix C). For example, to invoke a debugger for a 'C5xx that you had defined as CPU_A, you would enter:

  `spawn emu5xx –n CPU_A` ↵

- *filename* is an optional parameter that names an object file that the debugger loads into memory during invocation. The debugger looks for the file in the current directory; if the file 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. See Section 1.8, page 1-18, for a complete list of debugger options.

1.8 Debugger Options

Table 1–1 lists the debugger options that you can use when invoking a debugger, and the subsections that follow describe these options. You can also specify filename and option information with the D_OPTIONS environment variable (see *Setting up the environment variables* in your installation guide).

Table 1–1. Summary of Debugger Options

Option	Brief Description	Debugger Tools
–b[b]	Select the screen size	All
–d *machinename*	Display the debugger on a different machine (X Windows only)	Emulator and simulator
–f *filename*	Identify a new board configuration file	Emulator
–i *pathname*	Identify additional directories	All
–n *processor name*	Identify processor for debugging	Emulator under OS/2
–p *port address*	Identify the port address	EVM and emulator
–profile	Enter the profiling environment	Emulator and simulator
–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. If you'd like to use a different screen size, the method for doing so varies, depending on the type of system that you're using:

- **PC systems.** You can use the –b or –bb option to select one of these preset screen sizes:
  - **–b** Screen size is 80 characters by 43 lines for EGA or VGA displays.
  - **–bb** Screen size is 80 characters by 50 lines for a VGA display only.

- **Sun systems.** When you run multiple debuggers, the default screen size is a good choice because you can easily fit up to five default-size debuggers on your screen. However, you can change the default screen size by using one of the –b options, which provide a preset screen size, or by resizing the screen at run time. (Note that when you are running a standalone debugger, you can also change the screen size by using one of these methods.)
  - **Using a preset screen size.** Use the –b or –bb option to select one of these preset screen sizes:
    - **–b** Screen size is 80 characters by 43 lines.
    - **–bb** Screen size is 80 characters by 50 lines.
  - **Resizing the screen at run time.** You can resize the screen at run time by using your mouse to change the size of the operating-system window that contains the debugger. The maximum size of the debugger screen is 132 characters by 60 lines.

Displaying the debugger on a different machine (–d option)

This option is valid only when you are using the emulator or simulator. If you are using the X Window System, you can display the debugger on a different machine than the one the program is running on by using the –d option. For example, if you are running a debugger on a machine called opie and you want the debugger display to appear on a machine called barney, use the following command to invoke the debugger:

```
emu5xx -d barney:0
```

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

Identifying a new board configuration file (–f option)

This option is valid only when you are using the emulator. The –f option allows you to specify a board configuration file that will be used instead of board.dat. The format for this option is:

–f *filename*

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:

sim5xx –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).

Identifying the processor that will be debugged (–n option)

The –n option is valid only when using the emulator under OS/2. The –n option allows you to specify which particular ’C5xx you plan to debug. The processor name must match one of the names defined in your board.cfg file. For example, if you wanted to debug a ’C5xx that you defined as cpu_a, you would specify:

```
–n cpu_a
```

Processor names can be any string less than 32 characters long; however, they cannot contain double quotes, a line feed, or a newline character. For more information about the board.cfg file, refer to Chapter 1 of the *TMS320C5xx Emulator Installation Guide*.

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 EVM or emulator. 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.

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 the D_OPTIONS environment variable (described in the installation guide).

1.9 Exiting a Debugger or the PDM

To exit any version of the debugger, enter the following command from the COMMAND window of the debugger you want to close:

```
quit ↵
```

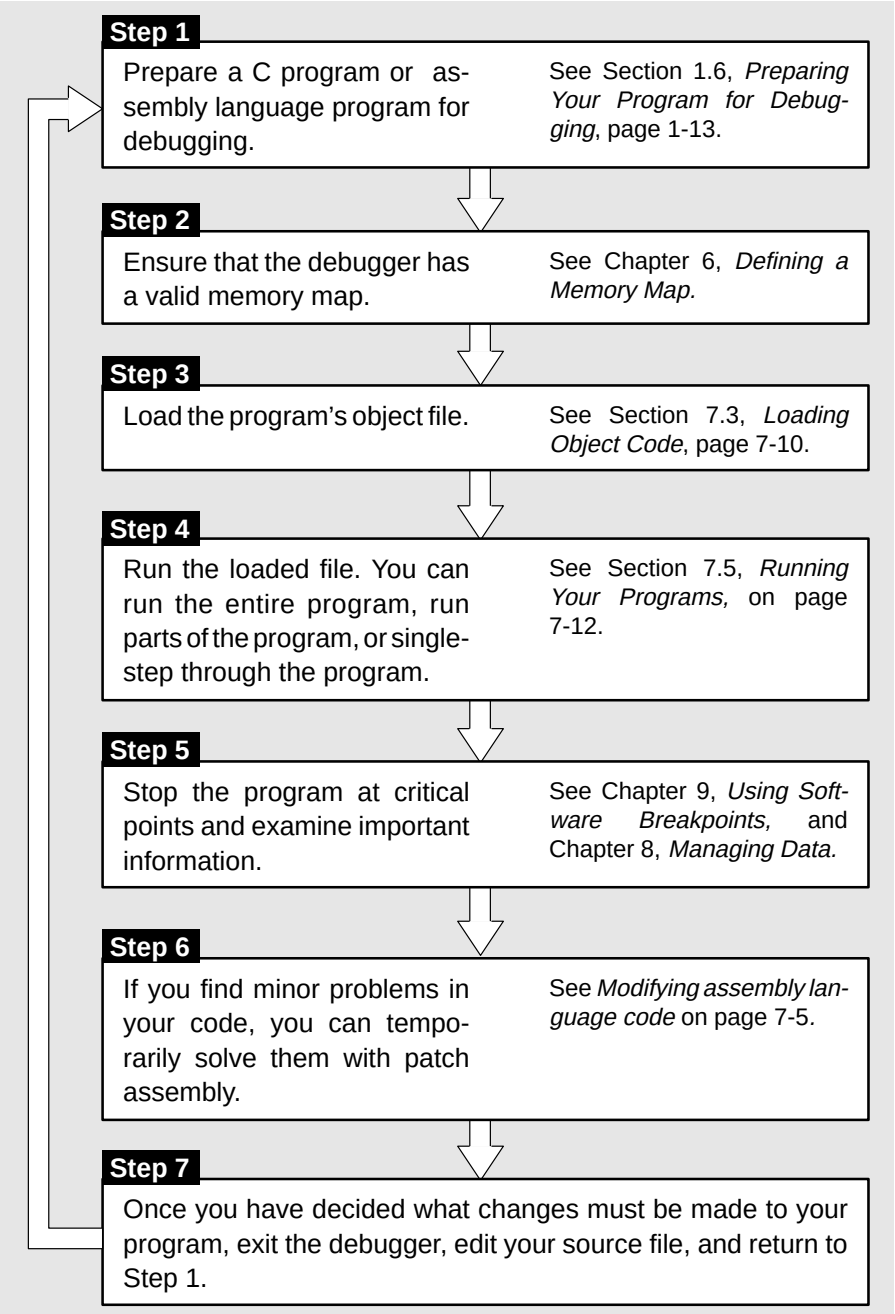
You don't need to worry about where the cursor is in the debugger window—just type. If a program is running, press ESC to halt program execution before you quit the debugger.

If you're running a standalone debugger under Microsoft Windows, you can exit the debugger by selecting the exit option from the Microsoft Windows menu bar.

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

1.10 Debugging TMS320C5xx Programs

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



Chapter 2

Using the Parallel Debug Manager

The TMS320C5xx emulation system is a true multiprocessing debugging system. It allows you to debug your entire application by using the parallel debug manager (PDM). The PDM is a command shell that controls and coordinates multiple debuggers. This chapter describes the functions that you can perform with the PDM.

Refer to Chapter 1, *Overview of a Code Development and Debugging System,* for information about invoking the PDM and debuggers.

Topic **Page**

2.1 Identifying Processors and Groups

You can send commands to an individual processor or to a group of processors. To do this, you must assign names to the individual processors or to groups of processors. Individual processor names are assigned when you invoke the individual debuggers; you can assign group names with the SET command after the individual processor names have been assigned.

Note:

Each debugger that runs under the PDM must have a unique processor name. The PDM does not keep track of existing processor names. When you send a command to a debugger, the PDM will validate the existence of a debugger invoked with that processor name.

Assigning names to individual processors

You must associate each debugger within the multiprocessing system with a unique name, referred to as a *processor name*. The processor name is used for:

- Identifying a processor to send commands to
- Assigning a processor to a group
- Setting the default prompts for the associated debuggers. For example, if you invoke a debugger with the processor name CPU_A, that debugger's prompt will be CPU_A>.
- Identifying the individual debuggers on the screen (Sun systems only). The processor name that you assign will appear at the top of the operating-system window that contains the debugger. Additionally, if you turn one of the windows into an icon, the icon name is the same as the processor name that you assigned.

To assign a processor name, you can use the –n option when you invoke a debugger. For example, to name one of the 'C5xx processors CPU_B, you would use the following command to invoke the debugger:

```
spawn emu5xx –n CPU_B ↵
```

From this point on, whenever you needed to identify this debugger, you could identify it by its processor name, CPU_B.

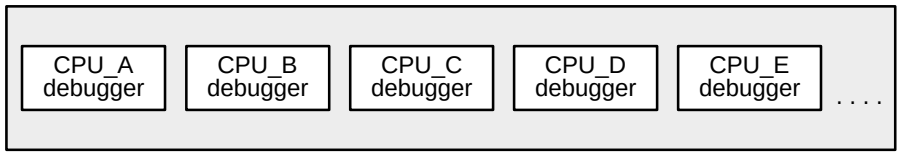
The processor name that you supply can consist of up to eight alphanumeric characters or underscore characters and must begin with an alphabetic character. Note that the name is not case sensitive. The processor name must match one of the names defined in your board configuration file (refer to Appendix C, *Describing Your Target System to the Debugger*).

Organizing processors into groups

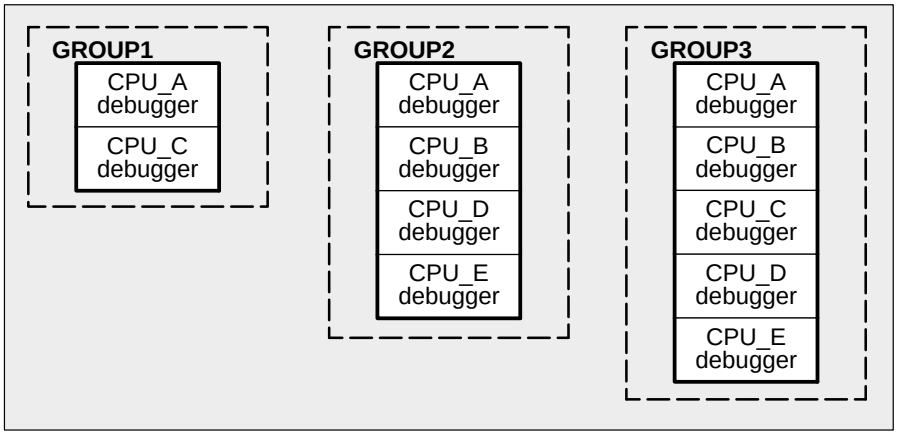
Processors can be organized into groups; these groups are identified by names defined with the SET command. Each processor can belong to any group, all groups, or a group of its own. Figure 2–1 *(a)* shows an example of processors that could exist in a system, and Figure 2–1 *(b)* illustrates three examples of named groups. GROUP1 contains two processors, GROUP2 contains four processors, and GROUP3 contains five processors.

Figure 2–1. Grouping Processors

(a) All possible processors in a system



(b) Examples of how processors could be grouped



To define and manipulate software groupings of named processors, use the SET and UNSET commands.

❑ **Defining a group of processors**

To define a group, use the SET command. The format for this command is:

set [*group name* [**=** *list of processor names*]]

This command allows you to specify a group name and the list of processors you want in the group. The *group name* can consist of up to 128 alphanumeric characters or underscore characters.

For example, to create the GROUP1 group illustrated in Figure 2–1 *(b)*, you could enter the following on the PDM command line:

```
set GROUP1 = CPU_A CPU_C ↵
```

The result is a group called GROUP1 that contains the processors named CPU_A and CPU_C. Note that the order in which you add processors to a group is the same order in which commands will be sent to the members of that group.

❑ **Setting the default group**

Many of the PDM commands can be sent to groups; if you often send commands to the same group and you want to avoid typing the group name each time, you can assign a default group.

To set the default group, use the SET command with a special group name called dgroup. For example, if you want the default group to contain the processors called CPU_B, CPU_D, and CPU_E, enter:

```
set dgroup = CPU_B CPU_D CPU_E ↵
```

The PDM will automatically send commands to the default group when you don't specify a group name.

❑ **Modifying an existing group or creating a group based on another group**

Once you've created a group, you can add processors to it by using the SET command and preceding the existing *group name* with a dollar sign ($) in the list of processors. You can also use a group as part of another group by preceding the existing group's name with a dollar sign. The dollar sign tells the PDM to use the processors listed previously in the group as part of the new list of processors.

Suppose GROUPA contained CPU_C and CPU_D. If you wanted to add CPU_E to the group, you'd enter:

```
set GROUPA = $GROUPA CPU_E ↵
```

After entering this command, GROUPA would contain CPU_C, CPU_D, and CPU_E.

If you decided to send numerous commands to GROUPA, you could make it the default group:

```
set dgroup = $GROUPA ↵
```

❑ Listing all groups of processors

To list all groups of processors in the system, use the SET command without any parameters:

```
set ↵
```

The PDM lists all of the groups and the processors associated with them:

```
GROUP1   "CPU_A CPU_C"
GROUPA   "CPU_C CPU_D CPU_E"
dgroup   "CPU_C CPU_D CPU_E"
```

You can also list all of the processors associated with a particular group by supplying a group name:

```
set dgroup ↵
dgroup   "CPU_C CPU_D CPU_E"
```

❑ Deleting a group

To delete a group, use the UNSET command. The format for this command is:

unset *group name*

You can use this command in conjunction with the SET command to remove a particular processor from a group. For example, suppose GROUPB contained CPU_A, CPU_C, CPU_D, and CPU_E. If you wanted to remove CPU_E, you could enter:

```
unset GROUPB ↵
set GROUPB = CPU_A CPU_C CPU_D ↵
```

If you want to delete all of the groups you have created, use the UNSET command with an asterisk instead of a group name:

```
unset * ↵
```

Note that the asterisk *does not* work as a wild card.

Note:

When you use UNSET * to delete all of your groups, the default group (dgroup) is also deleted. As a result, if you issue a command such as PRUN and don't specify a group or processor, the command will fail because the PDM can't find the default group name (dgroup).

2.2 Sending Debugger Commands to One or More Debuggers

The SEND command sends a debugger command to an individual processor or to a group of processors. The command is sent directly to the command interpreter of the individual debuggers. You can send any valid debugger command string.

The syntax for the SEND command is:

send [**–r**] [**–g** {*group* | *processor name*}] *debugger command*

- The **–g** option specifies the group or processor that the debugger command should be sent to. If you don't use this option, the command is sent to the default group (dgroup).
- The **–r** (return) option determines when control returns to the PDM command line:
  - **Without –r**, control is not returned to the command line until each debugger in the group finishes running code. Any results that would be printed in the COMMAND window of the individual debuggers will also be echoed in the PDM command window. These results will be displayed by the processor. For example:

    ```
    send ?pc ↵
    [CPU_C]  0x200A
    [CPU_D]  0x2008
    ```

    If you want to break out of a synchronous command and regain control of the PDM command line, press CONTROL C in the PDM window. This will return control to the PDM command line. However, no debugger executing the command will be interrupted.
  - **With –r**, control is returned to the command line immediately, even if a debugger is still executing a command. When you use –r, you *do not* see the results of the commands that the debuggers are executing.

    The –r option is useful when you want to exit from a debugger but not from the PDM. When you send the QUIT command to a debugger or group of debuggers without using the –r command, you will not be able to enter another PDM command until all debuggers that QUIT was sent to finish quitting; the PDM waits for a response from all of the debuggers that are quitting. By using –r, you can gain immediate control of the PDM and continue sending commands to the remaining debuggers.

The SEND command is useful for loading a common object file into a group of debuggers. For example, to load a file called test.out into the debuggers contained in GROUP_A, you could use the following command:

```
send –g GROUP_A load test.out ↵
```

2.3 Running and Halting Code

The PRUN, PRUNF, and PSTEP commands synchronize the debuggers to cause the processors to begin execution at the same real time.

- PRUNF starts the processors running free, which means they are disconnected from the emulator.
- PRUN starts the processors running under the control of the emulator.
- PSTEP causes the processors to single-step synchronously through assembly language code with interrupts disabled.

The formats for these commands are:

prunf [**–g** {*group* | *processor name*}]

prun [**–r**] [**–g** {*group* | *processor name*}]

pstep [**–g** {*group* | *processor name*}] [*count*]

- The **–g** option identifies the group or processor that the command should be sent to. If you don't use this option, the command is sent to the default group (dgroup).
- The **–r** (return) option for the PRUN command determines when control returns to the PDM command line:
  - **Without –r**, control is not returned to the command line until each debugger in the group finishes running code. If you want to break out of a synchronous command and regain control of the PDM command line, press CONTROL C in the PDM window. This will return control to the PDM command line. However, no debugger executing the command will be interrupted.
  - **With –r**, control is returned to the command line immediately, even if a debugger is still executing a command. You can type new commands, but the processors can't execute the commands until they finish with the current command; however, you can perform PHALT, PESC, and STAT commands when the processors are still executing.
- You can specify a *count* for the PSTEP command so that each processor in the group will step for *count* number of times.

Note:

If the current statement that a processor is pointing to has a breakpoint, that processor will not step synchronously with the other processors when you use the PSTEP command. However, that processor will still single-step.

Halting processors at the same time

You can use the PHALT command after you enter a PRUNF command to stop an individual processor or a group of processors (global halt). Each processor in the group is halted at the same real time. The syntax for the PHALT command is:

phalt [**–g** {*group* | *processor name*}]

Sending ESCAPE to all processors

Use the PESC command to send the escape key to an individual processor or to a group of processors after you execute a PRUN command. Entering PESC is essentially like typing an escape key in all of the individual debuggers. However, the PESC command is *asynchronous;* the processors don't halt at the same real time. When you halt a group of processors, the individual processors are halted in the order in which they were added to the group.

The syntax for this command is:

pesc [**–g** {*group* | *processor name*}]

Finding the execution status of a processor or a group of processors

The STAT command tells you whether a processor is running or halted. If a processor is halted when you execute this command, then the PDM also lists the current PC value for that processor. The syntax for the command is:

stat [**–g** {*group* | *processor name*}]

For example, to find the execution status of all of the processors in GROUP_A after you've executed a global PRUN, enter:

```
stat –g GROUP_A ↵
```

After entering this command, you'll see something similar to this in the PDM window:

```
[CPU_C] Running
[CPU_D] Halted   PC=201A
[CPU_E] Running
```

2.4 Entering PDM Commands

The PDM provides a flexible command-entry interface that allows you to:

- Execute PDM commands from a batch file
- Record the information shown in the PDM display area
- Conditionally execute or loop through PDM commands
- Echo strings to the PDM display area
- Pause command execution
- Repeat previously entered commands (use the command history)

This section describes the PDM commands that you can use to perform these tasks.

Executing PDM commands from a batch file

The TAKE command tells the PDM to execute commands from a batch file. The syntax for the PDM version of this command is:

take *batch filename*

The *batch filename* **must** have a .pdm extension, or the PDM will not be able to read the file. If you don't supply a pathname as part of the filename, the PDM first looks in the current directory and then searches directories named with the D_DIR environment variable.

The TAKE command is similar to the debugger version of this command (described on page 5-17). However, there are some differences when you enter TAKE as a PDM command instead of a debugger command.

- **Similarities.** As with the debugger version of the TAKE command, you can nest batch files up to 10 deep.
- **Differences.** Unlike the debugger version of the TAKE command:
  - There is no suppress-echo-flag parameter. Therefore, all command output is echoed to the PDM window, and this behavior cannot be changed.
  - To halt batch-file execution, you must press CONTROL C instead of ESC.
  - The batch file must contain only PDM commands (no debugger commands).

The TAKE command is advantageous for executing a batch file in which you have defined often-used aliases. Additionally, you can use the SET command in a batch file to set up group configurations that you use frequently, and then execute that file with the TAKE command. You can also put your flow-control commands (described in *Controlling PDM command execution on* page 2-10) in a batch file and execute the file with the TAKE command.

Recording information from the PDM display area

By using the DLOG command, you can record the information shown in the PDM display area into a log file. This command is identical to the debugger DLOG command described on page 5-6.

- To begin recording the information shown in the PDM display area, use:

  dlog *filename*

  This command opens a log file called *filename* that the information is recorded into. If you plan to execute the log file with the TAKE command, the filename *must* have a .pdm extension.

- To end the recording session, enter:

  dlog close ↵

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

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

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

- **Appending to an existing file.** Use the **a** parameter to open an existing file and append the information in the display area.
- **Writing over an existing file.** Use the **w** parameter to open an existing file and write over the current contents of the file. 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.

Controlling PDM command execution

You can control the flow of PDM commands in a batch file or interactively. With the IF/ELIF/ELSE/ENDIF or LOOP/BREAK/CONTINUE/ENDLOOP flow-control commands, you can conditionally execute debugger commands or set up a looping situation, respectively.

- To conditionally execute PDM commands, use the IF/ELIF/ELSE/ENDIF commands. The syntax is:

  if *expression*
  PDM commands
  [**elif** *expression*
  PDM commands]
  [**else**
  PDM commands]
  endif

- If the expression for the IF is nonzero, the PDM executes all commands between the IF and the ELIF, ELSE, or ENDIF.
- The ELIF is optional. If the expression for the ELIF is nonzero, the PDM executes all commands between the ELIF and the ELSE or ENDIF.
- The ELSE is optional. If the expressions for the IF and ELIF (if present) are false (zero), the PDM executes the commands between the ELSE and the ENDIF.

❑ To set up a looping situation to execute PDM commands, use the LOOP/BREAK/CONTINUE/ENDLOOP commands. The syntax is:

loop *Boolean expression*
PDM commands
[**break**]
[**continue**]
endloop

The PDM version of the LOOP command is different from the debugger version of this command (described on page 14-32). Instead of accepting any expression, the PDM version of the LOOP command evaluates only Boolean expressions. If the Boolean expression evaluates to true (1), the PDM executes all commands between the LOOP and the BREAK, CONTINUE, or ENDLOOP. If the Boolean expression evaluates to false (0), the loop is not entered.

- The optional BREAK command allows you to exit the loop without having to reach the ENDLOOP. This is helpful when you are testing a group of processors and want to exit if an error is detected.
- The CONTINUE command, which is also optional, acts as a goto and returns command flow to the enclosing LOOP command. CONTINUE is useful when the part of the loop that follows is complicated, and returning to the top of the loop avoids further nesting.

You can enter the flow-control commands interactively or include the commands in a batch file that is executed by the TAKE command. When you enter LOOP or IF from the PDM command line, a question mark (?) prompts you for the next entry:

```
PDM:11>>if $i > 10 ↵
?echo ERROR IN TEST CASE ↵
?endif ↵
ERROR IN TEST CASE

PDM:12>>
```

The PDM continues to prompt you for input using the ? until you enter ENDIF (for an IF command) or ENDLOOP (for a LOOP command). After you enter ENDIF or ENDLOOP, the PDM immediately executes the IF or LOOP command.

If you are in the middle of interactively entering a LOOP or IF statement and want to abort it, type CONTROL C.

You can use the IF/ENDIF and LOOP/ENDLOOP commands together to perform a series of tests. For example, within a batch file, you can create a loop like the following (the SET and @ commands are described in Section 2.8, beginning on page 2-18):

```
set i = 10                                  Set the counter (i) to 10.
loop $i > 0                                 Loop while i is greater than 0.
  .
  test commands
  .
  if $k > 500                               Test for error condition.
    echo ERROR ON TEST CASE 8               Display an error message.
  endif
  .
  @ i = $i – 1                              Decrement the counter.
endloop
```

You can record the results of this loop in a log file (refer to page 2-10) to examine which test cases failed during the testing session.

Echoing strings to the PDM display area

You can display a string in the PDM display area by using the ECHO command. This command is especially useful when you are executing a batch file or running a flow-control command such as IF or LOOP. The syntax for the command is:

echo *string*

This displays the *string* in the PDM display area.

You can also use ECHO to show the contents of a system variable (system variables are described in Section 2.8):

```
echo $var_proc1 ↵
34
```

The PDM version of the ECHO command works in exactly the same way as the debugger version described on page 5-18 works, except that you can use the PDM version outside of a batch file.

Pausing command execution

Sometimes you may want the PDM to pause while it's running a batch file or when it's executing a flow control command such as LOOP/ENDLOOP. Pausing is especially helpful in debugging the commands in a batch file.

The syntax for the PAUSE command is:

pause

When the PDM reads this command in a batch file or during a flow control command segment, the PDM stops execution and displays the following message:

```
<< pause – type return >>
```

To continue processing, press ⏎.

Using the command history

The PDM supports a command history that is similar to the UNIX command history. The PDM prompt identifies the number of the current command. This number is incremented with every command. For example, PDM:12>> indicates that eleven commands have previously been entered, and the PDM is now ready to accept the twelfth command.

The PDM command history allows you to re-enter any of the last twenty commands:

- To repeat the last command that you entered, type:

  !! ⏎

- To repeat any of the last twenty commands, use the following command:

  !*number*

  number is the number of the PDM prompt that contains the command that you want to re-enter. For example,

  ```
  PDM:100>>echo hello ⏎
  hello
  PDM:101>>echo goodbye ⏎
  goodbye
  PDM:102>>!100 ⏎
  echo hello
  hello
  ```

  Notice that the PDM displays the command that you are re-entering.

- An alternate way to repeat any of the last twenty commands is to use:

  !*string*

  This command tells the PDM to execute the last command that began with *string.* For example,

```
PDM:103>>pstep –g GROUPA ↵
PDM:104>>send –g GROUPA ?pc ↵
[CPU_C]  0x2000
[CPU_D]  0x2004
PDM:103>>pstep –g GROUPB ↵
PDM:104>>send –g GROUPB ?pc ↵
[CPU_A]  0x201A
[CPU_E]  0x2014
PDM:105>>!p ↵
pstep –g GROUPB
```

- To see a list of the last twenty commands that you entered, type:

  history ↵

The command history for the PDM works differently from that of the debugger (described on page 5-5); the TAB and F2 keys have no command-history meaning for the PDM.

2.5 Defining Your Own Command Strings

The ALIAS command provides a shorthand method of entering often-used commands or command sequences. The UNALIAS command deletes one or more ALIAS definitions. The syntax for the PDM version of each of these commands is:

alias [*alias name* [**,** "*command string*"]]
unalias {*alias name* | *}

The PDM versions of the ALIAS and UNALIAS commands are similar to the debugger versions of these commands. You can:

- Include several commands in the command string by separating the individual commands with semicolons
- Define parameters in the command string by using a percent sign and a number (%1, %2, etc.) to represent a parameter whose value will be supplied when you execute the aliased command
- List all currently defined PDM aliases by entering ALIAS with no parameters
- Find the definition of a PDM alias by entering ALIAS with only an alias-name parameter
- Nest alias definitions
- Redefine an alias
- Delete a single PDM alias by supplying the UNALIAS command with an alias name or delete all PDM aliases by entering UNALIAS *

Like debugger aliases, PDM alias definitions are lost when you exit the PDM. However, individual commands within a PDM command string don't have an expanded-length limit.

For more information about these features, refer to Section 5.5, *Defining Your Own Command Strings*.

The PDM version of this command is especially useful for aliasing often-used command strings involving the SEND and SET commands.

- You can use the ALIAS command to create PDM versions of debugger commands. For example, the ML debugger command lists the memory ranges that are currently defined. To make a PDM version of the ML command to list the memory ranges of all the debuggers in a particular group, enter:

  ```
  alias ml, "send -g %1 ml" ↵
  ```

You could then list the memory maps of a group of processors such as those in group GROUPA:

```
ml GROUPA ↵
```

- The ALIAS command can be helpful if you frequently change the default group. For example, suppose you plan to switch between two groups. You can set up the following alias:

  ```
  alias switch, "set dgroup $%1; set prompt %1" ↵
  ```

  The %1 parameter will be filled in with the group information that you enter when you execute SWITCH. Notice that the %1 parameter is preceded by a dollar sign ($) to set up the default group. The dollar sign tells the PDM to evaluate (take the list of processor names defined in the group instead of the actual group name). However, to change the prompt, you don't want the PDM to evaluate (use the processors associated with the group name as the prompt)—you just want the group name. As a result, you don't need to use the dollar sign when you want to use only the group name.

  Assume that GROUP3 contains CPU_A, CPU_B, and CPU_D. To make GROUP3 the current default group and make the PDM prompt the same name as your default group, enter:

  ```
  switch GROUP3 ↵
  ```

  This causes the default group (dgroup) to contain CPU_A, CPU_B, and CPU_D, and it changes the PDM prompt to GROUP3:x>>.

2.6 Entering Operating-System Commands

The SYSTEM command provides you with a method of entering operating-system commands. The format for the PDM version of this command is:

system *operating-system command*

The SYSTEM command is similar to the debugger's SYSTEM command (described on page 5-24), but there are some differences.

- **Similarities.** You can enter operating-system commands without having to leave the primary environment (in this case, the PDM) and without having to open another operating-system window.
- **Differences.** Unlike the debugger version of the SYSTEM command:
  - The PDM version of the SYSTEM command cannot be entered without an operating-system command parameter. Therefore, you cannot use the command to open a shell.
  - There is no flag parameter; command output is always displayed in the PDM window.

2.7 Understanding the PDM's Expression Analysis

The PDM analyzes expressions differently than individual debuggers do (expression analysis for the debugger is described in Chapter 15, *Basic Information About C Expressions*). The PDM uses a simple integral expression analyzer. You can use expressions to cause the PDM to make decisions as part of the @ command and the flow control commands (described on pages 2-19 and 2-10, respectively).

Note that you cannot evaluate string variables with the PDM expression analyzer. You can evaluate only constant expressions.

Table 2–1 summarizes the PDM operators. The PDM interprets the operators in the order in which they're listed in Table 2–1 (left to right, top to bottom).

Table 2–1. PDM Operators

Operator	Definition	Operator	Definition
( )	take highest precedence	*	multiplication
/	division	%	modulo
+	addition (binary)	–	subtraction (binary)
< <	left shift	~	complement
<	less than	> >	right shift
>	greater than	< =	less than or equal to
= =	is equal to	> =	greater than or equal to
&	bitwise AND	! =	is not equal to
|	bitwise OR	^	bitwise exclusive-OR
| |	logical OR	&&	logical AND

2.8 Using System Variables

You can use the SET, @, and UNSET commands to create, modify, and delete system variables. In addition, you can use the SET command with system-defined variables.

Creating your own system variables

The SET command lets you create system variables that you can use with PDM commands. The syntax for the SET command is:

set [*variable name* [**=** *string*]]

The *variable name* can consist of up to 128 alphanumeric characters or underscore characters.

For example, suppose you have an array that you want to examine frequently. You can use the SET command to define a system variable that represents that array value:

```
set result = ar1[0] + 100 ↵
```

In this case, result is the variable name, and ar1[0] + 100 is the expression that will be evaluated whenever you use the variable result.

Once you have defined result, you can use it with other PDM commands, such as the SEND command:

```
send CPU_D ? $result ↵
```

The dollar sign ($) tells the PDM to replace result with ar1[0] + 100 (the string defined in result) as the expression parameter for the ? command. You *must* precede the name of a system variable with a $ when you want to use the string value you defined with the variable as a parameter.

You can also use the SET command to concatenate and substitute strings.

❑ **Concatenating strings**

The dollar sign followed by a system variable name enclosed in braces (**{** and **}**) tells the PDM to append the contents of the variable name to a string that precedes or follows the braces. For example:

```
set k = Hel ↵                          Set k to the string Hel.
set i = ${k}lo ${k}en ↵                Concatenate the contents of k before
                                       lo and en, and set the result to i.
echo $i ↵                              Show the contents of i.
Hello Helen
```

❑ **Substituting strings**

You can substitute defined system variables for parts of variable names or strings. This series of commands illustrates the substitution feature:

```
set err0 = 25 ↵                    Set err0 to 25.
set j = 0 ↵                        Set j to 0.
echo $err$j ↵                      Show the value of $err$j → $err0 → 25.
25
```

Note that substitution stops when the PDM detects recursion (for example, $k = k).

Assigning a variable to the result of an expression

The @ (substitute) command is similar to the SET command. You can use the @ command to assign the result of an expression to a variable. The syntax for the @ command is:

@ *variable name* **=** *expression*

The following series of commands illustrates the differences between the @ command and the SET command. Assume that mask1 equals 36 and mask2 equals 47.

```
set mask3 = $mask1+$mask2 ↵        Set mask3 to the contents of mask1
                                   plus the contents of mask2.
echo $mask3 ↵                      Show the contents of mask3.
36+47
@ mask3 = $mask1+$mask2 ↵          Set mask3 to the result of the
                                   expression $mask1+$mask2.
echo $mask3 ↵                      Show the contents of mask3.
83
```

Notice the difference between the two commands. The SET command lets you create system variables that you can use with PDM commands. The @ command evaluates the expression and assigns the result to the variable name.

The @ command is useful in setting loop counters. For example, you can initialize a counter with the following command:

```
@ j = 0 ↵
```

Inside the loop, you can increment the counter with the following statement:

```
@ j = $j + 1 ↵
```

Changing the PDM prompt

The PDM recognizes a system variable called prompt. You can change the PDM prompt by setting the prompt variable to a string. For example, to change the PDM prompt to 3PROCs, enter:

```
set prompt = 3PROCs ↵
```

After entering this command, the PDM prompt will look like this: 3PROCs:x>>.

Checking the execution status of the processors

In addition to displaying the execution status of a processor or group of processors, the STAT command (described on page 2-8) sets a system variable called status.

- If *all* of the processors in the specified group are running, the status variable is set to 1.
- If one or more of the processors in the group is halted, the status variable is set to 0.

You can use this variable when you want an instruction loop to execute until a processor halts:

```
loop stat == 1
send ?pc
.
.
```

Listing system variables

To list all system variables, use the SET command without parameters:

```
set ↵
```

You can also list the contents of a single variable. For example,

```
set j ↵
j  ”100”
```

Deleting system variables

To delete a system variable, use the UNSET command. The format for this command is:

unset *variable name*

If you want to delete all of the variables you have created and any groups you have defined (as described on page 2-3), use the UNSET command with an asterisk instead of a variable name:

```
unset * ↵
```

> **Note:**
>
> When you use UNSET * to delete all of your system variables and processor groups, variables such as prompt, status, and dgroup are also deleted.

2.9 Evaluating Expressions

The debugger includes an EVAL command that evaluates an expression (see Section 8.2, *Basic Commands for Managing Data,* for more information about the debugger version of the EVAL command). The PDM has a similar command called EVAL that you can send to a processor or a group of processors. The EVAL command evaluates an expression in a debugger and sets a variable to the result of the expression. The syntax for the PDM version of the EVAL command is:

eval [**–g** {*group* | *processor name*}] *variable name***=***expression*[**,** *format*]

- The **–g** option specifies the group or processor that EVAL should be sent to. If you don't use this option, the command is sent to the default group (dgroup).
- When you send the EVAL command to more than one processor, the PDM takes the *variable name* that you supply and appends a suffix for each processor. The suffix consists of the underscore character (_) followed by the name that you assigned the processor. That way, you can differentiate between the resulting variables.
- The *expression* can be any expression that uses the symbols described in Section 2.7.
- When you use the optional *format* parameter, the value that the variable is set to will be in one of the following formats:

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

Suppose the program that CPU_A is running has two variables defined: j is equal to 5, and k is equal to 17. Also assume that the program that CPU_B is running contains variables j and k: j is equal to 12, and k is equal to 22.

```
set dgroup = CPU_A CPU_B ↵
eval val = j + k ↵
set ↵
dgroup     "CPU_A CPU_B"
val_CPU_A  "23"
val_CPU_B  "34"
```

Notice that the PDM created a system variable for each processor: val_CPU_A for CPU_A and val_CPU_B for CPU_B.

Chapter 3

An Introductory Tutorial to the C Source Debugger

This chapter provides a step-by-step, hands-on demonstration of the 'C5xx 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 PC, performing the lessons in the order that they're presented. The tutorial contains two sets of lessons (11 in the first set, 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.

Important! If you are using the EVM or emulator, this step assumes that you are using the default I/O switches or that you have identified the I/O switches 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:

  `emu5xx c:\c5xxhll\sample`↵

- For the **EVM**, enter:

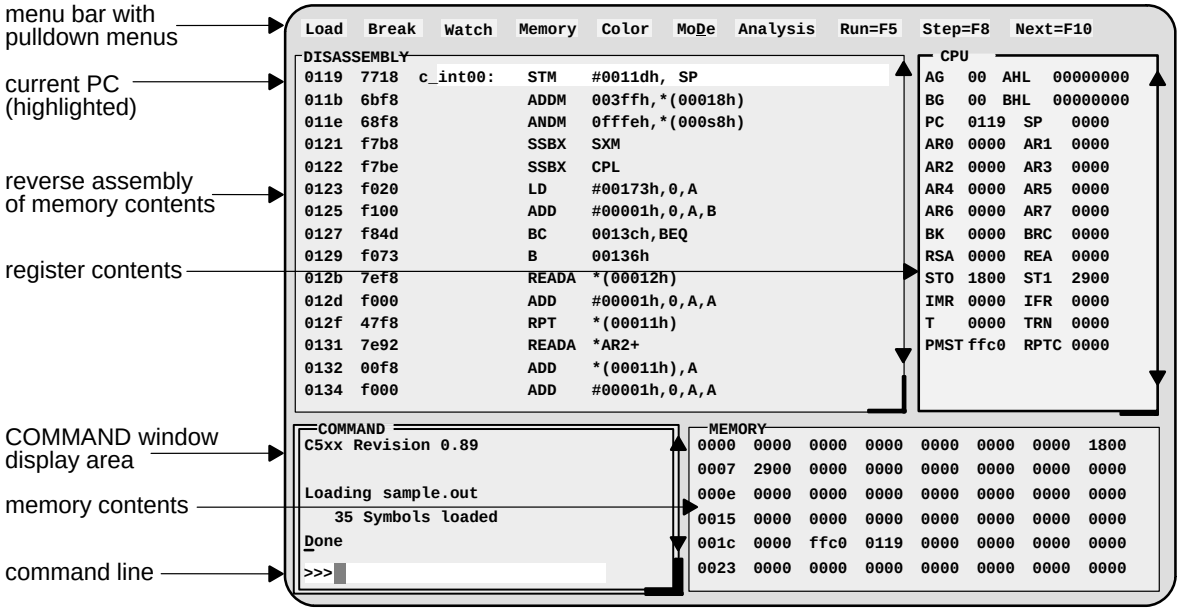
  `evm5xx c:\c5xxhll\sample` ↵

- For the **simulator**, enter:

  `sim5xx c:\sim5xx\sample` ↵

Take a look at the display. . .

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



- ❑ 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 'C5xx processor:

     `reset` ↵

  2) Load the sample program again:

     `load c:\c5xxhll\sample` ↵ (EVM and emulator)

     `load c:\sim5xx\sample` ↵ (simulator)

- ❑ If you see a display and the first few lines of the DISASSEMBLY window still show ADD instructions or say *Invalid address* after resetting the 'C5xx processor, your EVM or emulator board may not be installed snugly. Check your board to see if it is correctly installed, and reenter 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.out 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 0x0119.

> Modify the MEMORY display to show the same object code that is displayed in the DISASSEMBLY window:
>
> **mem 0x0119@prog** ↵

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

Try This: The 'C5xx has separate program and data spaces. You can access either program or data memory by following the location with **@prog** for program memory or **@data** for data memory. If you'd like to see the contents of location 0x800 in data memory, enter:

mem 0x800@data ↵

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

mem PC@prog ↵

Select the active window

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

lesson continues on the next page →

Make the CPU window the active window:

`win CPU` ↵

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

Important! If you don't see a change in the appearance of the CPU window, look at the way you entered the command. Did you enter *CPU* in uppercase letters? For this command, it's important that you enter the parameter in uppercase, as shown.

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

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

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

2) Click the left mouse button.

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

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

  Press ESC to get out of this.

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

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

Resize the active window

This lesson shows you how to resize the active window.

Important! Be sure the CPU window is still active.

Make the CPU window as small as possible:

`size 4,3` ↵

This tells the debugger to make the window 4 characters by 3 lines, which is the smallest a window can be. (If it were any smaller, the debugger wouldn't be able to display all four corners of the window.) If you try to enter smaller values, the debugger will warn you that you've entered an *Invalid window size*. The maximum width and length depend on which 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 ZOOM command 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 –b 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 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 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 14.5, page 14-68, summarizes the functions of all the special keys and key sequences and how they affect the different windows.

Display the C source version of the sample file

Now that you can find your way around the debugger interface, you can become familiar with some of the debugger's more significant features. It's time to load 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 the first several statements in the DISASSEMBLY window are highlighted because these statements are associated with the current C statement (which is highlighted in the FILE window).
- The CALLS window, which tracks functions as they're called, now points to main().
- The values of the PC and SP (and possibly some additional registers) are highlighted in the CPU window because they were changed by program execution.

Become familiar with the three debugging modes

The debugger has three basic debugging modes:

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

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

The following steps show you how to switch debugging modes.

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

1) Look at the top of the display: the first line shows a row of pull-down menu selections.

2) Point to the word MoDe on the menu bar.

3) Press the left mouse button, but don't release it; drag the mouse downward until Asm (the second entry) is highlighted.

4) Release the button.

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

Switch to auto mode:

1) Press [ALT][D]. This displays and freezes the MoDe menu.
2) Now select C(auto). 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 your 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 ↵
```

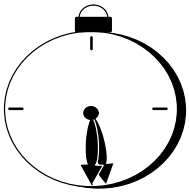
You're still in auto mode, but you should now see the DISASSEMBLY window. The current PC should be at the statement that defines the meminit label.

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

asm switches to assembly 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 3-3). Then turn to page 3-14 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` ↵

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` ↵

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` ↵

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*.

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` ↵

Set a software breakpoint and run your program:

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) Reset the program entry point:

   `restart` ↵

3) Enter the run command:

   `run` ↵ *Program execution halts at the breakpoint.*

Once again, you should see that some statements are highlighted in the CPU window, showing that they were changed by program execution. But this time, you know that the changes were caused by code from the beginning of the program to line 38 in the FILE window.

Clear the breakpoint:

1) Point the mouse cursor at the statement on line 38. (It should still be highlighted from setting the breakpoint.)

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

Benchmark a section of code

If you're using the 'C5xx emulator, EVM, or simulator, you can use breakpoints to help you benchmark a section of code. This means that you'll ask the debugger to count the number of CPU clock cycles that are consumed by a certain portion of code.

Benchmark some code:

1) In sample.c (displayed in the FILE window), set two software breakpoints: one at line 38 (the meminit() statement) and one at line 65 (the xcall(value) statement).

2) Reset the program entry point:

   `restart` ↵

3) Enter the run command:

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

4) Enter the runb command:

   `runb` ↵ *This runs to the second breakpoint (this may take several seconds).*

5) Now use the ? command to examine the contents of the CLK pseudoregister:

   `? 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 software 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 software 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 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 ar1, Stack Pointer ↵
wa pc ↵
wa *0x2059@prog, 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.

lesson continues on the next page →

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 20` ↵

Try This: Notice that the step command single-stepped each assembly language statement (in fact, you single-stepped through 20 assembly language statements). Did you also notice that the FILE window displayed the source for the meminit() function when it was called? The debugger supports additional single-step commands that have a slightly different flavor.

- For example, if you enter:

  `cstep 20` ↵

  you'll single-step 20 *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 20 assembly language statements, *but the FILE window won't display the source for the meminit() function when meminit() is executed.*

  `next 20` ↵

  (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, 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 ↵
```

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.

Set up for the conditional run examples:

1) Set software breakpoints at lines 39 and 41.
2) Reset the program entry point:

   ```
   restart ↵
   ```

3) Run the first part of the program:

   ```
   go main ↵
   ```

4) Reset the value of i:

   ```
   ?i=0 ↵
   ```

Now initiate the conditional run:

```
run i<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.

lesson continues on the next page →

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 {                          tiny6 is a structure.
          int u;
          int v;
          int x;
          int y;
          int z;
      } tiny6;
```
whatis call ↵
```
      int call();       call is a function that returns an integer.
```
whatis s ↵
```
      short s;                    s is a short unsigned integer.
```
whatis zzz ↵
```
      struct zzz {                    zzz is a very long structure.
          int b1;
          int b2;
```
Press 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 c5xxhll` or **`cd sim5xx`**	*Change back to the debugger directory.*

Display the contents of an aggregate data type

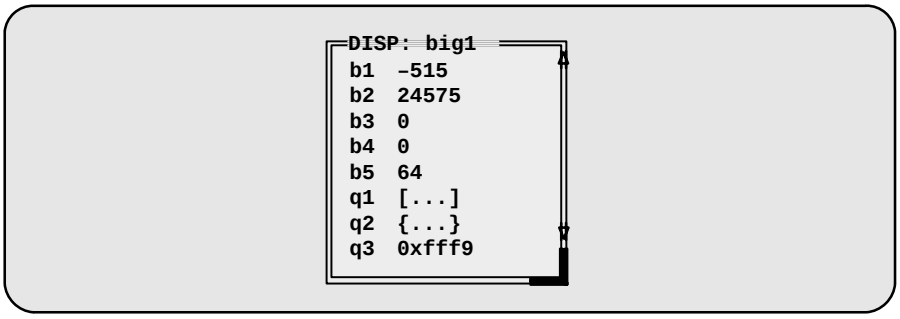
The WATCH window is convenient for watching single, or *scalar*, values. When you're debugging a C program, though, you may need to observe values that aren't scalar; for example, you might need to observe the effects of program execution on an array. The debugger provides another type of window called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

lesson continues on the next page →

Now you should see a display like the one below. The newly opened DISP window becomes the active window. In the DISP window, as in 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.



(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.

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.)

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 0x800 as type float (exponential floating-point format).

Display memory contents in floating-point format:

```
disp *(float *)0x800 ↵
```

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 0x800—*it isn't memory location 0*. Note that you can scroll through the memory displayed in the DISP window; item [1] is at 0x801, and item [–1] is at 0x07fe.

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 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 ↵
```

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, MEWi ↵
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.*

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 0x800:

   `mem 0x800` ↵

↖ 2) Point to the contents of memory location 0x800.

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

4) Type 0000.

5) Press ↵ to enter the new value.

6) Press ESC to conclude editing.

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

1) Make the CPU window the active window:

   `win CPU` ↵

↑↓ 2) Press the arrow keys until the field cursor (_) points to the PC contents.

F9 3) Press F9.

4) Type 2000.

↓ 5) Press ↓ enough times to point at the contents of register AR0.

6) Type ffff.

↵ 7) Press ↵ to enter the new value.

ESC 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 c5xxhll or sim5xx 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. The 'C5xx supports separate program and data spaces. Page 0 in the memory map is for program memory; page 1 is for data memory.

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

Change the memory map:

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

   md 0x0,0 ↵

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

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

   ma 0xc000,0,0x20,ROM ↵
   ma 0x100,1,0x7f,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 0xc000,0,0x20,ROM;`
   `ma 0x100,1,0x7f,RAM;ml"` ↵

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

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

   `mymap` ↵

   This is equivalent to entering the following four commands:

```
mr
ma 0x0,0,0x20,ROM
ma 0x100,1,0x7f,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 4

The Debugger Display

The TMS320C5xx C source debugger has a window-oriented display. This chapter shows what windows can look like and describes the basic types of windows that you'll use.

Topic **Page**

4.1 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.

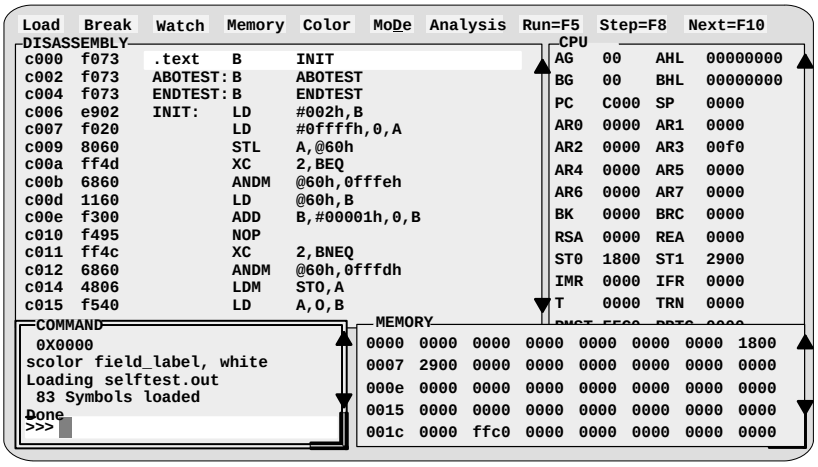
Auto mode

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 the one in Figure 4–1.

Auto mode has two types of displays:

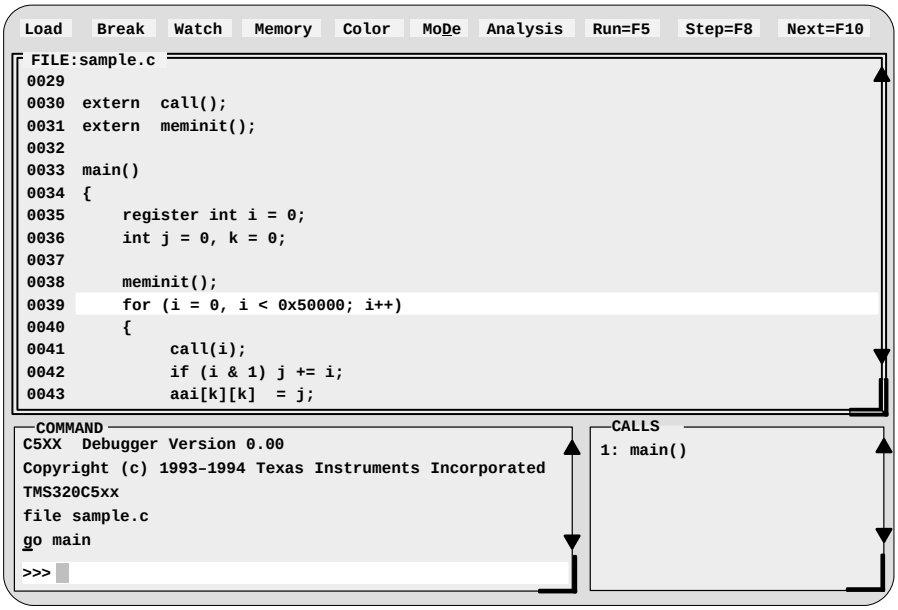
- When the debugger is running assembly language code, you'll see an assembly display similar to the one in Figure 4–1. The DISASSEMBLY window displays the reverse assembly of memory contents.

Figure 4–1. 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 4–2. (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 4–2. Typical C Display (for Auto Mode Only)



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

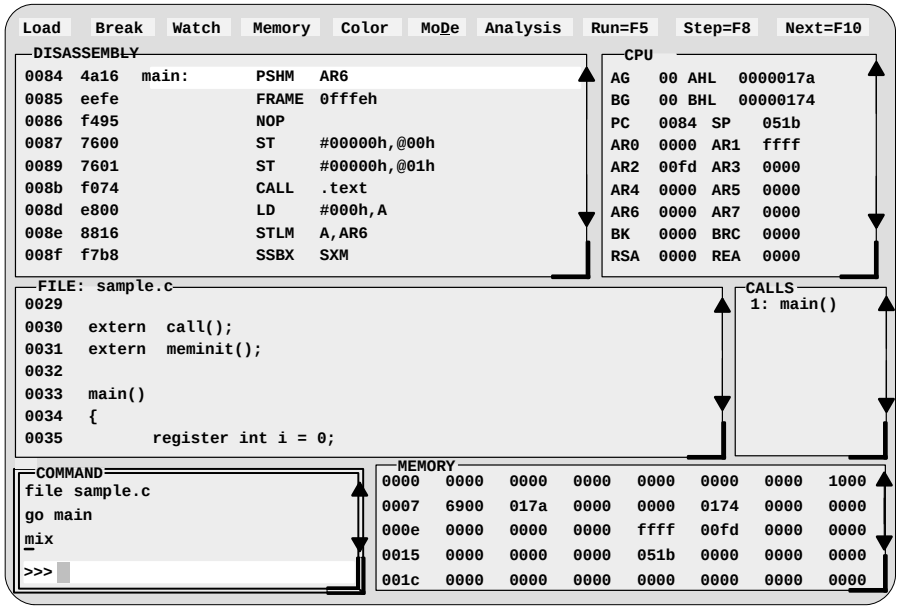
Assembly mode is for viewing assembly language programs only. In this mode, you'll see a display similar to the one shown in Figure 4–1. When you're in assembly mode, you'll always see the assembly display, regardless of whether C or assembly language is currently running.

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

Mixed mode

Mixed mode is for viewing assembly language and C code at the same time. Figure 4–3 shows the default display for mixed mode.

Figure 4–3. 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 'C5xx.

Restrictions associated with debugging modes

The assembly language code that the debugger shows you is the disassembly (reverse assembly) of 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

4.2 Descriptions of the Different Kinds of Windows and Their Contents

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

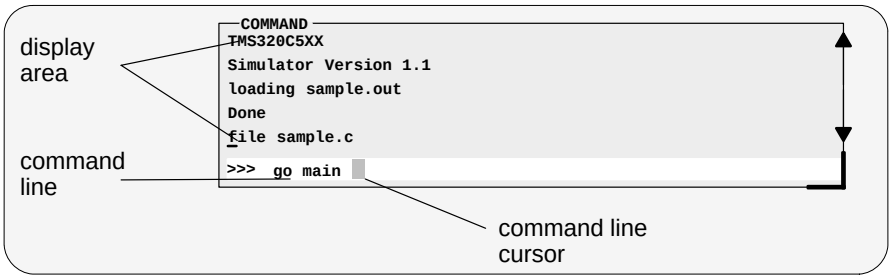
Every window is identified by a name in its upper left corner. Each type of window serves a specific purpose and has unique characteristics. There are nine windows, divided into four general categories:

- 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:
  - The MEMORY windows display 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 TMS320C5xx registers.
  - The DISP windows display 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 4.4, *The Active Window*, on page 4-19.

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:

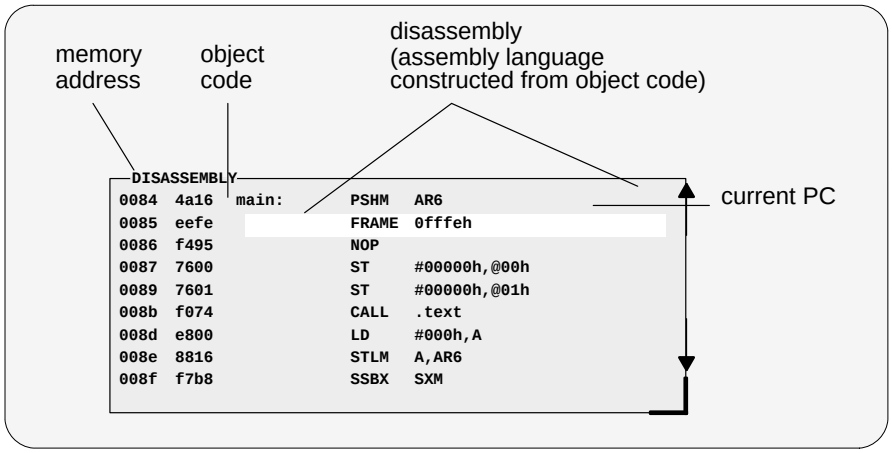
- ❑ **Command line.** This is where you enter commands. When you want to enter a command, just type—no matter which window is active. (There are a few instances, such as when you're editing a field, in which you can't type a command; see page 5-4 for more information.)

  The debugger keeps a list of the last 50 commands that you entered. You can select and reenter commands from the list without retyping them. (For more information on using the command history, see *Using the command history,* page 5-5.)

- ❑ **Display area**. This area of the COMMAND window echoes the command that you entered, shows any output from the command, and displays debugger messages.

For more information about the COMMAND window and entering commands, refer to Chapter 5, *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.

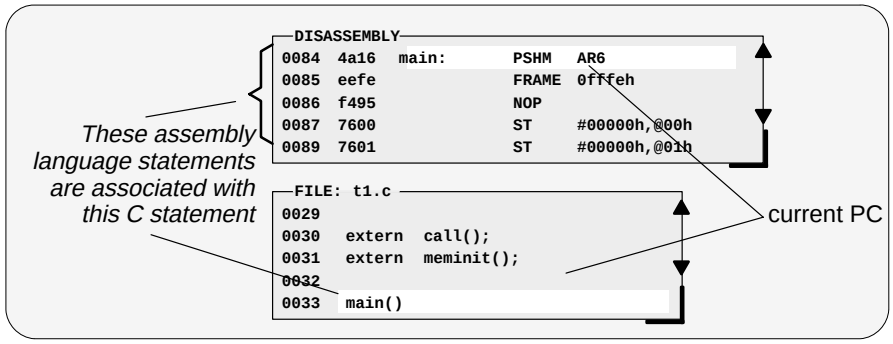
Modes Auto (assembly display only), assembly, and mixed

Created Automatically

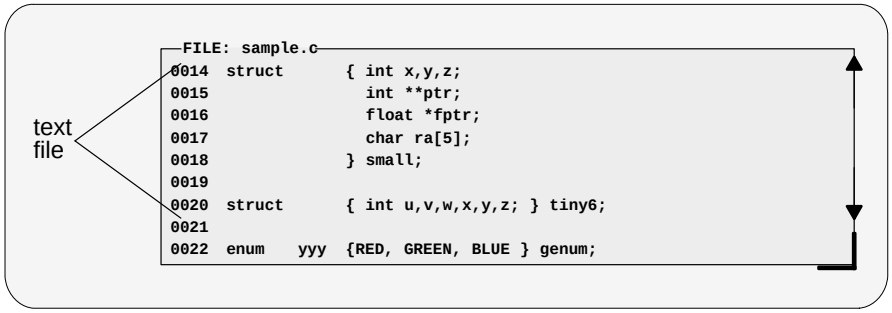
Affected by
- ❑ DASM and ADDR commands
- ❑ Breakpoint and run commands

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 the FILE command
- ❑ Automatically when you're in auto or mixed mode and your program begins executing C code

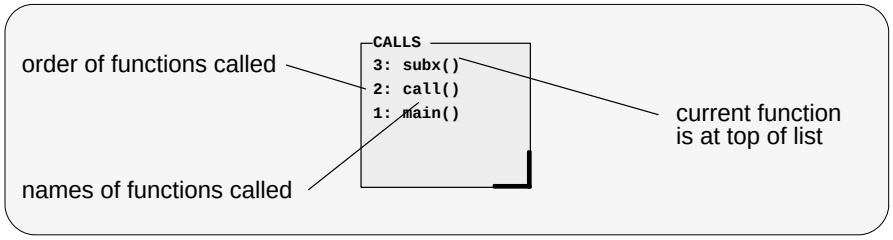
Affected by
- ❑ FILE, FUNC, and ADDR commands
- ❑ Breakpoint and run commands

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

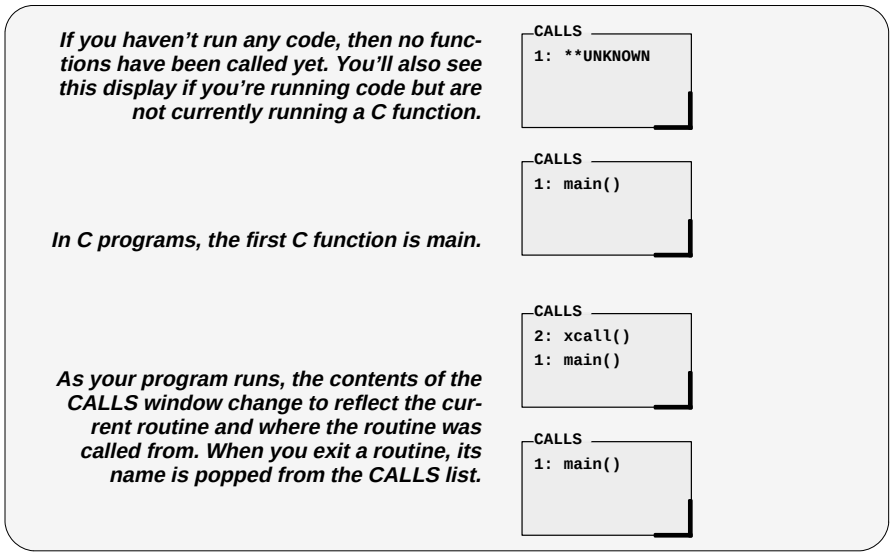
Editable? No; pressing the edit key (F9) or the left mouse button changes the FILE window display to show the source associated with the called function.

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 4.4, *The Active Window*, on page 4-19).

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

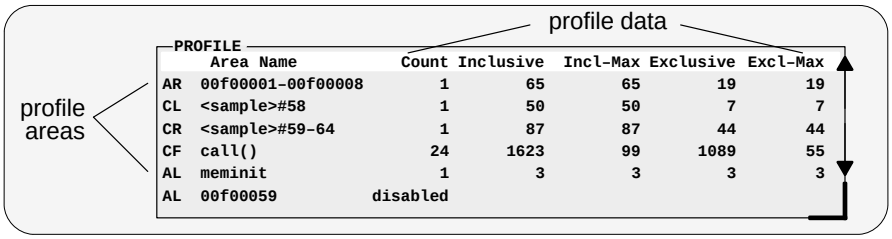
3) Press F9. This displays the selected function in the FILE window.

You can close and reopen the CALLS window.

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

  calls

PROFILE window



Purpose Displays statistics collected during a profiling session

Editable? No

Modes Auto

Created By invoking the debugger with the –profile option

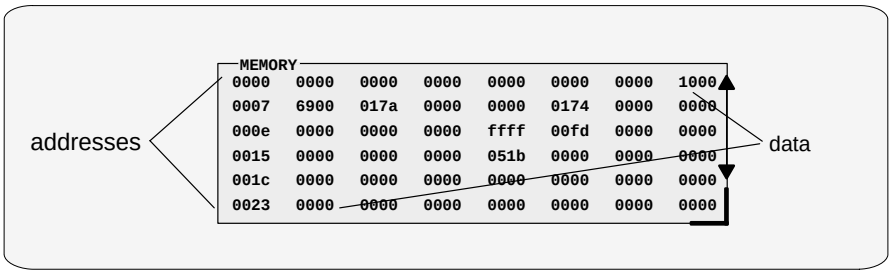
Affected by
- The PF and PQ commands
- Any commands on the View menu
- Clicking in the header area of the window

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 window, 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 13, *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

The MEMORY window has two parts:

- ❑ **Addresses.** The first column of numbers identifies the addresses of the first column of displayed data. No matter how many data columns you display, only one address column is displayed. Each address in this column identifies the address of the data immediately to its right.

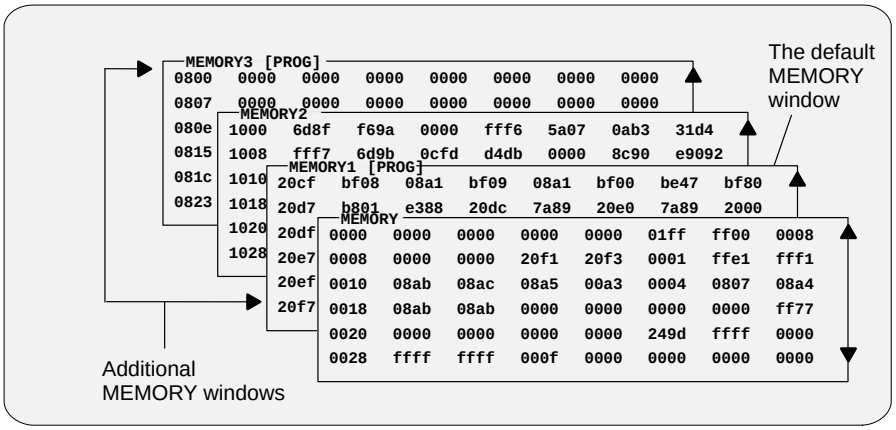
- ❑ **Data.** The remaining columns display the values at the listed addresses. You can display more data by making the window wider and/or longer.

  The MEMORY window above has seven columns of data, so each new address is incremented by 7. Although the window shows seven columns of data, there is only one column of addresses; the first value is at address 0x0000, the second at address 0x0001, etc.; the eighth value (first value in the second row) is at address 0x0007, the ninth at address 0x0008, etc.

As you run programs, some memory values change as a 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, as illustrated in Figure 4–4.

Figure 4–4. 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 0x800, you would enter:

```
mem1 0x0800
```

This displays a new window, MEMORY1, showing the contents of memory starting at the address 0x0800.

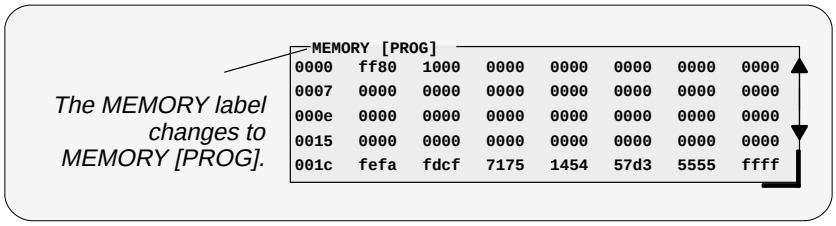
The ’C5xx has separate data, program, and I/O spaces. By default, the MEMORY window shows data memory. If you want to display program memory, you can enter the MEM command like this:

mem[#] *address@***prog**

The @**prog** suffix identifies the *address* as a program memory address. You can also use @**data** to display data memory. However, if you are displaying data memory, the @data is unnecessary since data memory is the default. If you are using an emulator or EVM, you can display I/O space by using @**IO**.

When you display program memory, the MEMORY window's label changes to remind you that you are no longer displaying data memory, as shown in Figure 4–5.

Figure 4–5. Default Data Memory Screen



❑ **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 0x0800 in the MEMORY1 window	**mem1** 0x0800
View another block of memory starting at address 0x002f in the MEMORY2 window	**mem2** 0x002f

Note:

To view a different block of memory explicitly in the default MEMORY window, use the aliased command MEM0. It works in *exactly* the same way that the MEM command works. 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 4.4, *The Active Window*, on page 4-19).

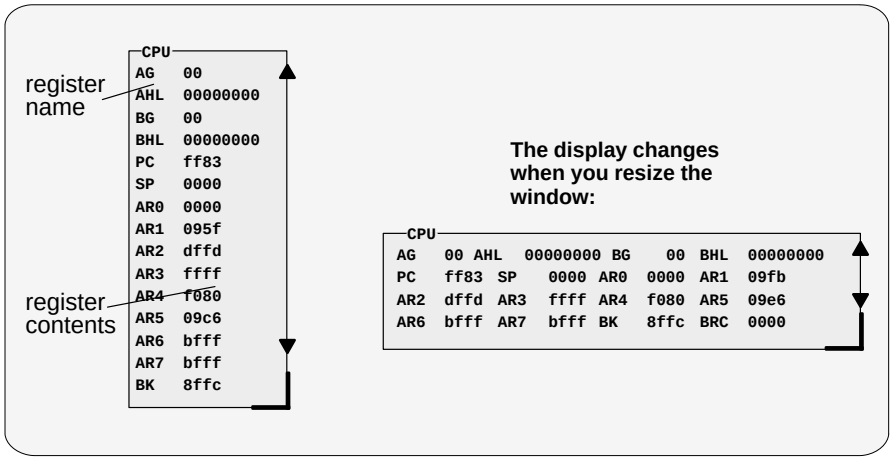
2) Press F4.

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.

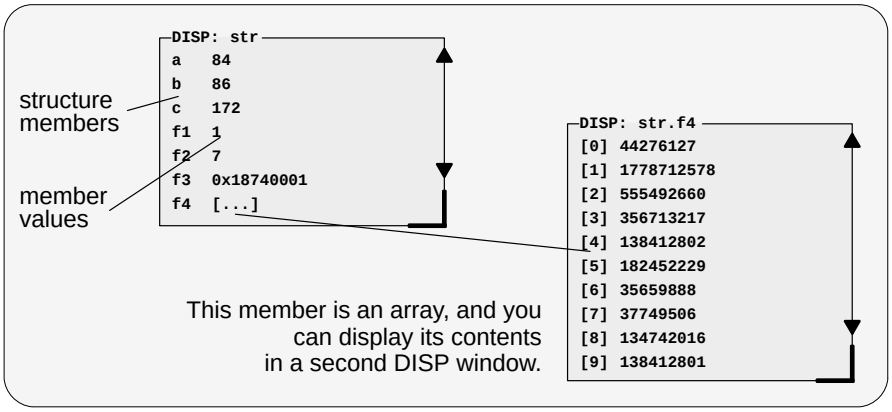
CPU window



Purpose	Shows the contents of the TMS320C5xx 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 a result of program execution. The debugger highlights 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	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:

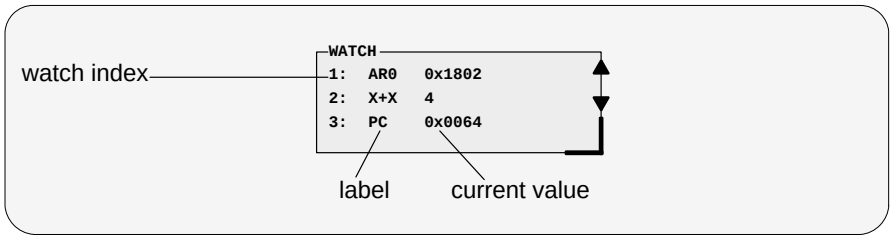
disp *expression*

Data is displayed in its natural format:

- Integer values are displayed in decimal format.
- 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 could not 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 basic syntax is:

wa *expression* [, *label*]

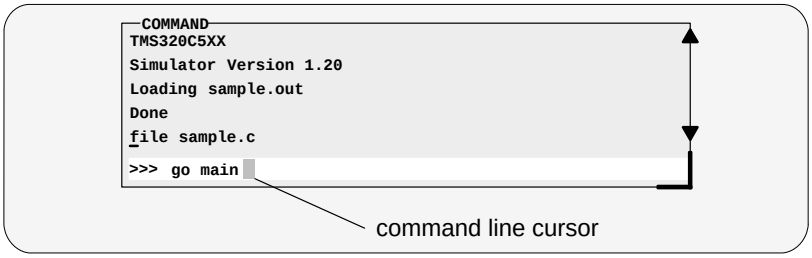
WA adds *expression* to the WATCH window. (If there's no WATCH window, then WA also opens a WATCH window).

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

4.3 Cursors

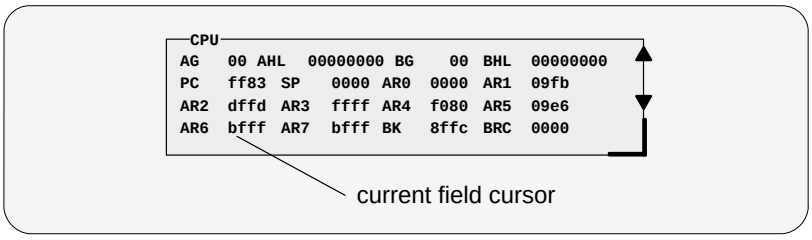
The debugger display has three types of cursors:

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



- The **mouse cursor** is a block-shaped cursor that tracks mouse movements over the entire display. This cursor is controlled by the mouse driver installed on your system; if you haven't installed a mouse, you won't see a mouse cursor on the debugger display.

- The **current-field cursor** identifies the current field in the active window. On PCs, this is the hardware cursor that is associated with your graphics card. Arrow keys *do* affect this cursor's movement.



4.4 The Active Window

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

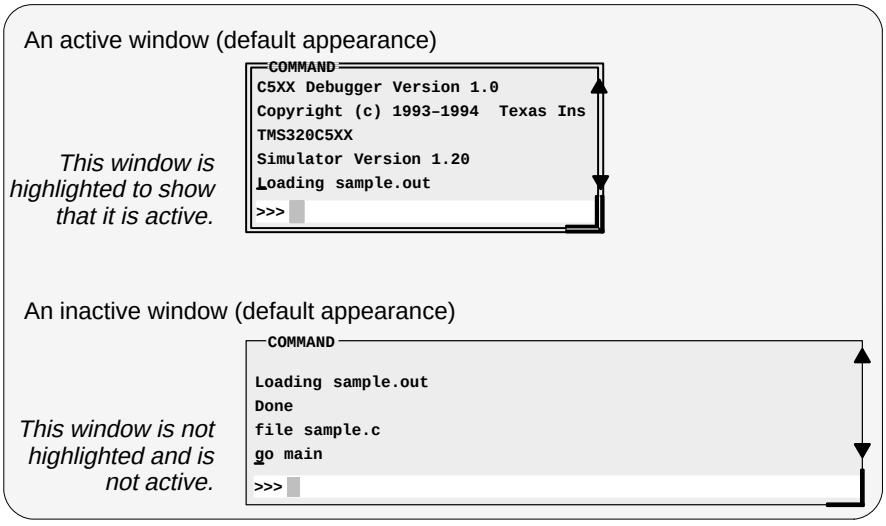
You can move, resize, or close *only one window at a time*; thus, only one window at a time can be the **active window**. Whether or not a window is active doesn't affect the debugger's ability to update information in a window—it affects only your ability to manipulate a window.

Identifying the active window

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

You can alter the active window's border style and colors if you wish; Figure 4–6 illustrates the default appearance of an active window and an inactive window.

Figure 4–6. 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.

4.5 Manipulating Windows

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

Note:

You can resize or move any window, but first the window must be *active*. For information about selecting the active window, see Section 4.4, *The Active Window*, on page 4-19.

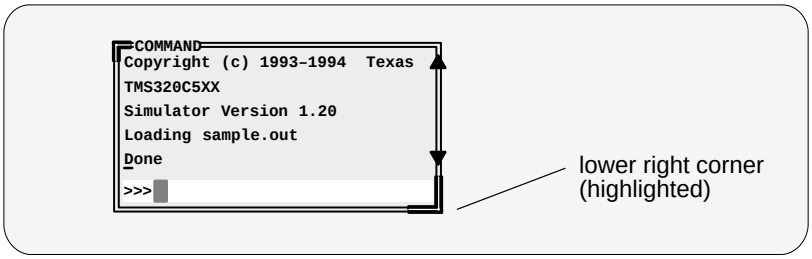
Resizing a window

The minimum window size is three lines by four characters. The maximum window size varies, depending on which screen size option 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 4-24.

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

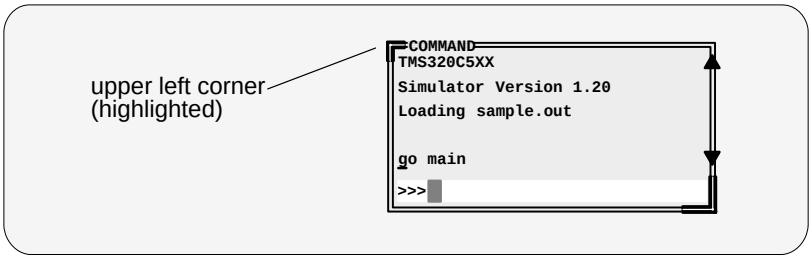
Another way to resize the active window is to zoom it. Zooming a window makes it as large as possible so that it takes up the entire display (except for the menu bar) and hides all the other windows. Unlike the SIZE command, zooming is not affected by the window's position in the display.

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

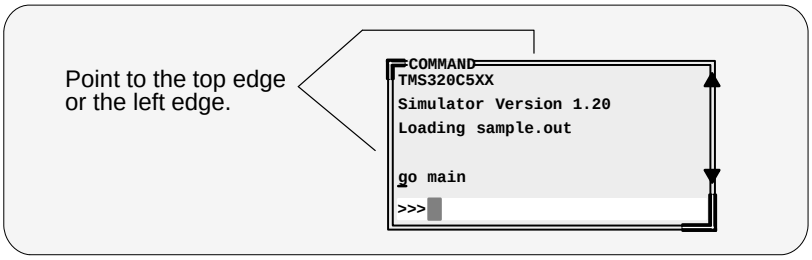
Moving a window

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 ZOOM 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 window 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
```

4.6 Manipulating a Window's Contents

Although you may be concerned with changing the way windows appear in the display—where they are and how big/small they are—you'll usually be interested in something much more important: *what's in the windows*. Some windows contain more information than a screen can display; 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 4.4, *The Active Window*, on page 4-19.

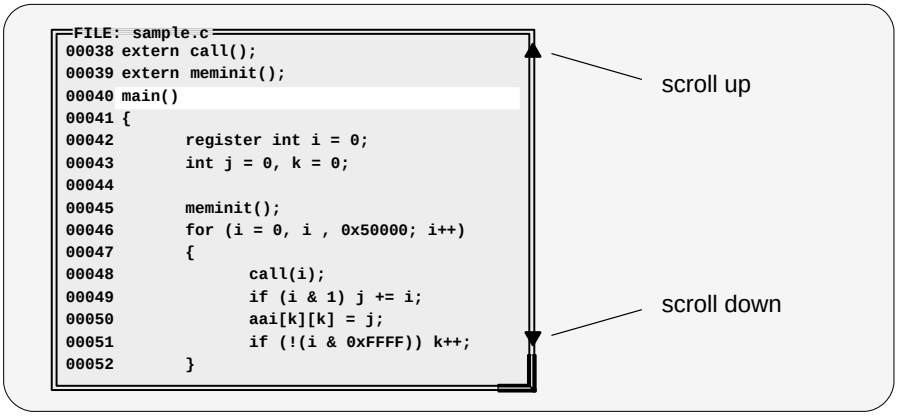
Scrolling through a window's contents

If you resize a window to make it smaller, you may hide information. Sometimes, a window may contain more information than a screen can display. 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, it moves the field cursor left one field; at the first field on a line, it 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, it moves the field cursor right one field; at the last field on a line, it 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. (These methods are described in detail in Section 8.3, *Basic Methods for Changing Data Values*, page 8-4.)

Note:

In the FILE, DISASSEMBLY, CALLS, and PROFILE 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 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.

4.7 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 windows, when you open one it will occupy the same position as the last one of that type that you closed.

Chapter 5
Entering and Using Commands

The debugger provides you with several ways of 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 the Sun version of the simulator. For descriptions of these restrictions, refer to the installation guide.

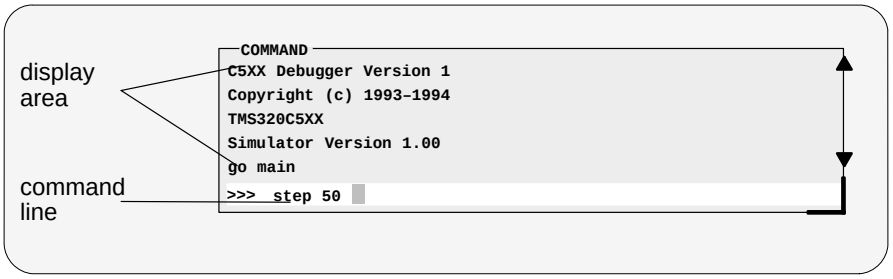
Topic **Page**

5.1 Entering Commands From the Command Line

The debugger supports a complete set of commands that help you to control and monitor program execution, customize the display, and perform other tasks. These commands are discussed in various sections throughout this book, as they apply to the current topic. Chapter 14 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 5–1 shows the COMMAND window.

Figure 5–1. The COMMAND Window



The COMMAND window serves two purposes:

- The **command line** portion of the window provides you with an area for entering commands. For example, the command line in Figure 5–1 shows that a STEP command was typed in (but not yet entered).
- The **display area** provides the debugger with an area for echoing commands, displaying command output, or displaying errors and messages for you to read. For example, the command output in Figure 5–1 shows the messages that are displayed when you first bring up the debugger and also shows that a GO MAIN command was entered.

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

How to type in and enter commands

You can type a command at almost any time; the debugger automatically places the text on the command line when you type. When you want to enter a command, just type—no matter which window is active. You don't have to worry about making the COMMAND window active or moving the field cursor to the command line. When you start to type, the debugger usually assumes that you're typing a command and puts the text on the command line (except under certain circumstances, which are explained in *Sometimes, you can't type a command* on page 5-4). 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	DEL
Move forward through text while erasing characters	SPACE
Insert text into the characters that are already on the command line	INSERT

Notes:

1) You cannot use the arrow keys to move through or edit text on the command line.

2) Typing a command doesn't make the COMMAND window the active window.

3) 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 the character as part of a command and displays it 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. This applies to DOS systems only.
- When a pulldown menu is displayed, typing a letter causes the debugger to execute a selection from the menu.
- When you're pressing the CONTROL key, pressing H or L moves the command-line cursor backward or forward through the text on the command line.
- When you're editing a field, typing enters a new value in the field.
- When you're using the MOVE or SIZE command interactively, pressing keys affects the size or position of the active window. Before you can enter any more commands, you must press ESC to terminate the interactive moving or sizing.
- When you've brought up a dialog box, typing enters a parameter value for the current field in the box. Refer to Section 5.3, *Using Dialog Boxes*, on page 5-11 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 reenter a command, you can move through this list, select a command that you've already executed, and reexecute 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.

For information about using the PDM's command history, refer to page 2-13.

Clearing the display area

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

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

cls

Recording information from the display area

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

Log files can be executed by using the TAKE command. When you use DLOG to record the information from the 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 reexecute 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 already recording to a log file, 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.

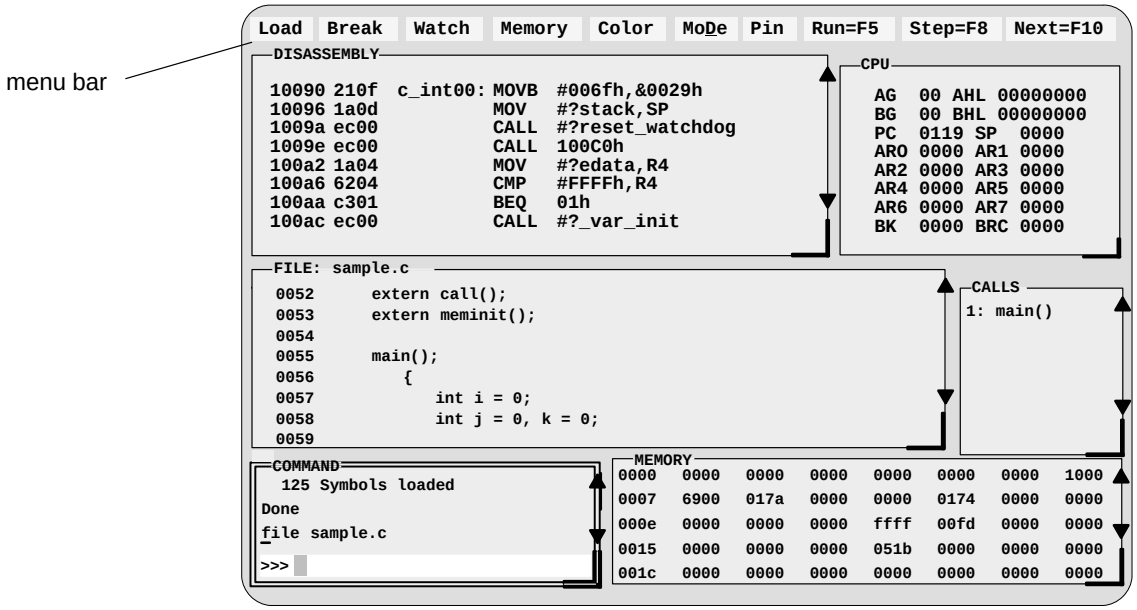
- **Writing over an existing file.** Use the **w** parameter to open an existing file if you want to write over the current contents of the file. Note that this is the default action if you specify an existing filename without using either the **a** or **w** options; you will lose the contents of an existing file if you don't use the append (a) option.

For more information about the PDM version of the DLOG command, see page 2-10.

5.2 Using the Menu Bar and the Pulldown Menus

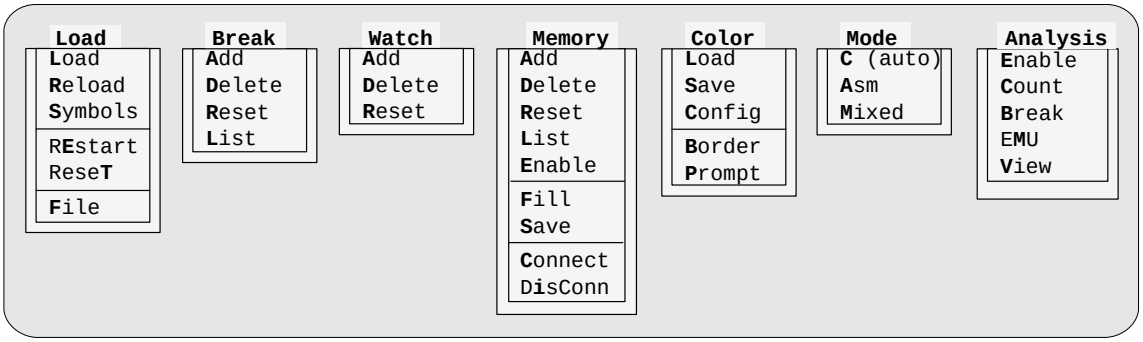
In all three debugger modes, you'll see a menu bar at the top of the screen. The menu selections offer you an alternative method for entering many of the debugger commands. Figure 5–2 points out the menu bar in a mixed-mode display. There are several ways to use the selections on the menu bar, depending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

Figure 5–2. 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 5–3.

Figure 5–3. All of the Pulldown Menus (Basic Debugger Display)



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

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 or only optional parameters, then the debugger executes the command as soon as you select it.
- If you select a command that has one or more required 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. Note that the mouse methods apply to DOS systems only.

Mouse method 1

1) Point the mouse cursor at an appropriate selection in the menu bar.
2) Press the left mouse button; don't release it.
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.

5.3 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.

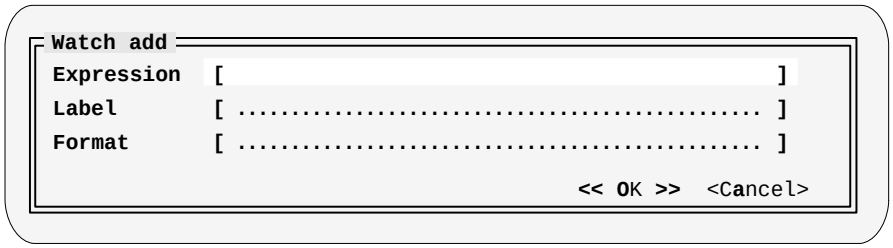
Some debugger commands have very simple dialog boxes that provide you with an alternative method for typing in values. Other commands, such as analysis commands, have more complex dialog boxes; in addition to typing in values, you may be asked to make selections from a list of predefined parameters.

Entering text in a dialog box

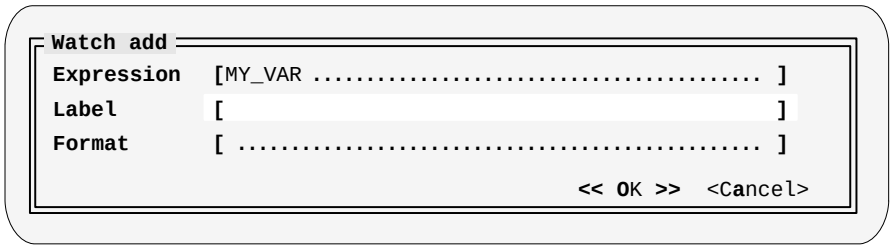
Entering text in a dialog box is much like entering commands on the command line. For example, the Add entry on the Watch menu is equivalent to entering the WA command. This command has three parameters:

wa *expression* [,[*label*] *[,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:



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



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, 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 5.1, page 5-2, 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 on <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.

Selecting parameters in a dialog box

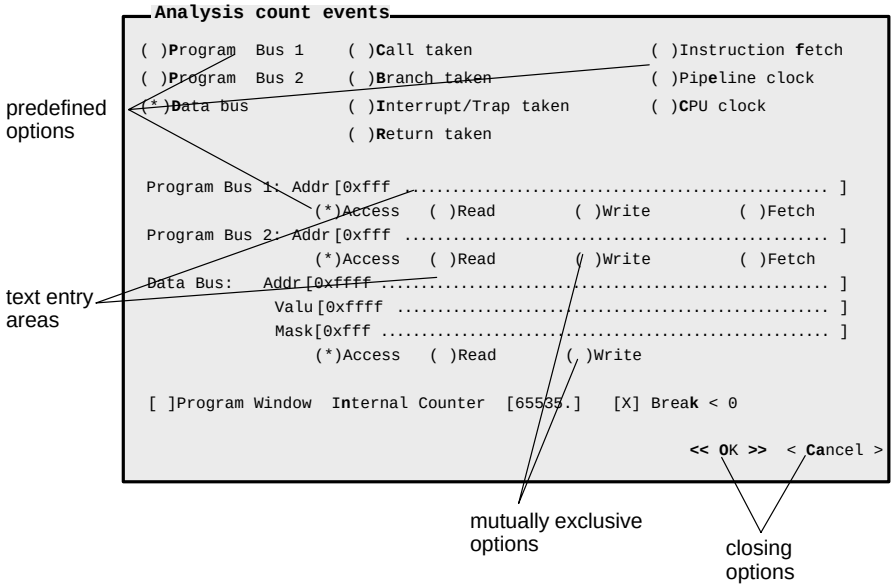
More complex dialog boxes, such as those associated with analysis commands, allow you to:

- **Enter text.** Entering text in a more complex dialog box is the same as entering text on the command line. Refer to *Entering text in a dialog box*, page 5-11, for more information.
- **Choose from a list of predefined options.** There are two types of predefined options in a dialog box. The first type of option allows you to enable one or more predefined options. The second type of option is *mutually exclusive*; therefore, you can enable only one at a time.

  Valid options (of the opened dialog box) are listed for you so that all you have to do is point and click to make your selections.
- **Close the dialog box.** The more complex dialog boxes do not close automatically. They allow you the option of saving or discarding any changes you made to your parameter choices. All you have to do to close the dialog box is point and click on the appropriate option, either OK or CANCEL.

Figure 5–4 shows you the components of a complex dialog box used with the analysis module.

Figure 5–4. The Components of a Dialog Box

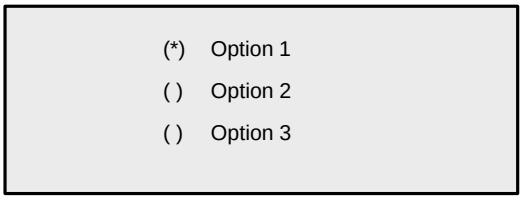


When you display a dialog box for the first time during a debugging session, nothing is enabled. When you bring up the same dialog box again, though, your previous selections will be remembered. (This is similar to having a command history.)

As Figure 5–4 shows, options are preceded by either square brackets or parentheses; mutually exclusive options are only preceded by parentheses. Enabling options preceded by square brackets is like turning a switch on and off. When the option is enabled, the debugger displays an X inside the brackets preceding the option. You can enable as many of these options as you want:

[X]	Option 1	[ ] Option 2	[X] Option 3
[ ]	Option 4	[X] Option 5	[X] Option 6
[X]	Option 7	[ ] Option 8	[ ] Option 9

Mutually exclusive options, however, are enabled when the debugger displays an asterisk inside the parentheses preceding your selection. The following example illustrates this:



Notice that only one option is enabled at a time. There are several ways to enable both types of options:

Mouse method

1) Point the cursor at the option you want to enable.

2) Click the left mouse button. This enables the event and displays an X next to the option (or an asterisk next to a mutually exclusive option).

Repeat these two steps to disable an option. When the X (or asterisk) is no longer displayed, that option has been disabled.

Keyboard method 1

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

2) Press and release the key that corresponds to the highlighted letter or number of the option that you want to enable. The debugger displays an X (or asterisk) next to the option, indicating that selection is enabled.

Repeat these two steps to disable an option. When the X (or asterisk) is no longer displayed, that option has been disabled.

Keyboard method 2

TAB 1) Press the TAB key to move through the dialog box until the cursor points to the option you want to enable.

↓ ↑ 2) Use the arrow keys to move up and down or left and right.

When you enable a mutually exclusive option, moving the arrow keys alone will place an asterisk inside the parentheses, indicating that option is enabled. However, to enable an option preceded by square brackets, you must:

SPACE Press the SPACE bar. The debugger displays an X next to your selection, thus enabling that particular option.

or

F9 Press the F9 key. The debugger displays an X next to your selection, thus enabling that particular option.

Repeat these steps to disable an option.

Closing a dialog box

The more complex dialog boxes do not close automatically; the debugger expects input from you. When you close a dialog box, you can:

- ❑ Save the changes you made, or
- ❑ Discard any of the changes you made.

Note:

The default option, <<OK>>, is highlighted; clicking on this option saves your changes and closes the dialog box.

There are several ways to close a dialog box:

↖ 1) Point the cursor at <<OK>> to close the dialog box and save your changes. Or you can opt to discard your changes by pointing the cursor at <<CANCEL>>.

2) Click the left mouse button. This executes your choice and closes the dialog box.

Keyboard method 1

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

X 2) Press and release the O key to save your changes. Press and release the A key to discard your changes. Both of these actions execute your choice and close the dialog box.

Keyboard method 2

TAB 1) Press the TAB key to move through the dialog box until your cursor is in the <<OK>> or <<CANCEL>> field.

← → 2) Use the arrow keys to switch between <<OK>> and <<CANCEL>>.

↵ 3) Press the ↵ key to accept your selection. This executes your choice and closes the dialog box.

5.4 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 to enable 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 ten deep. To halt the debugger's execution of a batch file, press ESC.

The format for the TAKE command is:

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

- The *batch filename* parameter identifies the file that contains commands.
  - If you supply path information with the *filename*, the debugger looks for the file 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 or OS/2 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 (for DOS) or your config.sys file (for OS/2). On DOS systems, you can also set D_DIR from within the debugger by using the SYSTEM command (see Section 5.6, *Entering Operating-System Commands*, page 5-24).
- 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.

For information about the PDM version of the TAKE command, see page 2-9.

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 display area of the COMMAND window.

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.

For information about the PDM version of the ECHO command, see page 2-12.

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

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

Constant	Debugger Tool
$$EMU$$	emulator
$$EVM$$	evaluation module (EVM)
$$SIM$$	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 15 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 \c5xxhll
take emuinit.cmd
.
.
.
endif

if $$EVM$$
echo Invoking initialization batch file for EVM.
use \c5xxhll
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 way that they evaluate in the run conditional command expression. (See Chapter 15 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. A Boolean 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.

See *Controlling PDM command execution*, page 2-10, for more information about the PDM versions of the IF and LOOP commands.

5.5 Defining Your Own Command Strings

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

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

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

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

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

  For example, suppose you always began a debugging session by loading the same object file, 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, %4;mem %1"`

  Then you could enter:

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

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

- **Listing all aliases.** To display a list of all 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,%4;mem %1
```

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

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

  `alias init` ↵

  Then you'd see:

```
"INIT" aliased as "load test.out; 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, reenter 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.

Notes:

1) Alias definitions are lost when you exit the debugger. If you want to re-use aliases, define them in a batch file.
2) 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.

For information about the PDM versions of the ALIAS and UNALIAS commands, see page 2-15.

5.6 Entering Operating-System Commands (DOS Only)

The debugger provides a simple method of 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, 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 preceding example, 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 displays until you press ↵.

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.

For information about the PDM version of the SYSTEM command, see page 2-16.

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 6

Defining a Memory Map

Before you begin a debugging session, you must supply the debugger with a memory map. The memory map tells the debugger which areas of memory it can and can't access. Note that you can use the Memory pulldown menu to enter the commands described in this chapter.

Topic **Page**

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

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

Note:

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. However, 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 of 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 the 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. The batch filename differs for each version of the debugger:

   - For the emulator, this file is called *emuinit.cmd*.
   - For the EVM, this file is called *evminit.cmd*.
   - For the simulator, this file is called *siminit.cmd*.

   If the debugger finds the file corresponding to your tool, it reads and executes the file.

3) If the debugger does not find the –t option or the initialization batch file, it looks for a file called *init.cmd*. This search mechanism allows you to have one initialization batch file for more than one debugger tool. To set up this file, you can use the IF/ELSE/ENDIF commands (for more information, see *Controlling command execution in a batch file*, on page 5-18) to indicate which memory map applies to each tool.

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. For specific error messages, see Appendix D, *Debugger and PDM Messages.*
- **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 6-7). Alternatively, you can turn memory mapping off during a load by using the MAP OFF command (see page 6-10).
- **Accessing conflict and extra cycles (simulator only).** If two memory read access requests come simultaneously during an execution, you may be unable to complete both requests within the same clock cycle. If both locations belong to the same physical memory block and the block is single-access memory, both requests cannot be processed within the same clock cycle.

6.2 Customizing the Memory Map

The customizable ’C5xx (cDSP) debugger allows you maximum flexibility in configuring a memory map. Because the size and address of the memory map is not fixed in the debugger, you can select any amount of ROM or RAM internally, externally, or both.

The following example shows how you can have both RAM and ROM mapped to the same address:

```
ma 0xc000, 0, 0x1000, R          ;Internal Program ROM

ma 0xc000, 0, 0x1000, R|W|EX     ;External Program ROM
```

During execution or when the debugger performs memory accesses, the block of memory accessed is based on the ’C5xx MP/$\overline{MC}$ bit located in the PMST register. When this bit is set to 0, the on-chip program ROM is enabled. When it is set to 1, the off-chip program RAM is enabled.

The next example shows you two blocks of RAM, one internal and one external, mapped to the same address.

```
ma 0x0080, 0, 0x0380, R|W        ;Internal Program RAM

ma 0x0080, 0, 0x0380, R|W|EX     ;External Program RAM
```

For the above example, the block of memory is accessed based on the OVLY bit located in the PMST register during execution or when the debugger performs memory accesses. When this bit is set to 1, the on-chip dual-access data RAM is mapped to internal program space. When it is set to 0, the off-chip program RAM is enabled.

The debugger accesses the three types of memory (data, program ROM, and program RAM) according to the type of memory and the values of the MP/$\overline{MC}$ bits. The following table summarizes how the debugger accesses memory:

Type of Memory	Memory Access
Data	Accesses internal memory block, then external memory block.
Program ROM	If MP/$\overline{MC}$ is set to 0, accesses internal memory block, then external memory block; if MP/$\overline{MC}$ is set to 1, accesses external memory block.
Program RAM	If OVLY is set to 1, accesses internal memory block, then external memory block; if OVLY is set to 0, accesses external memory block.

Programming your memory

The easiest time to set up your memory is during the initialization process. However, you can edit your memory map while your program is running.

Use the OVLY and MP/$\overline{MC}$ bits of the status/PMST registers to set the amount of external and internal program memory you need. The values for the OVLY and MP/$\overline{MC}$ bits are as follows:

- OVLY bit

  0 = internal program memory

  1 = external program memory

- MP/$\overline{MC}$ bit

  0 = internal program memory (ROM)

  1 = external program memory

You can edit the the values of the OVLY and MP/$\overline{MC}$ bits by using the debugger or by programming the PMST register. To edit the values of these bits, scroll down the CPU window until you see the PMST register. The CPU window is editable; you can enter the values for each bit.

6.3 A Sample Memory Map

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. Example 6–1 shows the memory map commands that are defined in the initialization batch file that accompanies the simulator. If you are using the simulator, you can use the file as is, edit it, or create your own memory map batch file. The files shipped with the emulator are similar to that of the simulator.

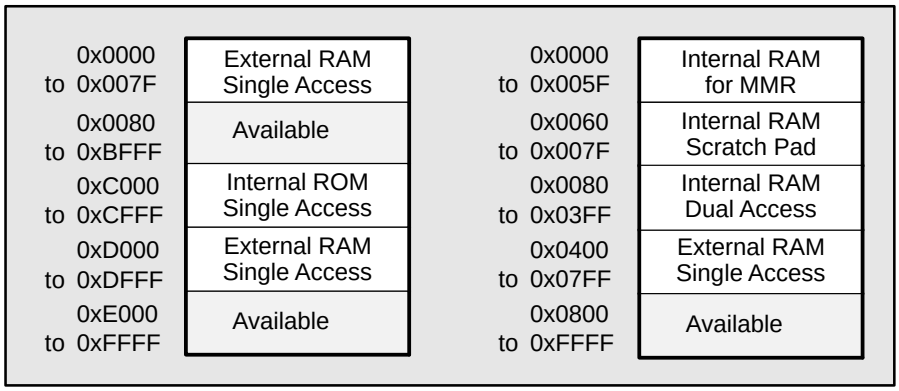
Example 6–1. Sample Initialization Batch File for Use with the TMS320C5xx

```
ma 0x0000,0, 0x80,   EX|RAM
ma 0xc000,0, 0x1000, ROM
ma 0xd000,0, 0x1000, EX|RAM

ma 0x0000,1, 0x0060, RAM
ma 0x0060,1, 0x0020, RAM
ma 0x0080,1, 0x0380, RAM
ma 0x0400,1, 0x0400, EX|RAM
```

The MA commands (shown in Example 6–1) define valid memory ranges and identify the read/write characteristics of the memory ranges. The MAP command enables mapping (see Section 6.5, *Enabling Memory Mapping*, on page 6-10). By default, mapping is enabled when you invoke the debugger. Figure 6–1 illustrates the memory map defined in Example 6–1.

Figure 6–1. Sample Memory Map for Use With the TMS320C5xx Simulator



6.4 Identifying Usable Memory Ranges

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

ma *address, page, length, type*

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

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

  ```
  Conflicting map range
  ```

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

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

- The *length* parameter defines the length of the range. This parameter can be any C expression.

- The *type* parameter identifies the read/write characteristics of the memory range. The *type* must be one of these keywords:

To identify this kind of memory,	Use this keyword as the *type* parameter
Read-only memory	**R** or **ROM**
Write-only memory	**W** or **WOM**
Read/write memory	**R|W** or **RAM**
Read/write external memory	**RAM|EX** or **R|W|EX**
Read-only peripheral frame	**P|R**
Read/write peripheral frame	**P|R|W**

Notes:

1) The debugger caches memory that is not defined as a port type (P|R, P|W, or P|R|W). For ranges that you don't want cached, be sure to map them as ports.

2) When you are using the simulator, you can use the parameter values P|R, P|W, and P|R|W to simulate I/O ports. See Section 6.9, *Simulating I/O Space.*

3) Be sure that the map ranges that you specify in a COFF file match those that you define with the MA command. A command sequence such as:

   ```
   ma x,0,y,ram; ma x+y,0,z,ram
   ```

   doesn't equal

   ```
   ma x,0,y+z,ram
   ```

   If you plan to load a COFF block that spans the length of y + z, you should use the second MA command example. Alternatively, you can turn memory mapping off during a load by using the MAP OFF command (see Section 6.5, page 6-10).

4) Although the address range for both of the following MA commands is the same (0x0400 to 0x0800), one range is internal and the other range is external.

   ```
   ma 0x0400, 0, 0x0800,  ROM
   ma 0x0400, 0, 0x0800,  EX|ROM
   ```

   When the simulator is operating in microcomputer mode, the internal program ROM is accessed. Otherwise, if the simulator is running in microprocessor mode, the external program memory module is used.

Memory mapping with the simulator

Unlike the emulator and EVM, the 'C5xx 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 'C5xx, do not allocate more than 20K words of memory in your memory map. For example, the following memory map allocates 64K words of 'C5xx program memory.

Example 6–2. Sample Memory Map for the TMS320C5xx Using Memory Cache Capabilities

```
MA 0,0,0x5000,R|W
MA 0x5000,0,0x5000,R|W
MA 0xa000,0,0x5000,R|W
MA 0xf000,0,0x1000,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 now 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 (40K bytes) of memory in your memory map, then your disk should have at least 40K bytes of free space available. To do this, you can enter:

```
ma 0x0, 0, 0x5000, ram ↵
```

Note:

You can also use the memory cache capability feature for the data memory.

6.5 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 as protected

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

```
Error in expression
```

6.6 Checking the Memory Map

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

ml

The ML command lists the page, starting address, ending address, and read/write characteristics of each defined memory range.

For example, assume you issue the following MA commands:

```
ma 0,0, 0x3000, ROM ↵
ma 0x4000, 0, 0x2000, EX|RAM ↵
ma 0, 1, 0x4000, RAM ↵
ma 0x8000, 1, 0x2000, EX|RAM ↵
ma 0x6, 2, 0x3, P|R ↵
```

If you type ML on the command window, the debugger will display the following on the command window display area:

```
Page        Memory range        Attributes
  0         0000 – 2fff         R
  0         4000 – 5fff         R|W|EX
  1         0000 – 3fff         R|W
  1         8000 – 9fff         R|W|EX
  2         0006 – 0008         P|R
```

starting address

ending address

page 0 = program memory
page 1 = data memory
page 2 = I/O space

6.7 Modifying the Memory Map During a Debugging Session

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

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

md *address, page*

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

  ```
  Specified map not found
  ```

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

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

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 (see *Disconnecting an I/O port*, page 6-18).

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

mr

This resets the debugger memory map.

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

ma *address, page, length, type*

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

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.

6.8 Using Multiple Memory Maps for Multiple Target Systems (Emulator Only)

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 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 the initialization batch file during invocation. 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 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.

6.9 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 P|R (input port), P|W (output port), or P|R|W (input/output port) as the memory type. Use page 2 to simulate I/O space. 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. Use page 1 for file connects to data memory.

Connecting an I/O port

mc The MC (memory connect) command connects P|R, P|W, or P|R|W to an input or output file. MC also allows you to connect any data memory location (except 0–1f) to an input or output file to read data from or write data into the file. The syntax for this command is:

mc *portaddress, page, length, filename, fileaccess*

- The *portaddress* parameter defines the address of the I/O space or data memory. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.

  The *portaddress* must be previously defined with the MA command (described on page 6-7) and have a keyword of either P|R (input port) or P|R|W (input/output port). The length of the address range defined for the port (or peripheral frame) can be 0x1000 to 0x1FFF bytes and does not have to be a multiple of 16.

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

To identify this page,	Use this value as the *page* parameter
Data memory	**1**
I/O space	**2**

- The *length* parameter defines the length of the range. This parameter can be any C expression.

- The *filename* parameter can be any filename. If you connect a port or memory location to read from a file, the file must exist, or the MC command will fail.

- The *fileaccess* parameter identifies the access characteristics of the I/O memory and data memory. The file access must be one of the keywords identified below:

To identify this file access type	Use this keyword as the *fileaccess* parameter
Input port (I/O space)	**P|R**
Simulator halt at EOF of input space (I/O space)	**R|P|NR**
Output port (I/O space)	**P|W**
Read-only internal memory	**R**
Read-only external memory	**EX|R**
Simulator halt at EOF of input file for internal memory	**R|NR**
Simulator halt at EOF of input file for external memory	**EX|R|NR**
Write-only internal memory	**W**
Write-only external memory	**EX|W**

  For I/O memory locations, the file is accessed during a read or write instruction to the associated port address. You can connect any I/O port to a file. A maximum of one input and one output file can be connected to a single port; however, multiple ports can be connected to a single file.

  For data memory locations, the debugger accesses the data as follows:

  - When you're executing code:
    - If you have specified a file, the debugger reads the data from the file and updates the memory location with that data.
    - If you have specified a file, the debugger writes the data to the memory location, as well as to the file.
  - When you're using the debugger:
    - The debugger reads the data value from the memory location, *not* from the connected file.
    - If you have specified a file, the debugger writes the data to the memory location, as well as to the file.

If you use the NR parameter, then the simulator halts execution when it reads an EOF. The debugger displays the appropriate message in the COMMAND window display area:

```
<addr> EOF reached – connected at port(I/O_PAGE)
```

or

```
<addr> EOF reached – connected at location (DATA_PAGE)
```

At this point, you can disconnect the file by using the MI command and attach a new file by using the MC command. If you don't do anything, then the input file rewinds automatically, and execution continues until EOF is read.

If you do not specify NR at EOF, execution does not halt, and you are not notified when EOF is reached. The input file rewinds automatically, and the simulator resumes reading from the file.

Example 6–3 shows how input and output ports can be connected to specific memory blocks.

Example 6–3. Connecting Input and Output Ports to Input or Output Files.

Assume that you have two data memory blocks:

```
ma 0x100,1, 0x10, EX|RAM   ;block1
ma 0x200,1, 0x10, RAM   ;block2
```

- You could use the MC command to set up and connect an input file to block1:

```
mc 0x100, 1, 0x1, my_input.dat, EX|R
```

- You could use the MC command to set up and connect an output file to block2:

```
mc 0x205, 1, 0x1, my_output.dat, W
```

- You could use the MC command to halt simulator at EOF of input file:

```
mc 0x100, 1, 0x1, my_input.dat, EX|R|NR
```

or

```
mc 0x100, 1, 0x1, my_input.dat, R|NR
```

Example 6–4 shows how an input port can be connected to an input file named in.dat.

Example 6–4. Connecting an Input Port to an Input File

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

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

Use MA and MC commands to set up and connect an input port:

```
MA     0x50,2,0x1,R|P          Configure port address 50h
                                 as an input port.
MC     0x50,2,0x1,in.dat,R     Open file in.dat and
                                 connect it to port address 50.
```

Assume that the following instruction is part of your program; it reads from the file in.dat:

```
       PORTR  050,data_mem     Read file in.dat, and put the
                                 value into the DATA_MEM location.
```

Notes:

1) You can only connect a file to configured location(s).
2) You cannot connect a file to program memory (page 0) locations.
3) You cannot connect a file to core MMR area (0x0000 to 0x001F) of data memory (page 1).
4) While connecting a file to a set of locations:
   - Locations must not spread across memory block boundaries.
   - Two read-only files must not overlap.
   - Two write-only files must not overlap.

Disconnecting an I/O port

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

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

mi *port address, page,* {**R**|**W**|**EX**}

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

6.10 Simulating External Interrupts (Simulator Only)

The ’C5xx simulator allows you to simulate the external interrupt signals $\overline{\text{INT}}$0 to $\overline{\text{INT}}$15 and to select the clock cycle where you want an interrupt to occur. To do this, you create a data file and connect it to one of the 16 interrupt pins, $\overline{\text{INT}}$0 to $\overline{\text{INT}}$15, or the $\overline{\text{BIO}}$ pin.

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, logic value*] **rpt** {*n* | **EOS**}

Note that the square brackets are used only with logic values for the $\overline{\text{BIO}}$ pin.

- The *clock cycle* parameter represents the CPU clock cycle in which 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 in which 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 performed on the clock cycle value; the interrupt occurs exactly as the clock cycle value is written.

  - **Relative**. You can also select a clock cycle that is relative to the time at which the last event occurred. 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 *logic value* parameter is only for the $\overline{\text{BIO}}$ pin. You must use a value to force the signal to go high or low at the corresponding clock cycle. A value of 1 forces the signal to go high, and a value of 0 forces the signal to go low. For example:

  ```
  [12,1] [23,0] [45,1]
  ```

  This causes the $\overline{\text{BIO}}$ pin to go high at the 12th cycle, low at the 23rd cycle, and high again at the 45th cycle.

- 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 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, list, and disconnect the interrupt pin to 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 pin, use the following command:

pinc *pinname, filename*

- The *pinname* identifies the pin and must be one of the 16 simulated pins ($\overline{\text{INT}}$0–$\overline{\text{INT}}$15) or the $\overline{\text{BIO}}$ pin.
- The *filename* is the name of your input file.

Example 6–5 shows you how to connect your input file using the PINC command.

Example 6–5. Connecting the Input File With the PINC Command

Suppose you want to generate an $\overline{\text{INT}}$2 external interrupt at the 12th, 34th, 56th, and 89th clock cycles.

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

```
12  34 56 89
```

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

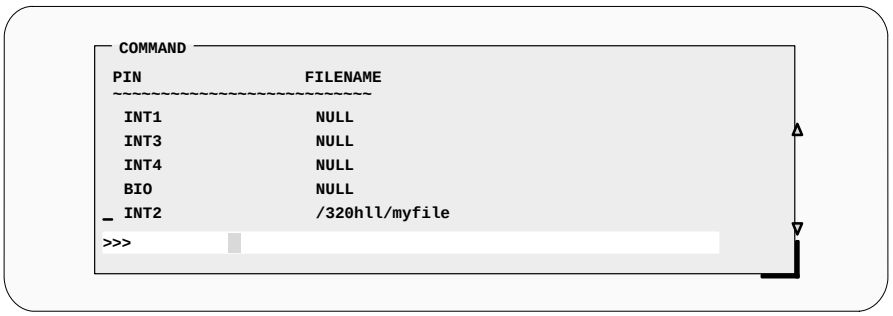
```
pinc  myfile, int2                    Connects your data file
                                      to the specific interrupt pin
```

This command connects myfile to the $\overline{\text{INT}}$2 pin. As a result, the simulator generates 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 pin that has been connected, it displays the name of the pin and the absolute pathname of the file in the COMMAND window.



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

pind *pinname*

The *pinname* parameter identifies the interrupt pin and must be one of the external interrupt pins ($\overline{\text{INT}}$0–$\overline{\text{INT}}$15) or the $\overline{\text{BIO}}$ pin. The PIND command detaches the file from the input pin. After executing this command, you can connect another file to the same pin.

Chapter 7

Loading, Displaying, and Running Code

The main purpose of a debugging system is to allow you to load and run your programs in a test environment. This chapter tells you how to load your programs into the debugging environment, run them on the target system, and view the associated source code. Many of the commands described in this chapter can also be executed from the Load pulldown menu.

Topic **Page**

7.1 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 4.1, *Debugging Modes and Default Displays*, on page 4-2.

Table 7–1. Debugging Modes and Display Windows

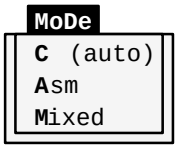
Use this mode	To view	The debugger uses these code-display windows
assembly mode	*assembly language code only* (even if your program is executing C code)	❑ DISASSEMBLY
auto mode	*assembly language code* (when that's what your program is running)	❑ DISASSEMBLY
auto mode	*C code only* (when that's what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



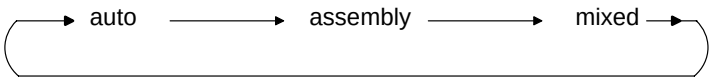
1) Point to the menu name.

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 5.2, *Using the Pulldown Menus*, on page 5-7.

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



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

c Changes from the current mode to auto mode.

asm Changes from the current mode to assembly mode.

mix Changes from the current mode to mixed mode.

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

7.2 Displaying Your Source Programs (or Other Text Files)

The debugger displays two types of code:

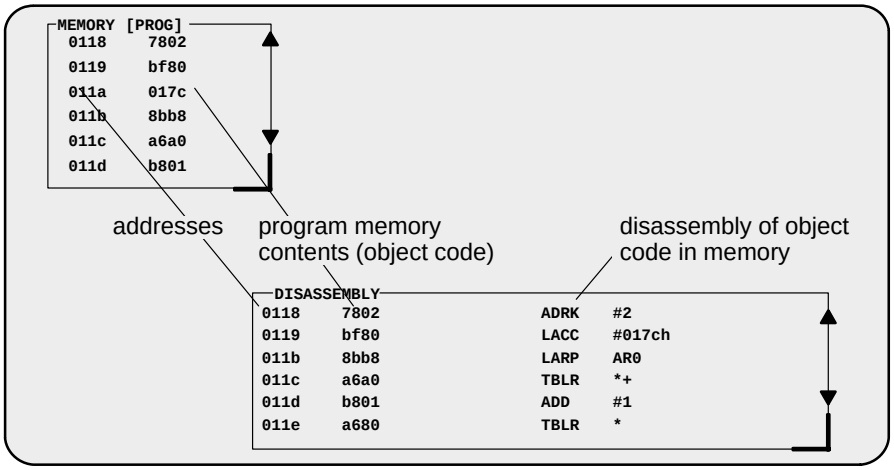
- It displays **assembly language code** in the DISASSEMBLY window in auto, assembly, or mixed mode.
- It displays **C code** in the FILE window in auto and mixed modes.

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 program memory contents. (This code doesn't come from any of your text files or from the intermediate assembly files produced by the compiler.)



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

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

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

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

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

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

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

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

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.

Note that you can't use the patch assembly feature if you're running the debugger under DOS.

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 an NOP, or unpredictable results may occur when you are running code.
- Substituting a larger instruction for a smaller one will partially or entirely overwrite the following instruction; you will lose the instruction and may be left with another fragment.

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

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

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

Additional information about modifying assembly language code

When you use patch assembly to modify code in the disassembly window, follow these guidelines:

- **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 *TMS320C5xx 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 *TMS320C5xx 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 opens 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 7.3 on page 7-10.)

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 and FILE work similarly, but when you use FUNC, 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 the C function.

2) Click the left mouse button.

(If the CALLS window is active, you can also use the arrow keys and F9 to display the function; see the *CALLS window* discussion on page 4-9 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. You can also view your original assembly language source files in the FILE window.

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

7.3 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 1.6, *Preparing Your Program for Debugging*, on page 1-13.)

Loading code while invoking the debugger

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

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

7.4 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

  - 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 or Windows 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.

7.5 Running Your Programs

To debug your programs, you must execute them on one of three 'C5xx debugging tools (emulator, EVM, and simulator). The debugger provides two basic types of commands to help you run your code:

- **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 ending point.
  - Press ESC.
  - Press the left mouse button.

- **Single-step** commands execute assembly language or C code, one statement at a time, and update the display after each execution.

Defining the starting point for program execution

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

- Finding its entry in the CPU window

  or

dasm

- Finding the appropriately highlighted line in the FILE or DISASSEMBLY window. To do this, execute the DASM or ADDR command. The format for these commands is:

  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 the ? or EVAL command. The format for these commands is:

?PC=*new value*
or **eval pc =** *new value*

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

Running code

The debugger supports several run commands.

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

run [*expression*]

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

- ❑ If you don't supply an *expression*, the program executes until it encounters a breakpoint or until you press ESC or the left mouse button.
- ❑ If you supply a logical or relational *expression*, this becomes a conditional run (see page 7-17).
- ❑ 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 7.7, *Benchmarking*, on page 7-19.

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.

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

- If you don't supply an *expression*, the program executes a single statement, then halts.
- If you supply a logical or relational *expression*, this becomes a conditional single-step execution (see page 7-17).
- 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 a target

EVM & emulator

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

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

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

halt

When you invoke the debugger, it automatically executes a HALT command. Thus, if you enter a RUNF, quit the debugger, and later reinvoke the debugger, you will effectively reconnect the 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 'C5xx 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.

7.6 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.

7.7 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*. This process is referred to as *benchmarking*.

Benchmarking code is a multiple-step process:

Step 1: Set the PC value at the statement that marks the beginning of the section of code that you'd like to benchmark. (You can do this either by editing the PC value at the command line or by setting a software breakpoint at the statement 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: 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.

Notes:

1) The value in CLK is valid only after using a RUNB command that is terminated by a software breakpoint. (The maximum value for CLK is 65535.)
2) When programming in C, do not use a variable named CLK.

Chapter 8

Managing Data

The debugger allows you to examine and modify many types of data related to the ’C5xx and to your program. You can display and modify the values of:

- Individual memory locations or a range of memory
- ’C5xx registers
- Variables, including scalar types (ints, chars, etc.) and aggregate types (arrays, structures, etc.)

This chapter tells you how to display and change data.

Topic **Page**

8.1 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 data memory, program memory, or I/O space.
register values	The **CPU window** displays the contents of 'C5xx CPU 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	The **WATCH window** displays selected data.

These windows are referred to as **data-display windows**.

8.2 Basic Commands for Managing Data

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

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

whatis *symbol*

This command 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 ? 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].a 43 giant[0].b –79 giant[0].c 19 etc.
? j	41
? j=0x5a	90
? i	–1
? i,x	0xff

When its *expression* parameter does not identify an aggregate type, the ? command behaves like the DISP command (which is described in detail on page 8-14).

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).

For information about the PDM version of the EVAL command, refer to Section 2.9, page 2-22.

8.3 Basic Methods for Changing Data Values

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

Editing data displayed in a window

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

- ❑ Registers displayed in the CPU window
- ❑ Memory contents displayed in a MEMORY window
- ❑ Elements displayed in a DISP window
- ❑ Values displayed in the WATCH window

There are two similar methods for overwriting displayed data:

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

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

3) Type the new information. If you make a mistake or change your mind, press ESC or move the mouse outside the field and press/release the left button; this resets the field to its original value.

4) When you finish typing the new information, press ↵ or any arrow key. This replaces the original value with the new value.

1) Select the window that contains the field that you'd like to modify; make this the active window. (Use the mouse, the WIN command, or F6. For more detail, see Section 4.4, *The Active Window*, on page 4-19.)

2) Use arrow keys to move the cursor to the field you'd like to edit.
   - ↑ Moves up one field at a time.
   - ↓ Moves down one field at a time.
   - ← Moves left one field at a time.
   - → Moves right one 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, the value of some variable in the expression changes as the result of evaluating the expression.

This means that you can coerce many commands into changing values for you. Specifically, it's most helpful to use ? and EVAL to change data as well as display it. For example, if you want to see what's in auxiliary register AR3, you can enter:

```
? AR3 ↵
```

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

`? AR3++`	*Side effect: increments the contents of AR3 by 1*
`eval --AR3`	*Side effect: decrements the contents of AR3 by 1*
`? AR3 = 8`	*Side effect: sets AR3 to 8*
`eval AR3/=2`	*Side effect: divides the contents of AR3 by 2*

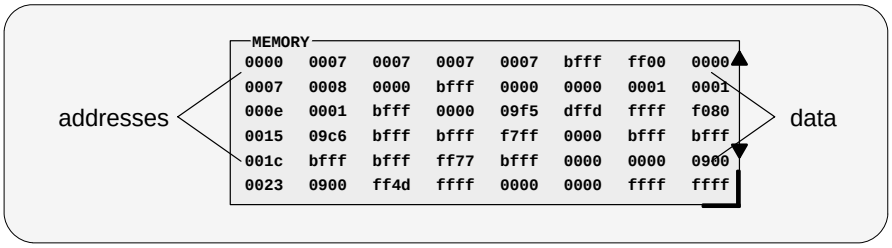
Note that not all expressions have side effects. For example, if you enter **`? AR3+4`**, the debugger displays the result of adding 4 to the contents of AR3 but does not modify AR3's contents. Expressions that have side effects must contain an assignment operator or an operator that implies an assignment. Operators that can cause a side effect are:

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

These operators are described in Chapter 15, *Basic Information About C Expressions*.

8.4 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 4-12.



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 8.3, page 8-4, 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. The basic syntax for this command is:

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 in *exactly* the same way that the MEM command works. To use this command, enter:

mem0 *address*

For more information, see the *MEMORY windows* discussion, page 4-12.

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 4-22, 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 data memory beginning from the very first address. You might enter this command:

  `mem 0x00`

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

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

  ```
  mem1 &SYM
  ```

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

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

  ```
  mem SP – AR0+ label
  ```

You can also change the display of any data-display window—including the MEMORY window—by scrolling through the window's contents. See the *Scrolling through a window's contents* discussion, page 4-27, 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 data memory location 26 (hex), you could enter:

  ```
  ? *0x26
  ```

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

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

  ```
  wa *0x26
  ```

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

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

Displaying program memory and I/O space

The ’C5xx has separate data, program, and I/O spaces. By default, the MEMORY window shows data memory. If you want to display program memory, you can enter the MEM command like this:

mem[#] *address@***prog**

The @prog suffix identifies the address as a program memory address.

Any of the examples presented in this section could be modified to display program memory or I/O space, for example:

```
mem 0x00@io
mem &SYM@prog
mem (SP – AR0 + label)@prog
? *0x26@io
wa *0x26@prog
disp *(float *)0x26@io
```

If you display program memory or I/O space in the MEMORY window, the debugger changes the window’s label to MEMORY [PROG] or MEMORY [IO] so that there is no confusion about what type of memory is displayed at any given time.

To return to data memory, enter the MEM command with an @data suffix.

Saving memory values to a file

ms Sometimes it's useful to save a block of memory values to a file. You can use the MS (memory save) command to do this; the files are saved in COFF format. (For more information about COFF, refer to the *TMS320C5xx Assembly Language Tools User's Guide*.) The syntax for the MS command is:

ms *address, page, length, filename*

- The *address* parameter identifies the first address in the block.
- The *page* parameter is a one-digit number that identifies the type of memory (program or data) to save:

To save this type of memory	Use this value as the *page* parameter
Program memory	**0**
Data memory	**1**
I/O space	**2**

- The *length* parameter defines the length, in words, of the block. This parameter can be any C expression.
- The *filename* parameter 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 0x0,1,0x10,memsave
```

To reload memory values that were saved in a file, use the LOAD command. For example, to reload the values that were stored in memsave, enter:

```
load memsave.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, page, length, data*

- The *address* parameter identifies the first address in the block.
- The *page* parameter is a one-digit number that identifies the type of memory (program or data) to fill:

To fill this type of memory	Use this value as the *page* parameter
Program memory	**0**
Data memory	**1**
I/O space	**2**

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

For example, to fill program memory locations 0x10ff–0x110d with the value 0xabcd, enter:

```
fill 0x10ff,0,0xf,0xabcd
```

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

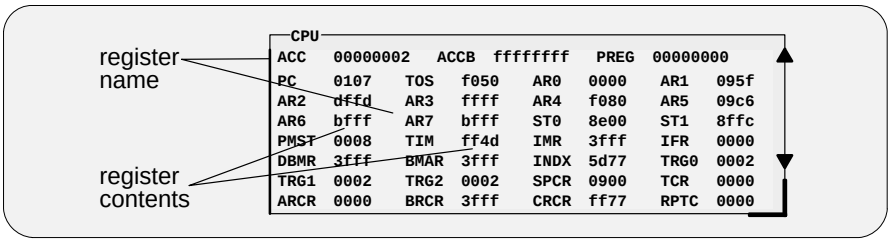
```
mem 0x10ff@prog
```

This changes the MEMORY window display to show the block of memory beginning at program memory address 0x10ff.

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

8.5 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 4-15.



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

Displaying register contents

The main way to observe register contents is to view the display in the CPU window. However, you may not be interested in all of the registers; if you're interested in only a few registers, you might want to make the CPU window small and use the extra screen space for the DISASSEMBLY or 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 AR0, enter:

  ```
  ? AR0
  ```

  The debugger displays AR0's current contents in the COMMAND window display area.

- If you want to observe a register over a longer period of time, you can use the WA command to display the register in a WATCH window. For example, if you want to observe the status register, you could enter:

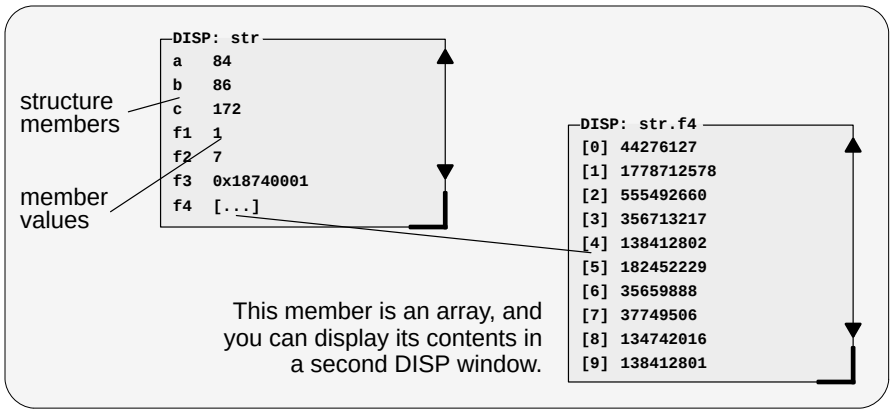
  ```
  wa ST0,Status Register 0
  ```

  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.

8.6 Managing Data in a DISP 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 4-16.



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

Displaying data in a DISP window

disp To open a DISP window, use the DISP command. Its 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	0x0000

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 can 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 4.4, *The Active Window*, on page 4-19).

Step 2: Press F4.

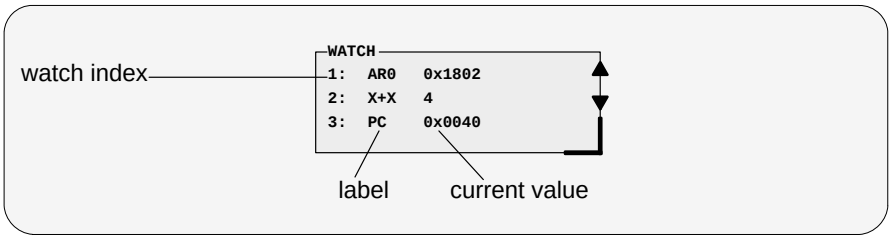
Note that you can close a window and all of its children by closing the original window.

Note:

The debugger automatically closes any DISP windows when you execute a LOAD or SLOAD command.

8.7 Managing Data in the 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.

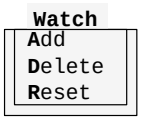


The debugger displays a WATCH window *only when you specifically request a WATCH window* with the WA command (described below). Note that there is only one WATCH window. For additional details concerning the WATCH window, see the *WATCH window* discussion, page 4-17.

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 8.3, page 8-4, 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 pulldown menus, refer to Section 5.2, *Using the Menu Bar and the Pulldown Menus*, page 5-7.



Displaying data in the WATCH window

The debugger has one command that you can use to add 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 8-16 points to these watch indexes.) The WD command's *index number* parameter must correspond to one of the watch indexes in the WATCH window.

Note that deleting an item (depending on where it is in the list) causes the remaining index numbers to be reassigned. Deleting the last remaining item in the WATCH window closes the WATCH window.

Note:

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

8.8 Managing Pipeline Information (Simulator Only)

The simulator allows you to monitor the pipeline through pseudoregisters that you can query with ? or DISP or that you can add to the WATCH window.

The instruction pipeline consists of five phases: instruction fetch, decode, operand access 1, operand access 2, and execution. During any cycle, one to five 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 ARAU updates auxiliary registers during the operand-access-1 phase.

The simulator provides ten pseudoregisters that display the opcode or address of the instructions in each phase of the pipeline. Table 8–1 identifies these registers.

Table 8–1. Pipeline Pseudoregisters

Pipeline phase	Opcode pseudoregister	Address pseudoregister
Instruction prefetch	p_ins	p_add
Instruction fetch	f_ins	f_add
Instruction decode	d_ins	d_add
Operand access	a_ins	a_add
Operand read	r_ins	r_add
Instruction execute	x_ins	x_add

For example, if you wanted to observe the decode phase during program execution, you could watch the de_ins and de_addr pseudoregisters in the WATCH window:

```
wa d_ins,Decode-Opcode ↵
wa d_add,Decode-Address ↵
```

This adds d_ins and d_add to the WATCH window and labels them as Decode-Opcode and Decode-Address, respectively.

8.9 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 8–2 lists the available formats and the corresponding characters that can be used as the *display format* parameter.

Table 8–2. Display Formats for Debugger Data

Display Format	Parameter	Display Format	Parameter
Default for the data type	*****	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**

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

Table 8–3. 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 an unsigned decimal, enter:

  `setf short, u` ↵

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

  `setf short, *` ↵

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

  `setf` ↵

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

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

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

  `setf *` ↵

Changing the default format with ?, MEM, 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. However, there is one exception: you cannot use the s display format parameter with the MEM command.

Chapter 9

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 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 7-17) and benchmarking (described on page 7-19).

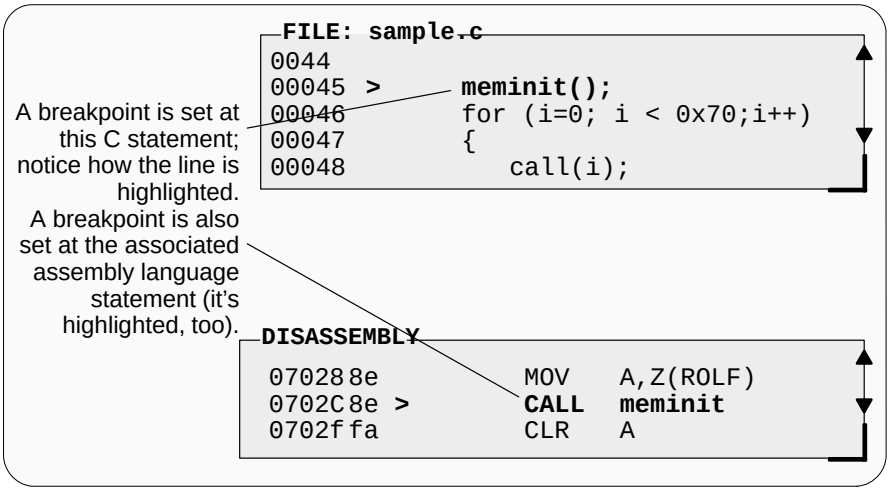
Topic **Page**

9.1 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.)



Notes:

1) After execution is halted by a breakpoint, you can continue program execution by reissuing any of the run or single-step commands.

2) You can set up to 200 breakpoints.

There are several ways to set a software breakpoint:

1) Make the FILE or DISASSEMBLY window the active window.

2) Point to the line of assembly language code or C code where you'd like to set a breakpoint.

3) Click the left mouse button.

Repeating this action clears the breakpoint.

1) Make the FILE or DISASSEMBLY window the active window.

2) Use the arrow keys to move the cursor to the line of code where you'd like to set a breakpoint.

3) Press the F9 key.

Repeating this action clears the breakpoint.

ba If you know the address where you'd like to set a 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.

9.2 Clearing a Software Breakpoint

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

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.

9.3 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
004d        in main, at line 60, "c:\c5xxhll\sample.c"
0051
```

The address is the memory address of the breakpoint. The symbolic information identifies the function, line number, and filename of the breakpointed C statement:

- If the breakpoint was set in assembly language code, you'll see only an address unless the statement defines a symbol.
- If the breakpoint was set in C code, you'll see the address together with symbolic information.

Chapter 10

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, the highlighting of the current statement, etc. In addition, if you're using a color display, you can change the colors of any area on the screen. Once you've customized the display to your liking, you can save the custom configuration for use in future debugging sessions.

Topic **Page**

10.1 Changing the Colors of the Debugger Display

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

Table 10–1. Colors and Other Attributes for the COLOR and SCOLOR Commands

(a) Colors

black	blue	green	cyan
red	magenta	yellow	white

(b) Other attributes

bright	blink

The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

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

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

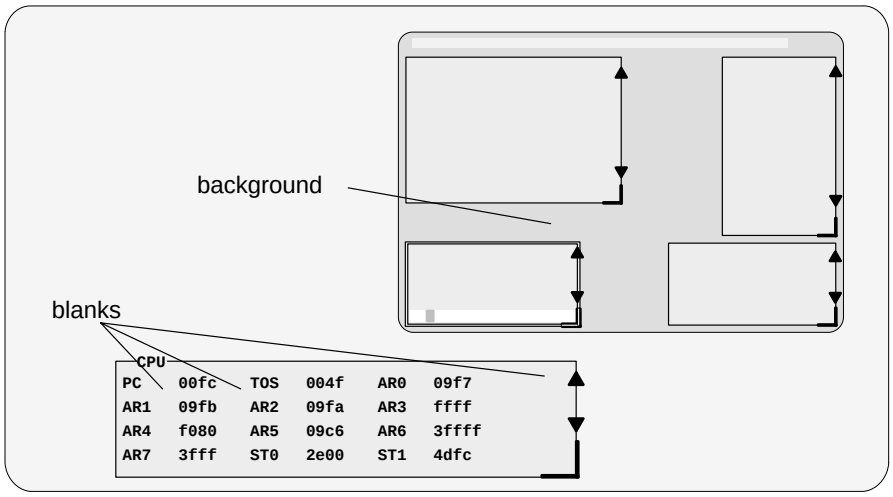
menu_bar	menu_border	menu_entry	menu_cmd
menu_hilite	menu_hicmd	win_border	win_hiborder
win_resize	field_text	field_hilite	field_edit
field_label	field_error	cmd_prompt	cmd_input
cmd_cursor	cmd_echo	asm_data	asm_cdata
asm_label	asm_clabel	background	blanks
error_msg	file_line	file_eof	file_text
file_brk	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 in which they're listed in Table 10–2 (left to right, top to bottom).

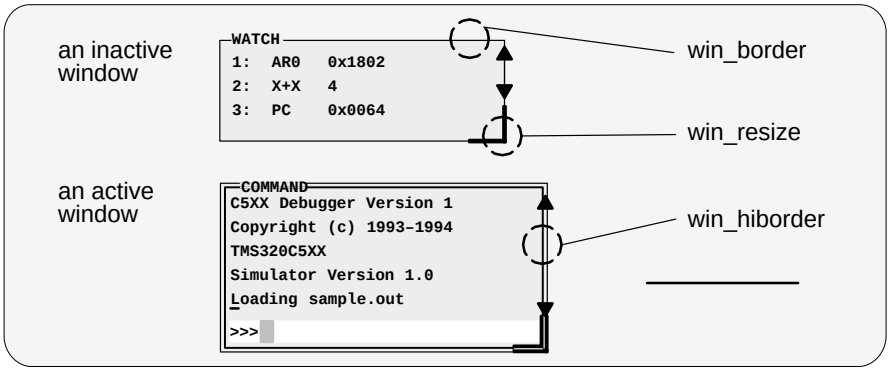
The remainder of this section identifies these areas.

Area names: common display areas



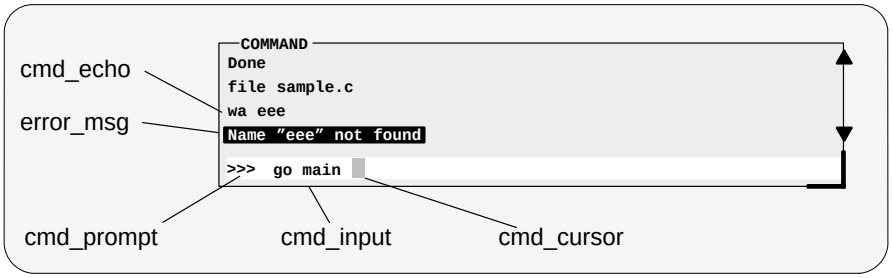
Area identification	Parameter name
Screen background (behind all windows)	background
Window background (inside windows)	blanks

Area names: window borders



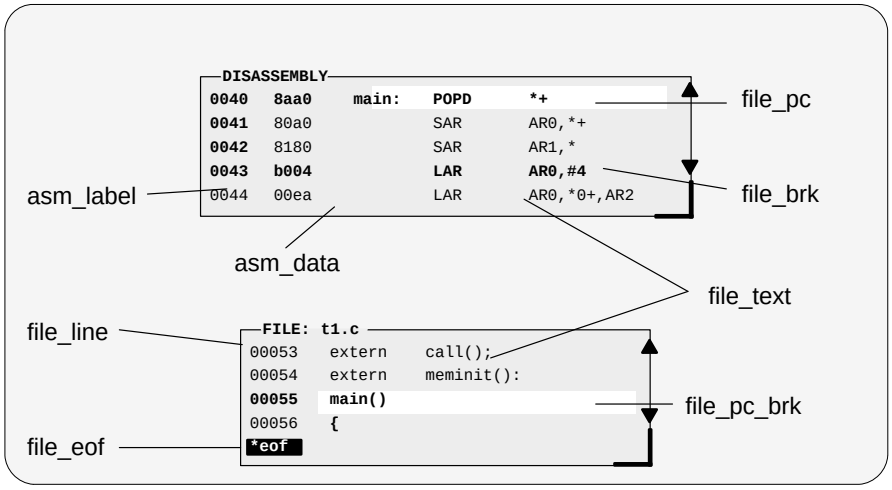
Area identification	Parameter name
Window border for any window that isn't active	win_border
The reversed L in the lower right corner of a resizable window	win_resize
Window border of the active window	win_hiborder

Area names: COMMAND window



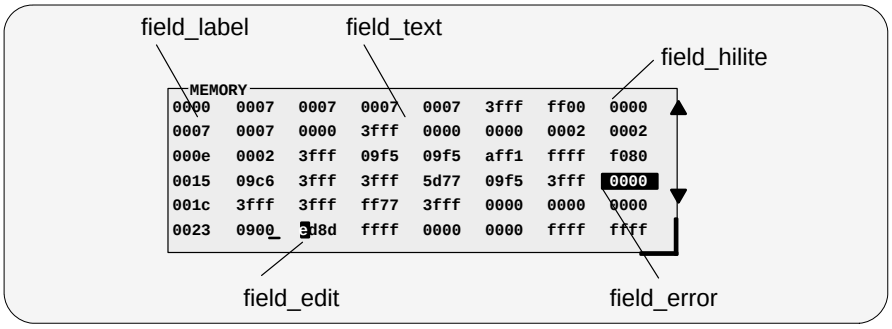
Area identification	Parameter name
Echoed commands in display area	cmd_echo
Errors shown in display area	error_msg
Command-line prompt	cmd_prompt
Text that you enter on the command line	cmd_input
Command-line cursor	cmd_cursor

Area names: DISASSEMBLY and FILE windows



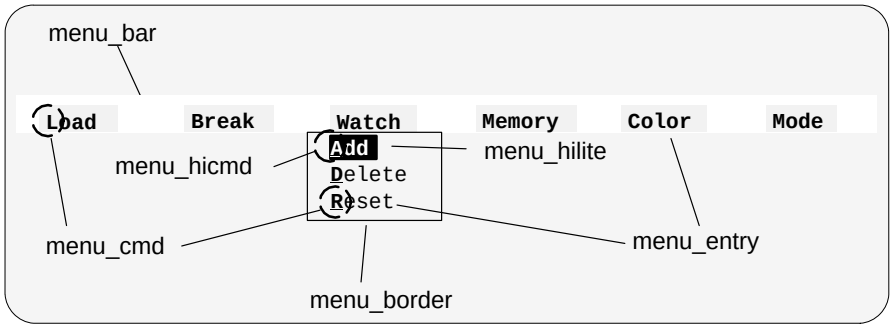
Area identification	Parameter name
Object code in DISASSEMBLY window	asm_data
Addresses in DISASSEMBLY window	asm_label
Line numbers in FILE window	file_line
End-of-file marker in FILE window	file_eof
Text in FILE or DISASSEMBLY window	file_text
Breakpointed text in FILE or DISASSEMBLY window	file_brk
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

10.2 Changing the Border Styles of the Windows

In addition to changing the colors of areas in the display, the debugger allows you to modify the border styles of the windows.

border Use the BORDER command to change window border styles. The format for this command is:

border [*active window style*] [,[*inactive window style*] [, *resize style*]]

This command can change the border styles of the active window, the inactive windows, and any window that is being resized. The debugger supports nine border styles. Each parameter for the BORDER command must be one of the numbers that identifies these styles:

Table 10–3. BORDER Command Parameters

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.

```
border 6,7,8          Change the style of the active window and the
                                          inactive window, and resize windows.
border 1,,2     Change the style of the active window and resize windows
border ,3                 Change the style of the inactive window.
```

Note that you can execute the BORDER command as the Border selection on the Color pulldown menu. The debugger displays a dialog box so that you can enter the parameter values; in the dialog box, *active window style* is called *foreground*, and *inactive window style* is called *background*.

10.3 Saving and Using Custom Displays

The debugger allows you to save and use as many custom configurations as you like.

When you invoke the debugger, it looks for a screen configuration file called init.clr. The screen configuration file defines how various areas of the display will appear. If the debugger doesn't find this file, it uses the default screen configuration.

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 file 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, 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.

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, rename the file or delete it. If you are in the debugger, have changed the configuration, and would like to revert to the default, just execute the SCONFIG command without a filename.

10.4 Changing the Prompt

prompt The debugger enables you to change the command-line prompt by using the PROMPT command. The format of this command is:

prompt *new prompt*

The *new prompt* can be any string of characters, excluding semicolons and commas. If you type a semicolon or a comma, it terminates the prompt string.

Note that the SSAVE command doesn't save the command-line prompt as part of a custom configuration. The SCONFIG command doesn't change the command-line prompt. If you change the prompt, it stays changed until you change it again, even if you use SCONFIG to load a different screen configuration.

If you always want to use a different prompt, you can add a PROMPT statement to the init.cmd batch file that the debugger executes at invocation time.

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

Note:

Whenever you select a default group of processors, the group name becomes the command-line prompt for the PDM. You *cannot* use the PROMPT command to change the PDM's command-line prompt. To change the PDM prompt, use the SET command (see page 2-20).

Chapter 11

Using the Simulator Analysis Interface

The ’C5xx has an analysis module on the chip that allows the simulator to simulate hardware functions. The debugger provides you with easy-to-use windows and dialog boxes that let you set data breakpoints on the occurrence of certain simulated hardware functions.

The debugger accesses the on-chip analysis module through a special set of pseudoregisters. The dialog boxes described in this chapter provide a transparent means of loading these registers. You will access the simulator analysis data breakpoint feature, unlike many of the other debugger features, through dialog boxes rather than through commands.

Topic **Page**

11.1 Introducing the Analysis Interface

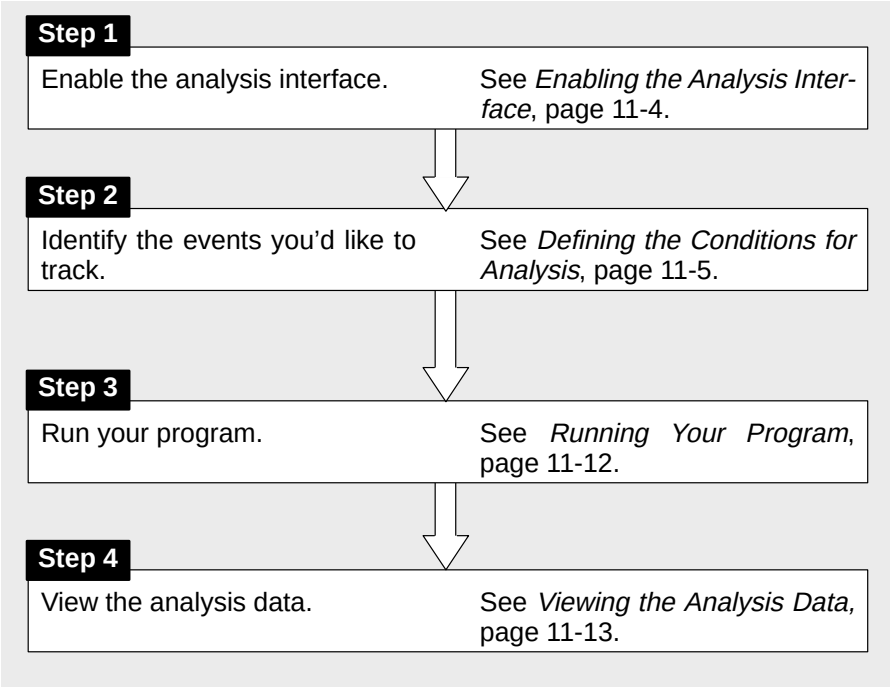
The ’C5xx analysis interface gives a detailed look into events occurring in hardware, expanding your debugging capabilities beyond software breakpoints. The analysis interface examines ’C5xx bus cycle information in real time and reacts to this information through actions such as data breakpoints.

The analysis interface allows you to set data breakpoints. You can monitor the data flow on the data or program bus to control access to specific memory addresses by setting simulated hardware breakpoints. This capability is critical when you are looking for a memory/pointer whose value is corrupted during execution for unknown reasons. You can monitor the following types of accesses during a debugging session:

- Read access on program or data memory
- Write access on program or data memory
- Read and write access on program or data memory.
- Read and/or write access for a particular data value
- Read and/or write access for a particular data pattern
- Instruction fetch on the program bus
- Data breakpoint between two instruction addresses

11.2 An Overview of the Analysis Process

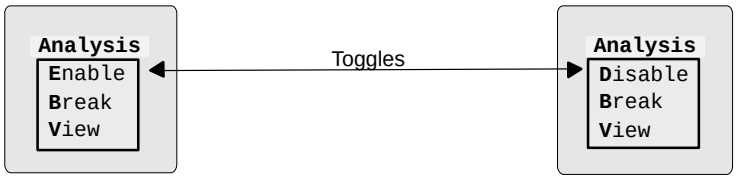
Completing an analysis session consists of four simple steps:



11.3 Enabling the Analysis Interface

To begin tracking events, you must explicitly enable the interface by selecting Enable on the Analysis menu. When you select Enable, the next time you open the menu Enable is replaced by Disable.

Figure 11–1. Enabling/Disabling the Analysis Interface



Selecting Disable turns the interface off; however, all events you previously enabled remain unchanged. By default, when the debugger comes up, the analysis interface is disabled. Selecting the Disable switch only temporarily suspends all analysis breakpoint checking until you select the Enable option again.

During a single debugging session, you may want to change the parameters of the analysis module several times. For example, you may want to define new parameters such as data bus accesses, etc. To do this, you must open the individual dialog boxes, deselect any previous events, and select the new events you want to track.

Note:

You have to enable the analysis interface only once during a debugging session. It is not necessary to enable the analysis interface each time you run your program.

11.4 Defining the Conditions for Analysis

The analysis module detects hardware events and monitors the internal signals of the processor. The interface to the analysis module allows you to define parameters that halt the processor.

First, however, you must define the conditions the analysis interface must meet to track a particular event. To do this, select the events you want to track by enabling the appropriate conditions in the Analysis Break Events dialog box found on the Analysis menu.

Halting the processor

You can set a hardware breakpoint on a program memory access or on a data bus address access. Table 11–1 below lists the type of accesses available:

Table 11–1. Types of Hardware Breakpoint Accesses

(a) Data memory

Access	Specifics	Criteria
Read	for any	data value
Write	for any	data value
Read and Write	for any	data value
Read	for a particular	data value
Write	for a particular	data value
Read and Write	for a particular	data value
Read	for a particular	data pattern
Write	for a particular	data pattern
Read and Write	for a particular	data pattern

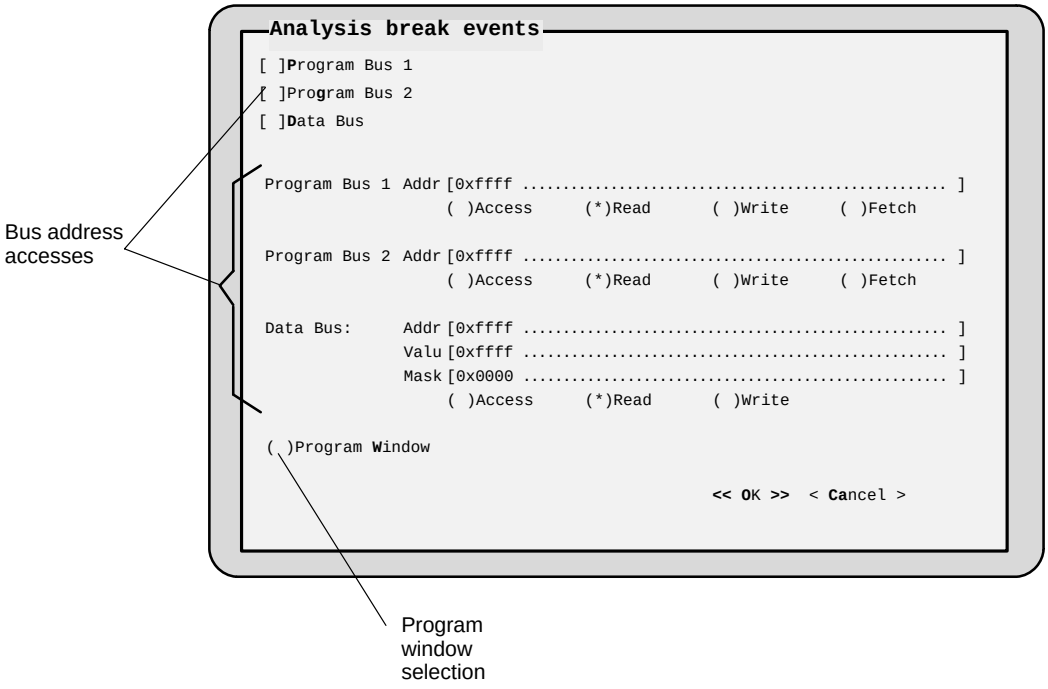
(b) Program memory

Access	Specifics	Criteria
Read	for any	value
Write	for any	value
Read and Write	for any	value
Fetch	for a particular	instruction word

The debugger allows you to specify one data memory breakpoint and two program memory breakpoints. Additionally, you can use the two program breakpoint addresses as a program window for activating data breakpoint checking.

Figure 11–2 shows the Analysis Break Events dialog box and the types of events that you can select.

Figure 11–2. Data Memory and Program Memory Breakpoints



To show an enabled event in the Analysis Break Events dialog box, the debugger displays an asterisk inside the parentheses preceding your selection. (Refer to Section 5.3 on page 5-11 for more information about selecting events in dialog boxes.)

When the program window selection is enabled, the debugger compares only values that occur in the logical window defined by program breakpoint 1 and program breakpoint 2. The program breakpoint 1 address is the window start address, and the program breakpoint 2 address is the window end address. The debugger checks for the data value according to the mask value within the logical window set by the program bus 1 address and the program bus 2 address.

If the mask value is 0, then any value in the given data address will be trapped, providing that the access type matches. If the mask value is 0xffff, then the mask value is not used; when the data address value matches the specified data value, the data breakpoint is trapped. Otherwise, the mask value and the data value will be used to continue checking for the desired data pattern according to the following algorithm:

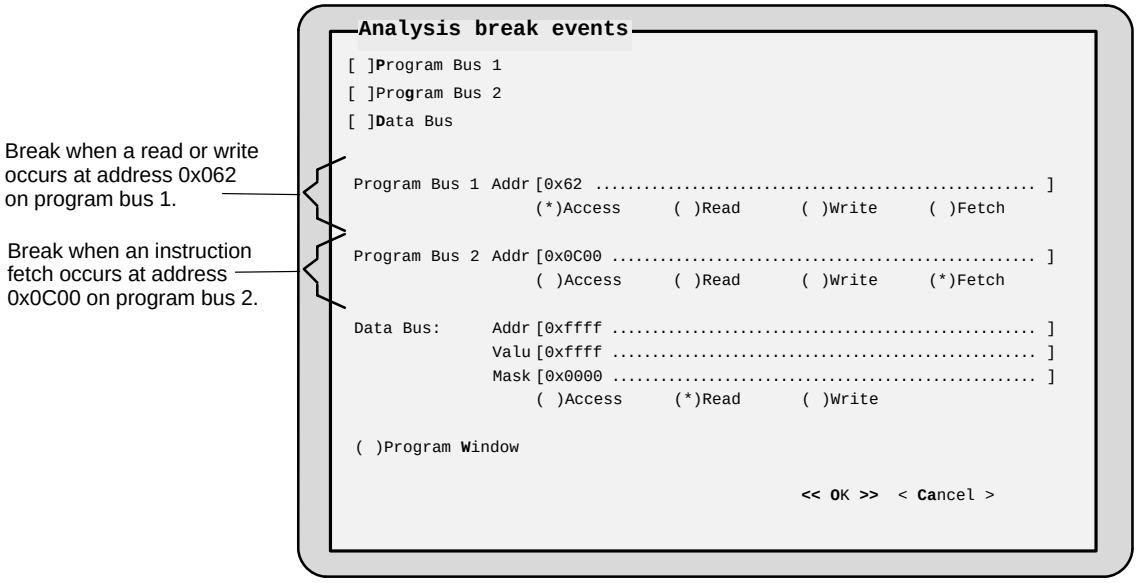
```
if ( NOT( (data_bus XOR data_val) AND mask_val) )
  {
   data_break_point_found = TRUE ;
  }
```

The above algorithm entails the following steps:

1) Find the difference (exclusive OR) between the value on the data bus and the data value specified.
2) Mask the result of step 1 to get the desired bits.
3) If the result of step 2 is 0, the desired data pattern was found, else the data pattern was not found.

Figure 11–3 shows the use of program bus data breakpoint events.

Figure 11–3. A Sample Data Breakpoint Set Up



In Figure 11–3, the debugger halts the processor whenever it detects a read or write at the program bus 1 address 0x62, or an instruction fetch at any program bus 2 address.

Instruction pipelining

The ’C5xx debugger uses a six-phase pipeline to process instructions. These phases are described in Table 11–2.

Table 11–2. Pipeline Phases

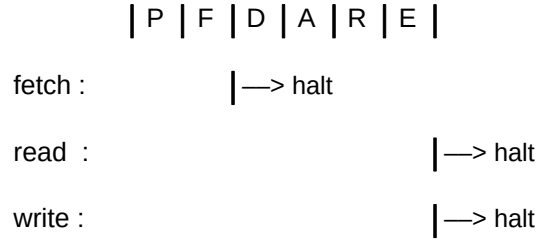
Phase	Initial	Description
Prefetch	P	Program memory is accessed via the program address bus.
Fetch	F	Program memory is read through the program bus. The instruction is then loaded in the instruction register.
Decode	D	Instructions are loaded into the instruction register and decoded.
Access	A	The address of the read operand on the data address bus is read. Auxiliary registers are also updated during the access phase.
Read	R	The read data operand is read from the data bus so it will be available input for the next step.
Write or Execute	E	The read data is executed; the data operand is sent to the data write bus.

Executing breakpoints

The simulator detects program address and data address breakpoints in the pipeline. It executes the breakpoint according to the type of breakpoint and the location where the breakpoint has been set, as explained below:

- Program address breakpoint

  This six-phase pipeline diagram shows you where in the pipeline a program address breakpoint actually halts the simulator.



  If you have a program address breakpoint set on a program fetch, then the simulator halts when the fetch phase is complete, before the instruction is decoded. However, if the address breakpoint is set on a read or write access, the instruction is executed completely before the simulator halts.

- Data address breakpoint

  | P | F | D | A | R | E |

  read : |—> halt

  write : |—> halt

  For a data address breakpoint, the instruction executes before the processor halts.

- Program window feature

  | P | F | D | A | R | E |

  fetch0: |—> enable data breakpoint checking

  fetch1: |—> disable data breakpoint checking

  With the logical program window enabled, the instruction specified by the program bus 1 address completes its read phase; then the simulator begins data breakpoint checking. Data breakpoint checking continues until the instruction specified by the program bus 2 address completes its read phase, then the simulator deactivates data breakpoint checking.

Note:

If you combine the last two items above, you see that data breakpoints resulting from instructions at the program window start address (specified by program bus 1 breakpoint address) are detected and notified, but data breakpoints caused by the instruction at the program window end address (specified by program bus 2 breakpoint address) are not detected.

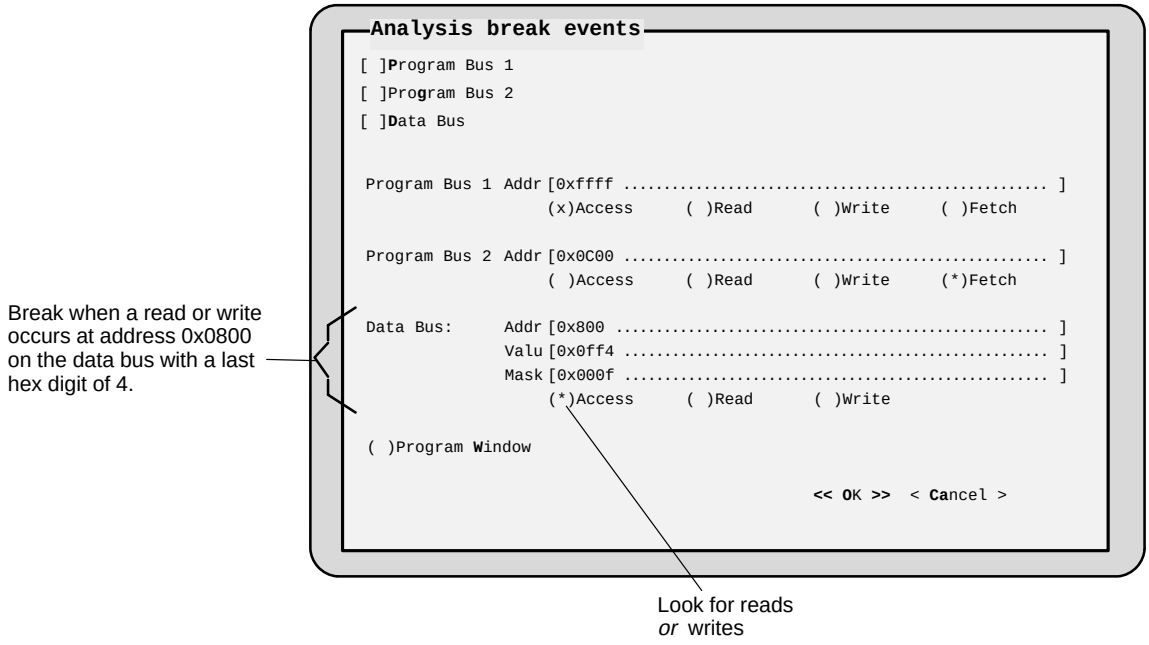
Setting up the event comparators

The analysis module has separate event comparators for the program and data buses. You can set up the analysis interface to halt the processor on accesses to a specified address.

The program bus supports noninstruction read and write accesses and program fetches. If you enable the access qualifier, noninstruction reads and writes are detected. If you enable the fetch qualifier, only program fetches are detected.

In Figure 11–4, the data address field is loaded with 0x0800 and *Access* is selected in the Analysis break events dialog box. The processor halts every time a read or write occurs at the data memory address 0x0800, and the program reads from or writes to the 0x0800 location any value with a last hex digit of 4 (mask value).

Figure 11–4. Set Up of Event Comparator

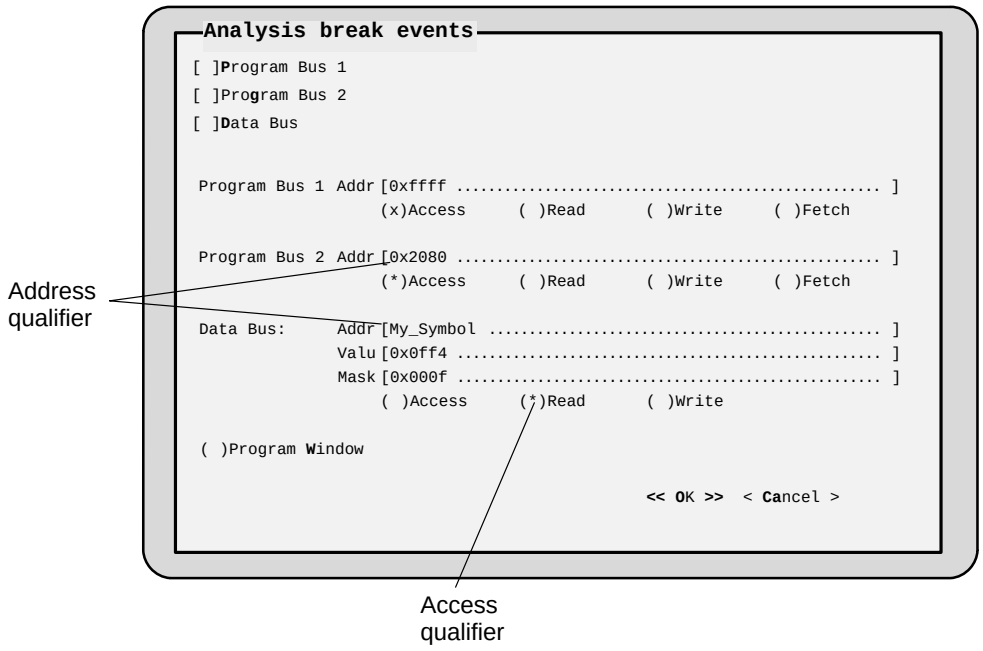


In Figure 11–5, program bus 2 and the data bus are enabled in the Analysis break events dialog box; however, data and program bus accesses need further qualification. They need:

- Address qualification
- Access qualification

For example, *Read* and *My_Symbol*, which are selected for the data bus, represent access and address qualifiers, respectively. They further define the conditions necessary to halt the processor.

Figure 11–5. Enabling Break Events



Address qualification allows you to enter an address expression (a specific address, symbol, or function name). Access qualification allows you to track reads, writes, both reads and writes (accesses), or program fetches from a program memory address. In Figure 11–5, the debugger halts the processor any time a read from the address at My_Symbol occurs.

11.5 Running Your Program

Once you have defined your parameters, the analysis interface can begin collecting data as soon as you run your program. It will stop collecting data when the defined conditions are met. The analysis interface monitors the progress of the defined events while your program is running. The basic syntax for the RUN command is:

run [*expression*]

You can use any of the debugger run commands (STEP, CSTEP, NEXT, etc.) described in Chapter 7 except the RUNF (run free) command.

Note:

The conditions for the analysis session must be defined *before* your analysis session begins; you cannot change conditions *during* execution of your program.

11.6 Viewing the Analysis Data

You can monitor the status of the analysis interface by selecting View on the Analysis menu. This window displays an ongoing progress report of the analysis module's activity. Through this window, you can monitor the status of the break events. An illustration of the Analysis window is shown below.

Figure 11–6. Global Analysis Breakpoint Checking Currently Disabled

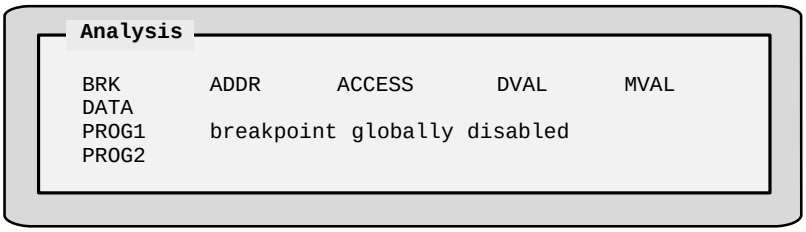
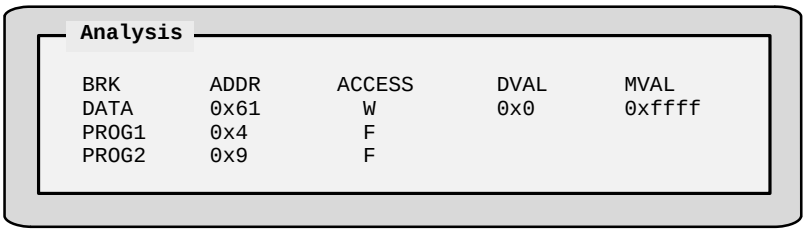


Figure 11–6 shows analysis breakpoint checking globally disabled.

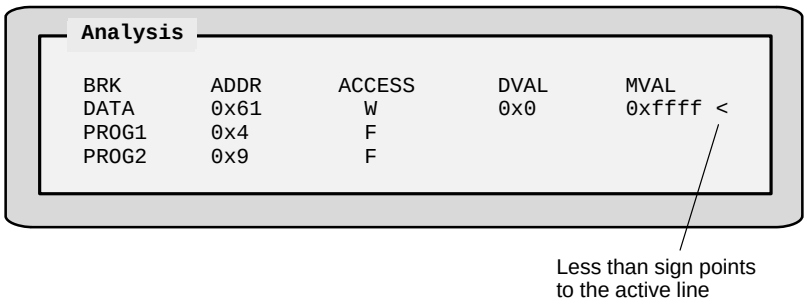
The BRK field displays the hardware breaks available: the data bus, program bus 1, and program bus 2. The ADDR field lists the address for entry of the BRK column. The ACCESS field lists the access qualifier set up for each line, if any. The DVAL field lists the data value, and the MVAL field lists the mask value for the data bus.

Figure 11–7. Analysis Interface View Window, Displaying a Beginning Status Report



Assume that a data breakpoint occurs during execution of the program. In the View window, the DATA row would be highlighted and a less than sign (<) placed to draw your attention as shown in Figure 11–8. Similarly, if the program bus 1 or the program bus 2 breakpoint was encountered, the corresponding row would be highlighted and marked with a less than sign (<).

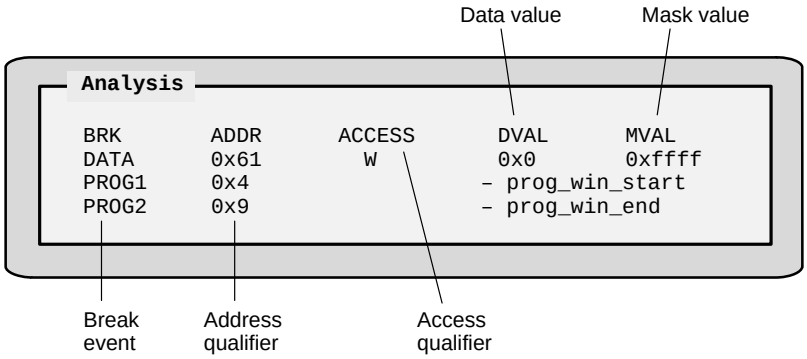
Figure 11–8. Analysis Interface View Window, Displaying an Ongoing Status Report



You can click on the PROG1 or PROG2 row during execution. Then you can adjust the disassembly window display with the program address displayed for the selected row in the View window.

Figure 11–9 shows the display when you select the program window option in the Analysis Break Events window. The address, access, data value, and mask value define the program window. See *Halting the Processor* beginning on page 11-5 for details on defining a logical program window.

Figure 11–9. Program Window Enabled



The Analysis window display updates as code executes and the selected break events occur. Multiple events can cause the processor to halt at the same time; these events are reflected in the BRK field of the Analysis window.

Setting a data read breakpoint with program window disabled

Data breakpoint is set at 0x62 for:
data value = 0x1924 and
mask value = 0x0f00 with
access type = Read

The initial setup is:

Data memory contents:	Register contents:
Location 0x61 has 0x3856.	ar3 has 0x61.
Location 0x65 has 0x0100.	ar4 has 0x62.
Location 0x62 has 0x0.	ar5 has 0x63.
	Accumulator A has 0.

Example 11–1. Data Read Breakpoint With Program Window Disabled

```
0  MVDD *ar3, *ar4  ; Move from location 0x61 to 0x62
1  ADD   A,@62h     ; Add location 0x61 to accumulator A
2  ADD   A,@65h     ; Add location 0x65 to accumulator A
3  DST   A,*ar5     ; Store A (high) to 0x63 and A (low) to 0x62
4  ADD   A,@62h     ; Add location 0x62 to A; 0x62 has 0x3956
5  ANDM  *ar4,0000h ; AND 0 with location 0x62; put result back
7  ORM   *ar4,0900h ; OR 0x0900 with loc 0x62; put result back
9  ADD   A,@62h     ; Add location 0x62 to A; 0x62 now 0x0900
10 NOP
```

If you execute the above instruction sequence, then you will see that data breakpoints are detected by the simulator. The data breakpoints detected are caused by the instruction at 4, then the instruction at 9.

Setting a data read breakpoint with program window enabled

The following criteria have been set for a data read breakpoint with the program window enabled:

- ❏ program_window start address (program bus breakpoint 1 address) = 0x5
- ❏ program_window end address (program bus breakpoint 2 address) = 0x10
- ❏ Data breakpoint is set at 0x62h for:
  data value = 0x1924 and
  mask value = 0x0f00 with
  access type = Read

The initial setup is:

Data memory contents:	Register contents:
Location 0x61 has 0x3856.	ar3 has 0x61.
Location 0x65 has 0x0100.	ar4 has 0x62.
Location 0x62 has 0x0.	ar5 has 0x63.
	Accumulator A has 0.

Example 11–2. Data Read Breakpoint With Program Window Enabled

```
0  MVDD *ar3, *ar4  ; Move from location 0x61 to 0x62
1  ADD   A,@62h     ; Add location 0x61 to accumulator A
2  ADD   A,@65h     ; Add location 0x65 to accumulator A
3  DST   A,*ar5     ; Store A(high) to 0x63 and A(low) to 0x62
4  ADD   A,@62h     ; Add location 0x62 to A; 0x62 has 0x3956
5  ANDM  *ar4,0000h ; AND 0 with location 0x62 put result back
7  ORM   *ar4,0900h ; OR 0x0900 with loc 0x62; put result back
9  ADD   A,@62h     ; Add location 0x62 to A; 0x62 now 0x0900
10 NOP
11 NOP
12 ADD    A,@62h    ; Add location 0x62 to A; 0x62 now 0x0900
13 NOP
14 NOP
```

If you execute the above instruction sequence, you will see that a data breakpoint is detected by the simulator. The data breakpoint is caused only by the instruction at 9. Data breakpoints are not detected at 4 and 12 because they are outside the program window.

Using Example 11–2, if you set the program window as follows, data breakpoints will be detected at 4 and 9:

- program_window start address = 0x4
- program_window end address = 0x10

A data breakpoint will be detected at 4 because the instruction at program_window start is inclusive within the program window. Refer to *Instruction pipelining*, page 11-8, for information on detecting breakpoints at the six pipeline phases.

Using Example 11–2, if you set the program window as follows, data breakpoints will be detected at 4 and 9:

- program_window start address = 0x4
- program_window end address = 0x12

A data breakpoint will not be detected at 12 because the instruction at program_window end is outside of the program window. Refer to *Instruction pipelining*, page 11-8, for information on detecting breakpoints at the six pipeline phases.

Chapter 12

Using the Emulator Analysis Interface

The ’C5xx has an analysis module on the chip that allows the emulator to monitor hardware functions. The debugger provides you with easy-to-use windows, dialog boxes, and analysis commands that let you count occurrences of certain hardware functions or set hardware breakpoints on these occurrences.

The debugger accesses the on-chip analysis module through a special set of pseudoregisters. The dialog boxes described in this chapter provide a transparent means of loading these registers. You will, in most cases, access the analysis features, unlike many of the other debugger features, through dialog boxes rather than through commands. If the dialog boxes do not meet your needs, you can use the special set of aliased commands that deal directly with the analysis pseudoregisters. These commands are described in Appendix A.

Topic **Page**

12.1 Introducing the Analysis Interface

The ’C5xx analysis interface provides a detailed look into events occurring in hardware, expanding your debugging capabilities beyond software breakpoints. The analysis interface examines ’C5xx bus cycle information in real time and reacts to this information through actions such as hardware breakpoints and event counting.

The analysis interface allows you to:

- **Count events.** The analysis interface can count nine types of *events*. You have the option of counting the number of times a defined event occurred during execution of your program or stopping your program after a certain number of events are detected.

  The analysis module has an *internal counter* that can count bus events as well as detect other internal events. Events that can be counted include:

  - Data accesses
  - Program accesses
  - CPU clock cycles
  - Calls taken
  - Pipeline clocks
  - Interrupts or traps taken
  - Returns from interrupts, traps, or calls
  - Instruction fetches
  - Branches taken

  You can count only one event at a time.

- **Set hardware breakpoints.** You can also set up the analysis interface to halt the processor during execution of your program. The events that cause the processor to stop are called *break events*. A break event can define a variety of conditions, including:

  - Data accesses
  - Discontinuity
  - Program accesses
  - Calls taken
  - Event counter passing 0
  - Low levels on EMU0/1 pins (EMU0 and EMU1)
  - Interrupts or traps taken
  - Returns from interrupts, traps, or calls
  - Pipeline clock
  - Branches taken

  Hardware break events allow you to set breakpoints in ROM as well as set separate breakpoints on program and data accesses. This enables you to break on events that you cannot break on by using software breakpoints alone. In addition, any of the debugger’s basic features available with software breakpoints can also be used with hardware breakpoints. As a result, you can take advantage of all the step and run commands.

- **EMU0/1 pins.** In a system of multiple ’C5xx processors connected by EMU0/1 (emulation event) pins, setting up the EMU0/1 pins allows you to create global breakpoints. Whenever one processor in your system reaches a breakpoint (software or hardware), *all* processors in the system can be halted.

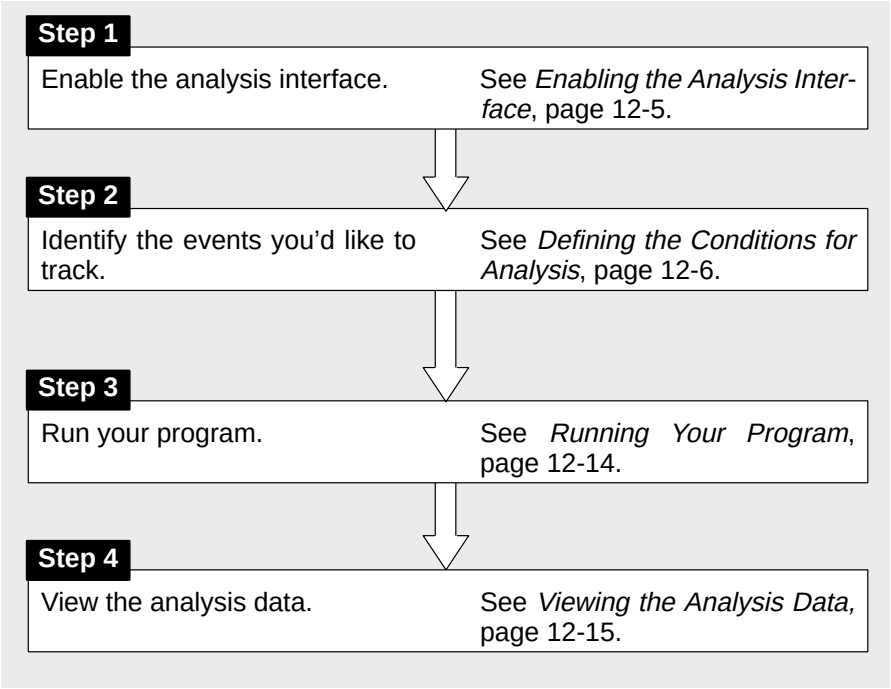
  In addition to setting global breakpoints, you can set up the EMU0/1 pins to take advantage of the emulator’s external counter. The *external counter* keeps track of the internal counter; each time the internal counter passes 0, a signal is sent through the EMU0/1 pins, incrementing the external counter.

- **The PC discontinuity stack.** *Discontinuity* occurs when the addresses fetched by the debugger become nonsequential as a result of loading the PC (through branches, calls, or return instructions, for example) with new values.

  You can view these values through the PC discontinuity stack and easily track the progress of your program to see exactly how the debugger reached its current state.

12.2 An Overview of the Analysis Process

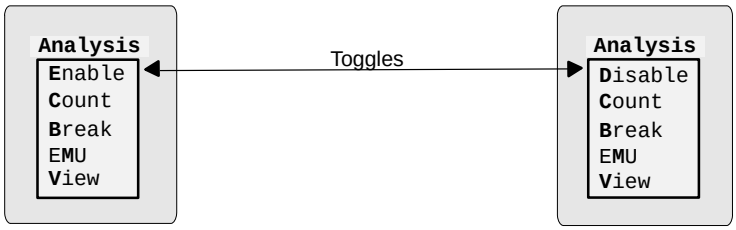
Completing an analysis session consists of four simple steps:



12.3 Enabling the Analysis Interface

To begin tracking hardware events, you must explicitly enable the interface by selecting Enable on the Analysis menu. When you select Enable, the next time you open the menu Enable is replaced by Disable.

Figure 12–1. Enabling/Disabling the Analysis Interface



Selecting Disable turns the interface off; however, all events you previously enabled remain unchanged. By default, when the debugger comes up, the analysis interface is disabled.

During a single debugging session, you may want to change the parameters of the analysis module several times. For example, you may want to define new parameters such as data bus accesses, tracking CPU clock cycles, etc. To do this, you must open the individual dialog boxes, deselect any previous events, and select the new events you want to track.

Note:

You have to enable the analysis interface only once during a debugging session. It is not necessary to enable the analysis interface each time you run your program.

12.4 Defining the Conditions for Analysis

The analysis module detects hardware events and monitors the internal signals of the processor. The interface to the analysis module allows you to define parameters that count events or halt the processor.

First, however, you must define the conditions the analysis interface must meet to track a particular event. To do this, select the events you want to track by enabling the appropriate conditions in the Analysis count events, Analysis break events, or Emulator Pins dialog boxes found on the Analysis menu.

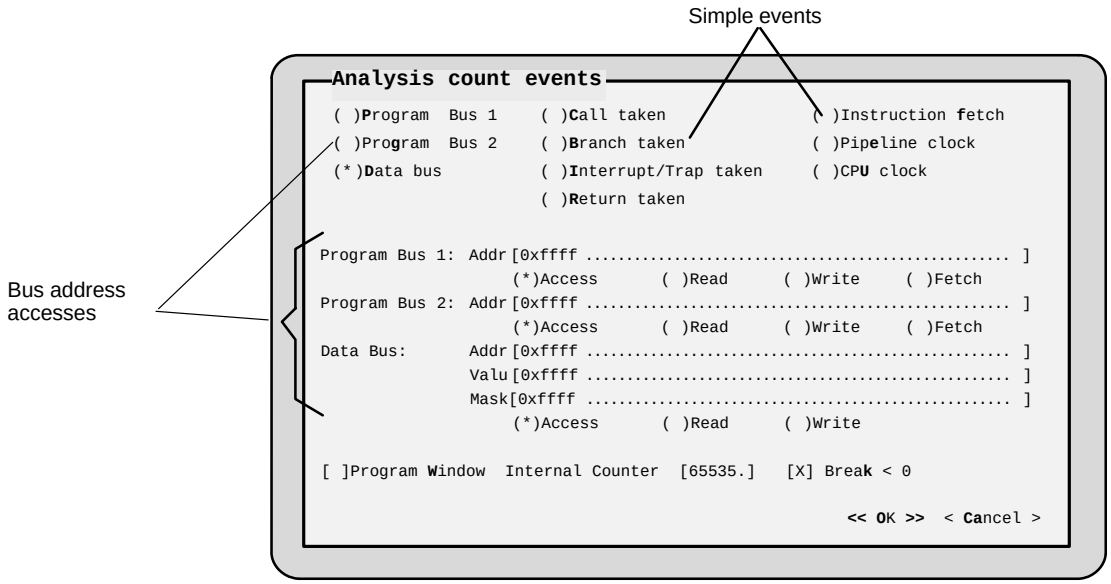
Counting events

You can count two basic types of events:

- Simple events
- Bus address accesses

Figure 12–2 shows the Analysis count events dialog box and the types of events that you can select.

Figure 12–2. The Two Basic Types of Events That Can Be Counted



You can count only one type of event at a time. To show an enabled event in the Analysis count events dialog box, the debugger displays an asterisk inside the parentheses preceding your selection. In this example, the debugger counts the number of instructions fetched. (Refer to Section 5.3 on page 5-11 for more information about selecting events in dialog boxes.)

You can use event counting in one of two ways: you can either stop processing after a certain number of events are detected, or you can count the number of times a defined event occurs. To count the number of times an event occurred, simply enable the event in the Analysis count events dialog box as shown in Figure 12–2.

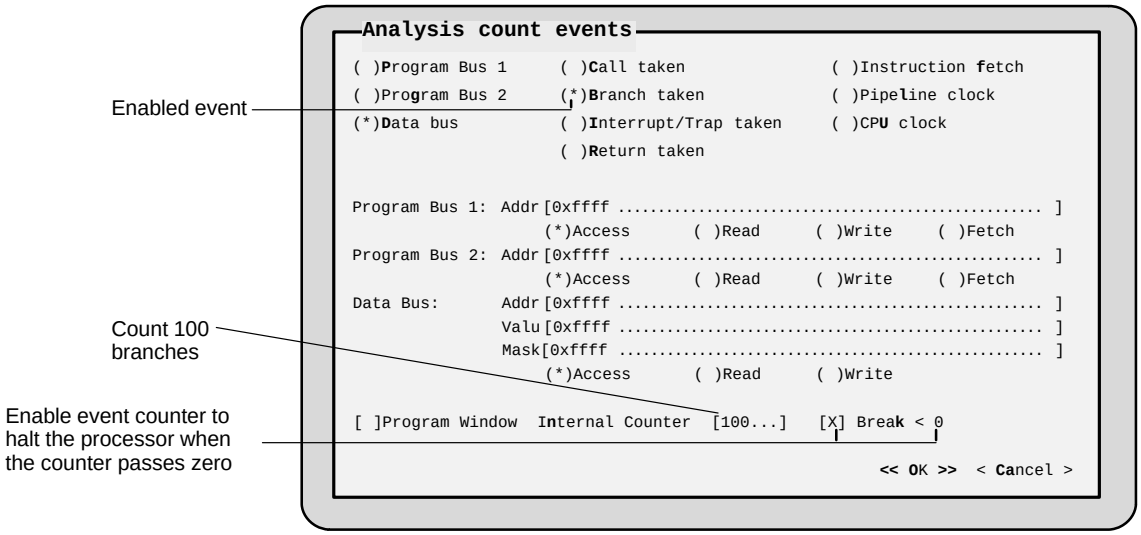
The analysis module has an internal counter that can count bus events as well as detect other internal events. This counter keeps track of how many times an event occurs. Therefore, to stop after a certain number of events are detected, you must:

- Specify the event you want to count,
- Enable the internal counter *(Break < 0*), and
- Load the counter with the number of events you want to count.

The internal counter decrements each time the specified event is detected.

For example, you may want to follow the progress of the branches taken during execution of your program, but you may want the processor to stop after 100 branches have occurred. In this case, the counter is responsible for keeping track of the branches taken and signaling the processor to stop after the 100th branch event occurs. When the internal counter passes 0, the debugger will halt the processor.

Figure 12–3. Enabling the Event Counter



To watch the progress of the event counter, open the Analysis window by selecting View on the Analysis menu. For more information on the Analysis window, see Section 12.6 on page 12-15.

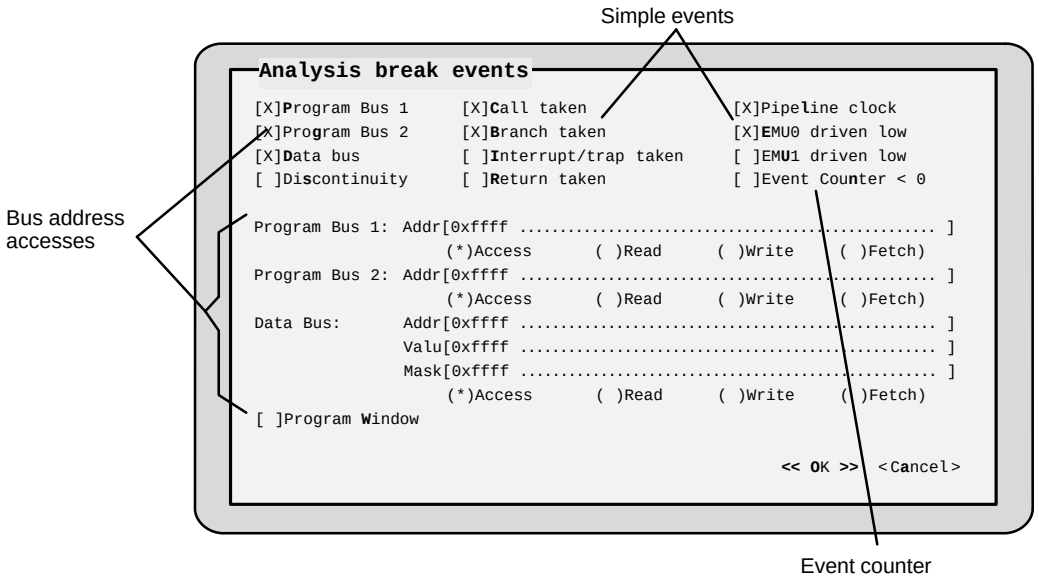
Halting the processor

You can set a hardware breakpoint on three basic types of events:

- The event counter passing 0
- Simple events
- Bus address accesses

Figure 12–4 shows the Analysis break events dialog box and the different types of break events that you can select.

Figure 12–4. Three Basic Types of Break Events



The events that cause the processor to halt when a specified event is detected include:

- Event counter < 0
- Calls taken
- Branches taken
- Instruction fetches
- Pipeline clocks
- Interrupts or traps taken
- Returns from interrupts, traps, or calls
- Discontinuity
- Low levels on EMU0/1 pins

Enabling events in the Analysis Break Events dialog box is like turning a switch on and off. When an event is enabled, the debugger displays an X next to the event. You can enable as many events as you want.

In Figure 12–4, the debugger halts the processor whenever it detects the occurrence of a call taken, a branch taken, an instruction fetch, the EMU0 pin being driven low, or a program or data bus being accessed.

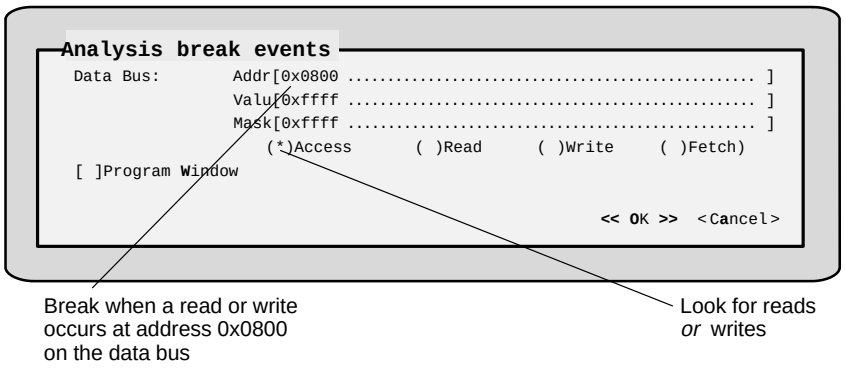
Setting up the event comparators

The analysis module has separate event comparators for the program and data buses. You can set up the analysis interface to either count the number of accesses to a certain bus address or halt the processor on accesses to a specified address.

The program and data bus fields in the Analysis break events dialog box are identical to the program and data bus fields in the Analysis count events dialog box. (The event comparators shown in Figure 12–2, Figure 12–3, and Figure 12–4 all have the same values.) As a result, changing the values in these fields in either dialog box affects *both* of these dialog boxes.

The program bus supports noninstruction read and write accesses and instruction fetches. If you enable the Access qualifier, noninstruction reads and writes are detected. If you enable the Fetch qualifier, only program fetches are detected.

Similarly, if you were to load the data address field with 0x0800 and select *Access* in the Analysis break events dialog box, the processor would halt every time a read or write occurs on the data bus at address 0x0800:



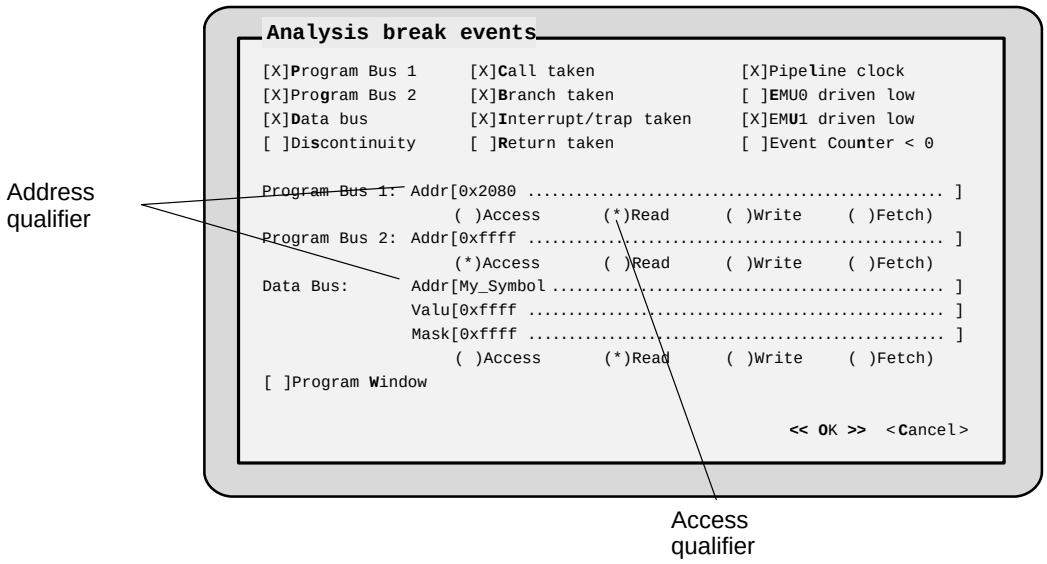
Enabling one of these comparators in the Analysis Count Events dialog box allows you to count the number of accesses that are detected or to stop processing after a certain number of accesses have occurred.

In Figure 12–5, several events are enabled in the Analysis Break Events dialog box, including the data bus; however, data and program bus accesses need further qualification. They need:

- Address qualification
- Access qualification

For example, *Read* and *My_Symbol*, which are selected for the data bus, represent access and address qualifiers, respectively. They further define the conditions necessary to halt the emulator.

Figure 12–5. Enabling Break Events



Address qualification allows you to enter an address expression (a specific address, symbol, or function name). Access qualification allows you to track reads, writes, both reads and writes (accesses), or program fetches to a single address. In Figure 12–5, the debugger halts the processor any time a read from the address at My_Symbol occurs.

When the program window selection is enabled, the debugger compares only values that occur in the logical window defined by program breakpoint 1 and program breakpoint 2. The program breakpoint 1 address is the window start address and the program breakpoint 2 address is the window end address. The debugger checks for the data value according to the mask value within the logical window set by the program bus 1 address and the program bus 2 address.

If the mask value is 0, then any value in the given data address will be trapped, providing that the access type matches. If the mask value is 0xffff, then the mask value is not used; when the data address value matches the specified data value, the data breakpoint is trapped. Otherwise, the mask value and the data value will be used to continue checking for the desired data pattern according to the following algorithm:

```
if ( NOT( (data_bus XOR data_val) AND mask_val) )
  {
   data_break_point_found = TRUE ;
  }
```

The above algorithm entails the following steps:

1) Find the difference (exclusive OR) between the value on the data bus and the data value specified.

2) Mask the result of step 1 to get the desired bits.

3) If the result of step 2 is 0, the desired data pattern was found, else the data pattern was not found.

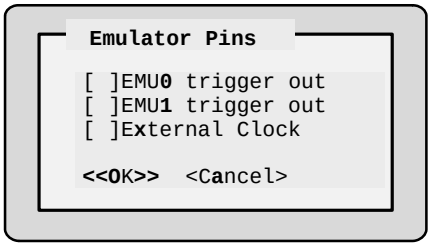
Setting up the EMU0/1 pins

The analysis interface allows you to access and set up the EMU0/1 (emulation event) pins on your processor. You can set these pins to:

- Use the emulator's external counter, or
- Set global breakpoints.

Selecting EMU0/1 from the Analysis menu opens the Emulator Pins dialog box shown in Figure 12–6.

Figure 12–6. The Emulator Pins Dialog Box



The emulator's *external counter* keeps track of the internal counter. The internal counter is a 16-bit, count-down counter that can keep track of a maximum of 65536 events. The external counter, however, is a 32-bit, incremental counter. Each time the internal counter passes 0, a signal sent through the EMU0/1 pins increments the external counter. To use the emulator's external counter, simply enable the external clock parameter in the Emulator Pins dialog box. (Refer to Section 5.3, page 5-11, for more information on enabling pa rameters in a dialog box.)

When you enable the external clock, the EMU0/1 pins are set up as totem-pole outputs; otherwise, by default the EMU0/1 pins are set up as open-collector outputs. You can only set up *one* 'C5xx device in the system to use the external counter. In doing so, no other device in the system can have EMU0/1 pins set up to trigger out.

The EMU1 pin provides a ripple-carry output signal from the internal counter that increments the emulator counter. The 'C5xx EMU0 pin is set up to send a signal to the debugger when a hardware or software breakpoint occurs. Other devices in the system can still be programmed to detect low levels on the EMU0 pin to provide you with global breakpoint capabilities.

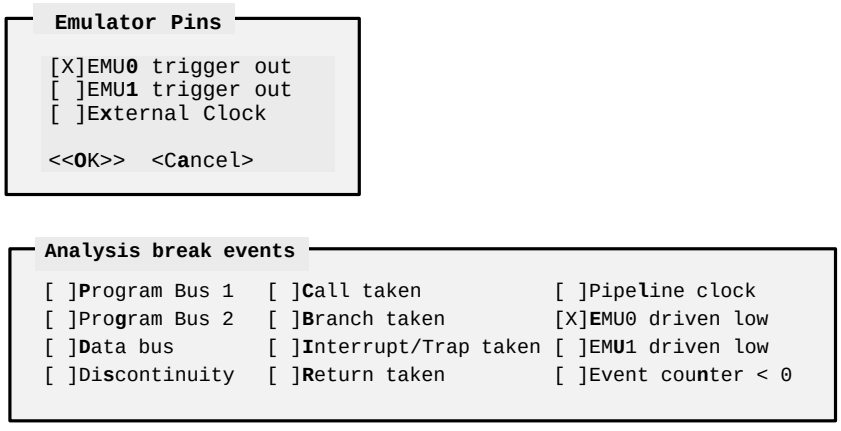
Notes:

1) Enabling the external clock disables all internal clock options in the Analysis break events and Analysis count events dialog boxes.
2) Enabling the external clock in the Emulator Pins dialog box carries the following restrictions:
   - You can enable only one external clock when you have multiple processors (that are connected by their EMU0/1 pins) in a system.
   - No other external devices can actively drive the EMU0/1 pins.

By default, the EMU0/1 pins are set up as input signals; however, you can set them up as output signals or *trigger out* whenever the processor is halted by a software or hardware breakpoint. This is extremely useful when you have multiple ’C5xx processors in a system connected by their EMU0/1 pins.

Selecting EMU0/1 does not, however, automatically halt all processors in the system. To do so, you must enable the EMU0/1 driven-low condition in the Analysis break events dialog box. For example, if you have a system consisting of two processors connected by their EMU0 pins and you want to halt both processors when this pin is driven low, you must enable the *EMU0 trigger out* parameter. Then you must enable the parameter *EMU0 driven low* in the Analysis break events dialog box, as shown in Figure 12–7.

Figure 12–7. Setting Up Global Breakpoints on a System of Two ’C5xx Processors



Setting up each processor in this way creates a global breakpoint so that any processor that reaches a breakpoint halts all other processors in the system.

12.5 Running Your Program

Once you have defined your parameters, the analysis interface can begin collecting data as soon as you run your program. It will stop collecting data when the defined conditions are met. The analysis interface monitors the progress of the defined events while your program is running. The basic syntax for the RUN command is:

run [*expression*]

You can use any of the debugger run commands (STEP, CSTEP, NEXT, etc.) described in Chapter 7 except the RUNF (run free) command.

The analysis interface provides capabilities in addition to those provided by the RUNB command. You may notice that with the RUNB command you can count the number of CPU clock cycles only during the execution of a specific section of code. However, the analysis interface not only allows you to count CPU clock cycles, it also allows you to count other events.

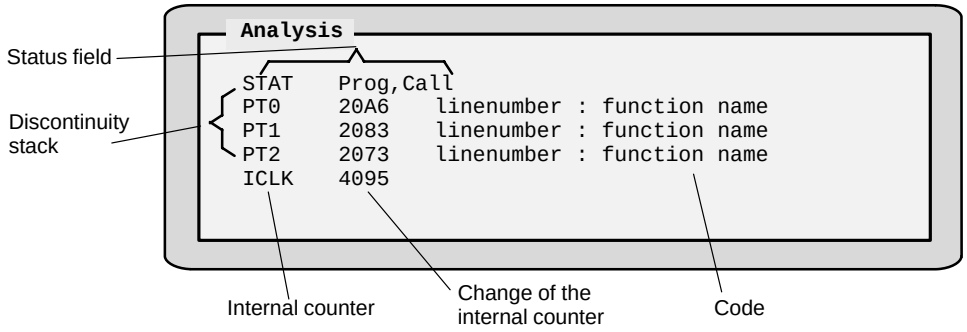
Note:

The conditions for the analysis session must be defined *before* your analysis session begins; you cannot change conditions *during* execution of your program.

12.6 Viewing the Analysis Data

You can monitor the status of the analysis interface by selecting View on the Analysis menu. This window displays an ongoing progress report of the analysis module's activity. Through this window, you can monitor the status of the break events, the value of both the internal and external event counters, and the values of the PC discontinuity stack. An illustration of the Analysis window is shown below.

Figure 12–8. Analysis Interface View Window, Displaying an Ongoing Status Report



Interpreting the status field

Multiple events can cause the processor to halt at the same time; these events are reflected in the STAT field of the Analysis window. The STAT field displays a list of the events that caused the processor to halt. If the analysis interface itself did not halt the processor, but something else (such as a software breakpoint) did, then the status line will display "No event detected".

Interpreting the discontinuity stack

The PC discontinuity stack allows you to see how the program reached its current position. The Analysis window displays both the PC discontinuity stack values and the corresponding C code. A program discontinuity occurs when the program addresses fetched by the debugger become nonsequential as a result of an action such as a branch or an interrupt.

The Analysis window has three fields that represent the PC discontinuity stack:

Analysis Field	Description
PT0	displays the address of the current code segment.
PT1	displays the address of the previous code segment.
PT2	displays the address of the oldest code segment.

For example, suppose you set a software breakpoint at address 0x00fb and the processor stopped on it.

Example 12–1. Sample Code and PC Discontinuity Values

(a) Sample code

Address		Code		Comment
00c4		retd		
00c5	call:	frame	0ffffh	
		.		
		.		
00d4		ld	@00h,1,a	*PT2 address*
00d6		st1	a,*(000f4h)	
00d8		ld	@00h,a	
00d9		call	xcall	
00db		b	call+47 (00f4h)	*PT1 address*
00dd		ld	@00h,a	
		.		
		.		
00eb		sub	*(00008h),b	
00ed		bc	call+15 (00d4h),beq	
00ef		ld	#003h,b	
		.		
		.		
		.		
00f8	xcall:	frame	0ffffh	*PT0 address*
		nop		
	>	stl	a,@00h	*BP address*

(b) PC discontinuity stack

```
Analysis

PT0    00f8          65:xcall( )
PT1    00dc          59:call( )
PT2    00d4          58:call( )
```

If you look at the PC discontinuity stack shown in Example 12–1 (b), you see that your program reached this point through a branch from address 0x00ed and a call from address 0x00d9. The PT2 and PT1 trace addresses contain the last addresses fetched before the branch and call instructions were executed. The PT0 trace address contains the destination of the call.

The fields next to the PT0, PT1, and PT2 fields list the C code associated with these addresses. This includes the function name and the line number within the function that caused the discontinuity. If no corresponding C code exists, the debugger displays “no function”. Clicking on any field in the PC discontinuity stack will cause the FILE and DISASSEMBLY windows to open (in the assembly or mixed mode). The address for that field is displayed in the DISASSEMBLY window, while the associated C code is shown in the FILE window. This allows you to easily track the PC discontinuity values back to their original source.

Interpreting the event counter

You can watch the progress of the event counters in the Analysis window. The CLK field displays the internal and external counter values with the appropriate prefix—I for internal, and X for external. The value shown next to the internal event counter represents the difference from the last counter value (*delta*). You can change the value of the internal counter by clicking on the appropriate field and entering a new value.

Note:

When counting CPU clock cycles, the counter value reflects startup and latency cycles.

Chapter 13

Profiling Code Execution

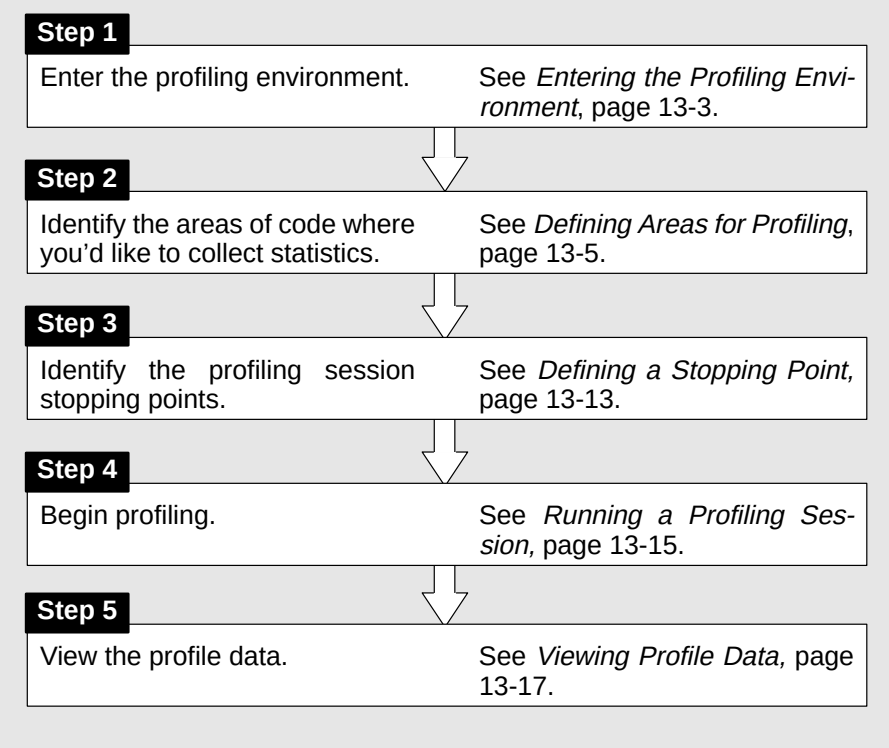
The profiling environment is a special debugger environment that lets you collect execution statistics for your code. All versions of the debugger support the profiling environment. This environment is available on all debugger platforms except DOS.

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**

13.1 An Overview of the Profiling Process

Profiling consists of five simple steps:



Note:

When you compile a program that will be profiled, you must use the –g and the –as 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.

13.2 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: **emu5xx –profile** ⏎
simulator: **sim5xx –profile** ⏎
EVM: **evm5xx –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 13–1 lists the debugger commands that can and can't be used in the profiling environment.

Table 13–1. 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			

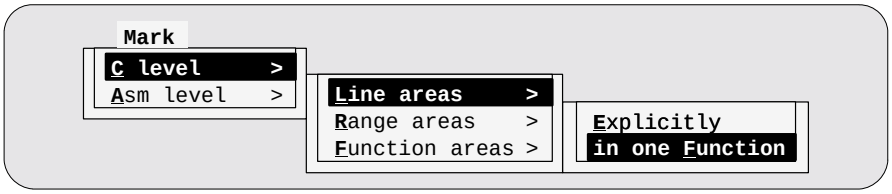
Be sure you don't use any of the *can't be used* commands in your initialization batch file.

Using pulldown menus in the profiling environment

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

The Load and mAp menus correspond to the Load and mAp menus available in the basic debugger environment. 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:



Chapter 14, *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.

13.3 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 they won't affect the profile data, and you can reenable 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 and more complex tasks.

The following subsections explain how to mark, disable, reenable, 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 13-3. Restrictions on the profiling areas are summarized on page 13-12.

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:

1) Marking an area in C *does not* mark the associated code in disassembly.
2) Areas can be nested; for example, you can mark a line within a marked range. The debugger will report statistics for both the line and the function.
3) 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 13–2 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 13–2. 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 13–3 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 13–3. 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		→**F**unction areas
❑ By function name	→**E**xplicitly	not applicable	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.

Enabling a disabled area

When an area has been disabled and you would like to profile it 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 using one of the commands listed in Table 13–4. However, the easiest way to enter these commands is by accessing them from the Enable menu.

Table 13–4. 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		→**F**unction areas
❑ By function name	→**E**xplicitly	not applicable	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 declares the function.

2) Click the right mouse button once.

The line will no longer be highlighted.

Table 13–5 lists the selections on the Unmark menu.

Table 13–5. Menu Selections for Unmarking Areas

To unmark this area	C only: **U**nmark→**C** level	Disassembly only: **U**nmark→**A**sm level	C *and* disassembly: **U**nmark→**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		→**F**unction areas
❑ By function name	→**E**xplicitly	not applicable	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.

13.4 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 an ***** 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 of 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 of 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

13.5 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 13.6, page 13-17). 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 is restarted and run up 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 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.

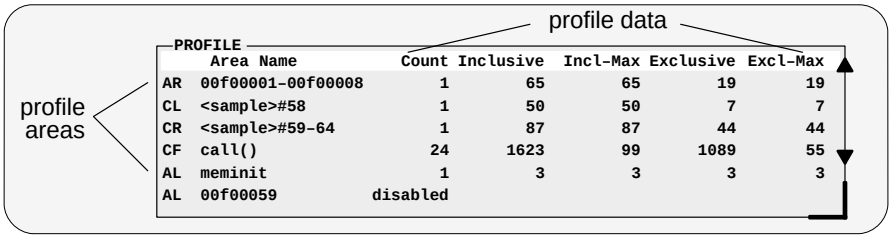
nonzero All previously collected profile data and internal profile stacks are cleared.

- The *update rate* parameter is the same as for the PF and PQ commands.

13.6 Viewing Profile Data

The statistics collected during a profiling session are displayed in the PROFILE window. Figure 13–1 shows an example of this window.

Figure 13–1. An Example of the PROFILE Window



The example in Figure 13–1 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

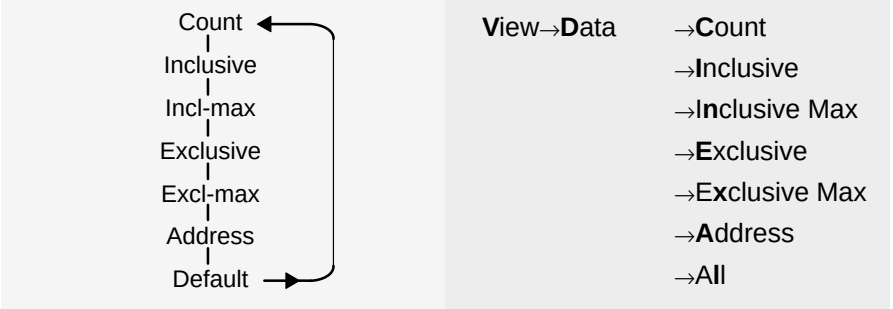
By default, the PROFILE window shows a set of statistics labeled Count, Inclusive, Incl-Max, Exclusive, and Excl-Max. The Address field, which is not included as part of the default statistics, can also be displayed. Table 13–6 describes the statistic that each field represents.

Table 13–6. 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.



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 13–7.

Table 13–7. 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		→**F**unction areas
❑ By function name	→**E**xplicitly	not applicable	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 right 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.

13.7 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 14

Summary of Commands and Special Keys

This chapter summarizes the basic debugger, profiling, and parallel debug manager (PDM) commands and the debugger's special key sequences.

Topic **Page**

14.1 Functional Summary of Debugger Commands

This section summarizes the debugger commands according to these categories:

- **Managing multiple debuggers.** These commands allow you to group debuggers, run code on multiple processors, and send commands to a group of debuggers.
- **Changing modes.** These commands enable you to switch freely between modes (auto, mixed, and assembly). You can select these commands from the Mode pulldown menu, also.
- **Managing windows.** These commands enable you to select the active window and move or resize the active window. You can perform these functions with the mouse, also.
- **Displaying and changing data.** These commands enable you to display and evaluate a variety of data items. Some of these commands are available on the Watch pulldown menu, also.
- **Performing system tasks.** These commands enable you to perform several DOS-like functions and provide you with some control over the target system.
- **Managing breakpoints.** These commands provide you with a command-line method for controlling software breakpoints. They are also available through the Break pulldown menu. You can also set/clear breakpoints interactively.
- **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.
- **Memory mapping.** These commands enable you to define the areas of target memory that the debugger can access or to fill a memory range with an initial value. You can also use the Memory pulldown menu to access these commands.
- **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.
- **Running programs.** These commands provide you with a variety of methods for running your programs in the debugger environment. The basic run and single-step commands are available on the menu bar, also.
- **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.

Managing multiple debuggers

To do this	Use this command	See page
Use the command history	!	14-13
Assign a variable to the result of an expression	@	14-13
Define a custom command string	alias	14-15
Record the information shown in the PDM display area	dlog	14-23
Display a string to the PDM display area	echo	14-24
Evaluate an expression in a debugger or group of debuggers and set a variable to the result	eval	14-25
List available PDM commands	help	14-28
View the description of a PDM command	help	14-28
List the last twenty commands	history	14-28
Conditionally execute PDM commands	if/elif/else/endif	14-29
Loop through PDM commands	loop/break/ continue/endloop	14-31
Pause the PDM	pause	14-42
Halt code execution	pesc	14-42
Global halt	phalt	14-43
Run code globally	prun	14-45
Run free globally	prunf	14-46
Single-step globally	pstep	14-46
Exit any debugger and/or the PDM	quit	14-47
Send a command to an individual processor or a group of processors	send	14-51
Change the PDM prompt	set	14-52
Check the execution status of the processors	set	14-52
Create your own system variables	set	14-52
Define or modify a group of processors	set	14-52
List all system variables or groups of processors	set	14-52
Set the default group	set	14-52
Invoke an individual debugger	spawn	14-56
Find the execution status of a processor or a group of processors	stat	14-57
Enter an operating-system command	system	14-58
Execute a batch file	take	14-59
Delete an alias definition	unalias	14-59
Delete a group or system variable	unset	14-60

Changing modes

To do this	Use this command	See page
Put the debugger in assembly mode	asm	14-15
Put the debugger in auto mode for debugging C code	c	14-18
Put the debugger in mixed mode	mix	14-39

Managing windows

To do this	Use this command	See page
Reposition the active window	move	14-39
Size the active window	size	14-54
Select the active window	win	14-63
Make the active window as large as possible	zoom	14-64

Displaying and changing data

To do this	Use this command	See page
Evaluate and display the result of a C expression	?	14-12
Evaluate a C expression without displaying the results	eval	14-25
Display the values in an array or structure or display the value that a pointer is pointing to	disp	14-22
Display a different range of memory in the MEMORY window	mem	14-37
Display a pop-up MEMORY window	mem[#]	14-37
Change the default format for displaying data values	setf	14-53
Continuously display the value of a variable, register, or memory location within the WATCH window	wa	14-62
Delete a data item from the WATCH window	wd	14-63
Show the type of a data item	whatis	14-63
Delete all data items from the WATCH window and close the WATCH window	wr	14-64

Performing system tasks

To do this	Use this command	See page
Define your own command string	alias	14-15
Change the current working directory from within the debugger environment	cd/chdir	14-18
Clear all displayed information from the COMMAND window display area	cls	14-19
List the contents of the current directory or any other directory	dir	14-22
Record the information shown in the COMMAND window display area	dlog	14-23
Display a string to the COMMAND window while executing a batch file	echo	14-24
Conditionally execute debugger commands in a batch file	if/else/endif	14-30
Loop debugger commands in a batch file	loop/endloop	14-32
Exit the debugger	quit	14-47
Reset the target system (emulator only), simulator, or EVM	reset	14-47
Associate a beeping sound with the display of error messages	sound	14-55
Enter any operating-system command or exit to a system shell	system	14-58
Execute commands from a batch file	take	14-59
Delete an alias definition	unalias	14-59
Delete *all* defined aliases	unalias *	14-59
Name additional directories that can be searched when you load source files	use	14-60
Check the current version of the debugger	version	14-61

Managing breakpoints

To do this	Use this command	See page
Add a software breakpoint	ba	14-16
Delete a software breakpoint	bd	14-16
Display a list of all the software breakpoints that are set	bl	14-16
Reset (delete) all software breakpoints	br	14-17

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	14-14
Reopen the CALLS window	calls	14-18
Display assembly language code at a specific address	dasm	14-21
Display a text file in the FILE window	file	14-26
Display a specific C function	func	14-27
Load an object file	load	14-30
Modify disassembly with the patch assembler	patch	14-42
Load only the object-code portion of an object file	reload	14-47
Load only the symbol-table portion of an object file	sload	14-55

Memory mapping

To do this	Use this command	See page
Initialize a block of memory	fill	14-26
Add an address range to the memory map	ma	14-32
Enable or disable memory mapping	map	14-33
Connect a simulated I/O port to an input or output file (simulator only)	mc	14-34
Delete an address range from the memory map	md	14-36
Disconnect a simulated I/O port (simulator only)	mi	14-38
Display a list of the current memory map settings	ml	14-39
Reset the memory map (delete all ranges)	mr	14-40
Save a block of memory to a system file	ms	14-40

Customizing the screen

To do this	Use this command	See page
Change the border style of any window	border	14-17
Change the screen colors, but don't update the screen immediately	color	14-20
Change the command-line prompt	prompt	14-45
Change the screen colors and update the screen immediately	scolor	14-50
Load and use a previously saved custom screen configuration	sconfig	14-51
Save a custom screen configuration	ssave	14-57

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	14-19
Single-step through assembly language or C code, one C statement at a time	cstep	14-21
Run a program up to a certain point	go	14-27
Halt the target system after executing a RUNF command (emulator and EVM only)	halt	14-28
Single-step through assembly language or C code; step over function calls	next	14-41
Reset the target system (emulator only), simulator, or EVM	reset	14-47
Reset the program entry point	restart	14-48
Execute code in a function and return to the function's caller	return	14-48
Run a program	run	14-48
Run a program with benchmarking—count the number of CPU clock cycles consumed by the executing portion of code (emulator, EVM, and simulator only)	runb	14-49
Disconnect the emulator from the target system and run free (emulator and EVM only)	runf	14-49
Single-step through assembly language or C code	step	14-57
Execute commands from a batch file	take	14-59

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 use some of these commands. For this reason and 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 14.4 on page 14-65.

To do this	Use this command	See page
Run a full profiling session	pf	14-43
Run a quick profiling session	pq	14-44
Resume a profiling session	pr	14-44
Add a stopping point	sa	14-49
Delete a stopping point	sd	14-51
List all the stopping points	sl	14-55
Delete all the stopping points	sr	14-56
Save all the profile data to a file	vaa	14-60
Save currently displayed profile data to a file	vac	14-61
Reset the display in the PROFILE window to show all areas and the default set of data	vr	14-61

14.2 How the Menu Selections Correspond to Commands

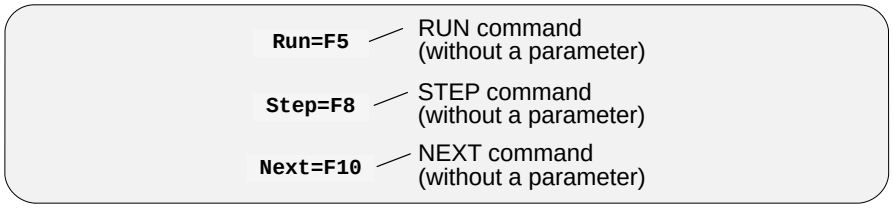
The following sample screens illustrate the relationship of the basic debugger commands to the menu bar and pulldown menus.

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 ESC to close the menu. Refer to Section 5.2, *Using the Menu Bar and the Pulldown Menus,* page 5-7, for more information.

Note:

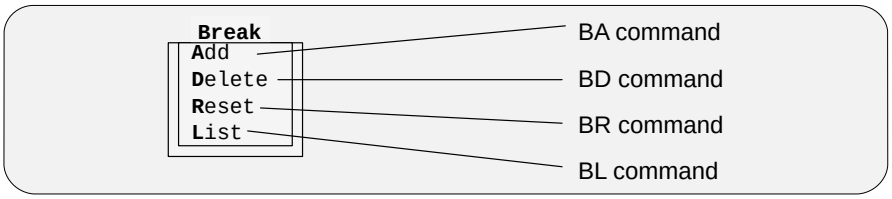
Because the profiling environment supports over 100 profile-specific commands, it's not practical to show the commands associated with the profile menu choices.

Program execution commands

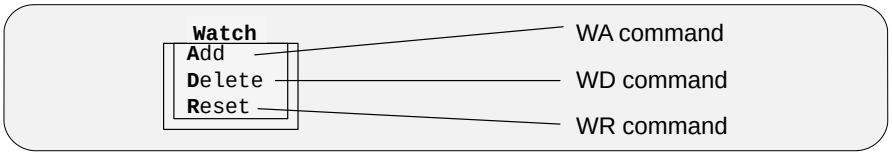


File/Load commands

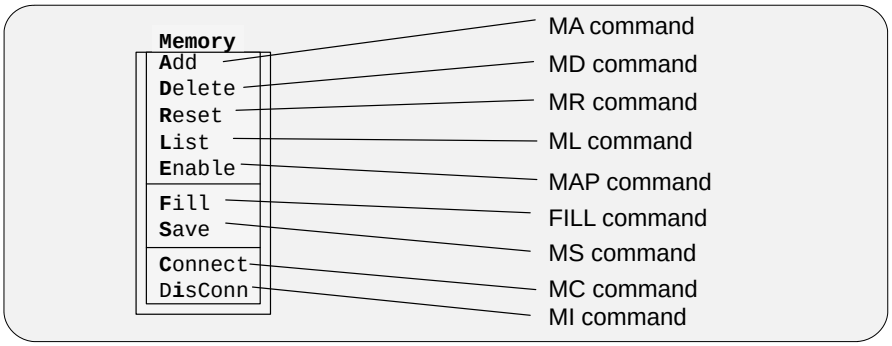
Breakpoint commands



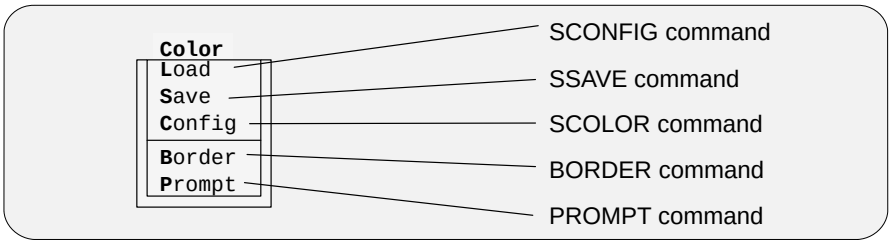
Watch commands



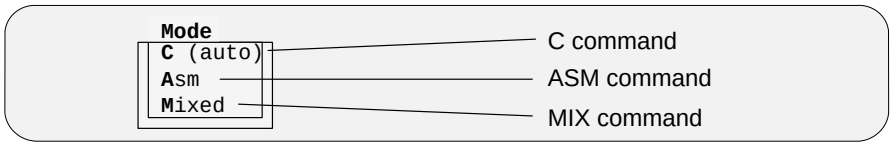
Memory commands



Screen-configuration commands

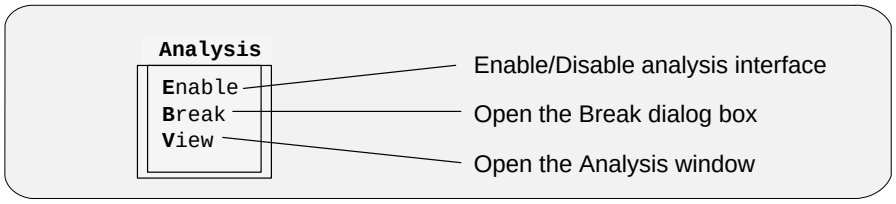


Mode commands



The Analysis menu

The Analysis pulldown menu does *not* correspond to specific debugger commands. Instead, the selections on this menu enable and disable the interface, as well as open dialog boxes that control the interface. Here are the functions of the Analysis menu selections.



14.3 Alphabetical Summary of Debugger and PDM Commands

Most of the commands can be used in the basic debugger environment and/or the profiling environment. Other commands can be used only by the parallel debug manager (PDM). A few commands can be used in two or more 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 ☐ PDM ☑ 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

! — *Use the PDM Command History*

Syntax

!{*prompt number* | *string*}
!!

Menu selection

none

Environments

☐ basic debugger ☑ PDM ☐ profiling

Description

The PDM supports a command history that is similar to the UNIX command history. The PDM prompt identifies the number of the current command. This number is incremented with every command. The PDM command history allows you to reenter any of the last twenty commands.

- The *prompt number* parameter is the number of the PDM prompt that contains the command that you want to reenter.
- The *string* parameter tells the PDM to execute the last command that began with *string.*
- The !! command tells the PDM to execute the last command that you entered.

@ — *Substitute Result of an Expression*

Syntax

@ *variable name* **=** *expression*

Menu selection

none

Environments

☐ basic debugger ☑ PDM ☐ profiling

Description

Unlike the SET command, the @ command first evaluates the *expression*, and then sets the *variable name* to the result. The *expression* can be any expression that uses the symbols described in Section 2.7, page 2-17. The *variable name* can consist of up to 128 alphanumeric characters or underscore characters.

addr — *Display Code at Specified Address*

Syntax

addr *address*[**@data | @prog | @io**]
addr *function name*

Menu selection

none

Environments

☑ basic debugger ☐ PDM ☐ profiling

Description

Use the ADDR command to display C code or 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.

By default, the *address* parameter is treated as a data memory address. However, you can follow it with @prog to identify program memory or with @data to identify data memory. If you are using an emulator or EVM, you can follow *address* with @io to identify I/O space.

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 ☑ PDM ☑ profiling

Description You can use the ALIAS command to define customized command strings for the debugger or for the PDM:

- ❑ The debugger version of the ALIAS command allows you to associate one or more debugger commands with a single *alias name.*
- ❑ The PDM version of the ALIAS command allows you to associate one or more PDM commands with a single *alias name or* associate one or more debugger commands with a single *alias name*.

You can include as many commands in the *command string* as you like, as long you separate them with semicolons and enclose the entire string of commands in quotation marks. You can also identify command parameters by a percent sign followed by a number (%1, %2, etc.). The total number of characters for an individual command (expanded to include parameter values) is limited to 132 (this restriction applies to the debugger version of the ALIAS command only).

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

Description The BD command clears a software breakpoint at a specific *address*. The *address* can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label.

bl — *List Software Breakpoints*

Syntax **bl**

Menu selection **B**reak→**L**ist

Environments ☑ basic debugger ☐ PDM ☐ 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 ☐ PDM ☐ profiling

Description The BORDER command changes the border style of the active window, the inactive windows, and any window that you're resizing. The debugger supports nine border styles. Each parameter for the BORDER command must be one of the numbers that identifies these styles:

Index	Style
0	Double-lined box
1	Single-lined box
2	Solid 1/2-tone top, double-lined sides 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

Note that you can execute the BORDER command as the Border selection on the Color pulldown menu. The debugger displays a dialog box in which 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 ☐ PDM ☐ 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 ☐ PDM ☐ 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 ☐ PDM ☐ profiling

Description The CALLS command displays the CALLS window. The debugger displays this window automatically when you are in auto/C or mixed mode. However, you can close the CALLS window; the CALLS command opens the window again.

cd, chdir — *Change Directory*

Syntax **cd** [*directory name*]
chdir [*directory name*]

Menu selection none

Environments ☑ basic debugger ☐ PDM ☑ profiling

Description The CD or CHDIR command changes the current working directory from within the debugger. You can use relative pathnames as part of the *directory name*. If you don't use a pathname, the CD command displays the name of the current directory. When it is implemented with the USE command, CD can affect any other command whose parameter is a filename, such as the FILE, LOAD, and TAKE commands. You can also use the CD command to change the current drive. For example,

```
cd c:
cd d:\csource
cd c:\c5xhll
```

cls — *Clear Screen*

Syntax **cls**

Menu selection none

Environments ☑ basic debugger ☐ PDM ☑ 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 ☐ PDM ☐ profiling

Description The CNEXT command is similar to the CSTEP command. It runs a program one C statement at a time, updating the display after executing each statement. If you're using CNEXT to step through assembly language code, the debugger won't update the display until it has executed all assembly language statements associated with a single C statement. Unlike CSTEP, CNEXT steps over function calls rather than stepping into them—you don't see the single-step execution of the function call.

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page 7-17, 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 ☐ PDM ☐ 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 ☐ PDM ☐ profiling

Description The CSTEP single-steps through a program one C statement at a time, updating the display after executing each statement. If you're using CSTEP to step through assembly language code, the debugger won't update the display until it has executed all assembly language statements associated with a single C statement.

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

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page 7-17, discusses this in detail).

dasm — *Display Disassembly at Specified Address*

Syntax **dasm** *address*[@**data** | @**prog**]
dasm *function name*

Menu selection none

Environments ☑ basic debugger ☐ PDM ☑ profiling

Description The DASM command displays code beginning at a specific point within the DISASSEMBLY window. The *address* is an absolute address. You can follow the address with @data to identify data memory, or with @prog to identify program memory.

The *function name* can be any C expression, the name of a C function, or the name of an assembly language label.

dir — *List Directory Contents*

Syntax **dir** [*directory name*]

Menu selection none

Environments ☑ basic debugger ☐ PDM ☑ 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*] [, **@data** | **@prog** | **@io**]

Menu selection none

Environments ☑ basic debugger ☐ PDM ☐ profiling

Description The DISP command opens a DISP window to display the contents of an array, structure, or pointer expressions to a scalar type (of the form *pointer). If the *expression* is not one of these types, then DISP acts like a ? command. If the *expression* identifies an address, you can follow it with @data to identify data memory or with @prog to identify program memory . If you are using an emulator or EVM, you can follow an address with @io to identify I/O space.

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	0x0000

You can display the additional data (the data pointed to or the members of the array or structure) in another DISP window by entering 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 Display Window*

Syntax

dlog *filename* [**,**{**a** | **w**}]
or
dlog close

Menu selection

none

Environments

☑ basic debugger ☑ PDM ☑ profiling

Description

The DLOG command allows you to record the information displayed in the COMMAND window or in the PDM display area into a log file.

❑ To begin recording the information shown in the display area of the COMMAND window or in the display area of the PDM, use:

dlog *filename*

Log files can be executed by using the TAKE command. When you use DLOG to record the information from the display area into a log file called *filename*, the debugger (or PDM) 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 **a** (append) option.

echo — *Echo String to Display Area*

Syntax **echo** *string*

Menu selection none

Environments ☑ basic debugger ☑ PDM ☑ profiling

Description The ECHO command displays *string* in the display area of the COMMAND window or in the display area of the PDM. You can't use quotation marks around the string, and any leading blanks in your command string are removed when the ECHO command is executed.

❏ You can execute the debugger version of the ECHO command only in a batch file.

❏ You can execute the PDM version of the ECHO command in a batch file or from the command line.

elif — *Test for Alternate Condition*

Description ELIF provides an alternative test by which you can execute PDM commands in the IF/ELIF/ELSE/ENDIF command sequence. See page 14-29 for more information about these commands.

else — *Execute Alternative Commands*

Description ELSE provides an alternative list of PDM or debugger commands in the IF/ELIF/ELSE/ENDIF or IF/ELSE/ENDIF command sequences, respectively. See pages 14-29 and 14-30 for more information about these commands.

endif — *Terminate Conditional Sequence*

Description

ENDIF identifies the end of a conditional-execution command sequence that begins with an IF command. See pages 14-29 and 14-30 for more information about these commands.

endloop — *Terminate Looping Sequence*

Description

ENDLOOP identifies the end of the LOOP/BREAK/CONTINUE/ENDLOOP and LOOP/ENDLOOP command sequences. See pages 14-31 and 14-32 for more information about these commands.

eval — *Evaluate Expression*

Syntax

eval *expression*[**@data** | **@prog** | **@io**]
e *expression*[**@data** | **@prog** | **@io**]

Menu selection

none

Environments

☑ basic debugger ☐ PDM ☑ profiling

Description

The EVAL command evaluates an expression in the same way that the ? command does, *but EVAL does not show the result* in the COMMAND window display area. EVAL is useful for assigning values to registers or memory locations in a batch file (where it's not necessary to display the result).

If the *expression* identifies an address, you can follow it with @data to identify data memory or with @prog to identify program memory. If you are using an emulator or EVM, you can follow an address with @io to identify I/O space.

eval — *Evaluate Expression and Set to Variable*

Syntax

eval [**–g** {*group* | *processor name*}] *variable name***=***expression*[**,** *format*]

Menu selection

none

Environments

☐ basic debugger ☑ PDM ☐ profiling

Description

The EVAL command evaluates an expression in a debugger and sets a variable to the result of the expression.

- The **–g** option specifies the group or processor that EVAL should be sent to. If you don't use this option, the command is sent to the default group (dgroup).
- When you send the EVAL command to more than one processor, the PDM takes the *variable name* that you supply and appends a suffix for each processor. The suffix consists of the underscore character (_) followed by the name that you assigned to the processor.

- The *expression* can be any expression that uses the symbols described in Section 2.7, page 2-17.
- When you use the optional *format* parameter, the value that the variable is set to will be in one of the following formats:

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

file — *Display Text File*

Syntax **file** *filename*

Menu selection **L**oad→**F**ile

Environments ☑ basic debugger ☐ PDM ☑ 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 intended primarily for displaying C source code. You can view only one text file at a time.

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

fill — *Fill Memory*

Syntax **fill** *address, page, length, data*

Menu selection **M**emory→**F**ill

Environments ☑ basic debugger ☐ PDM ☐ profiling

Description The FILL command fills a block of memory with a specified value. This command has four parameters:

- The *address* parameter identifies the beginning of the block.

- The *page* parameter is a one-digit number that identifies the type of memory (program, data or I/O) to be filled:

To fill this type of memory	Use this value as the *page* parameter
Program memory	**0**
Data memory	**1**
I/O space	**2**

- 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.

func — *Display Function*

Syntax

func *function name*
func *address*

Menu selection none

Environments ☑ basic debugger ☐ PDM ☑ profiling

Description The FUNC command displays a specified C function in the FILE window. You can identify the function by its name or its address. Note that FUNC works the same way that FILE works, but you don't need to identify the name of the file that contains the function.

go — *Run to Specified Address*

Syntax **go** [*address*]

Menu selection none

Environments ☑ basic debugger ☐ PDM ☐ 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* — **Emulator & EVM Only**

Syntax **halt**

Menu selection none

Environments ☑ basic debugger ☐ PDM ☐ 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.

help — *List PDM Commands*

Syntax **help** [*command*]

Menu selection none

Environments ☐ basic debugger ☑ PDM ☐ profiling

Description The HELP command provides a brief description of the requested PDM command. If you omit the *command* parameter, the PDM lists all of the available PDM commands.

history — *List the Last Twenty PDM Commands*

Syntax **history**

Menu selection none

Environments ☐ basic debugger ☑ PDM ☐ profiling

Description The HISTORY command displays the last twenty PDM commands that you've entered.

if/elif/else/endif — *Conditionally Execute PDM Commands*

Syntax

if *expression*
PDM commands
[**elif** *expression*
PDM commands]
[**else**
PDM commands]
endif

Menu selection

none

Environments

☐ basic debugger ☑ PDM ☐ profiling

Description

These commands allow you to conditionally execute PDM commands in a batch file or from the command line.

- If the expression for the IF is nonzero, the PDM executes all commands between the IF and the ELIF, ELSE, or ENDIF.
- The ELIF is optional. If the expression for the ELIF is nonzero, the PDM executes all commands between the ELIF and the ELSE or ENDIF.
- The ELSE is optional. If the expressions for the IF and the ELIF (if present) are false (zero), the PDM executes the commands between the ELSE and the ENDIF.

The IF/ELIF/ELSE/ENDIF commands can be entered interactively or included in a batch file that is executed by the TAKE command. When you enter IF from the PDM command line, a question mark (?) prompts you for the next entry. The PDM continues to prompt you for input using the ? until you enter ENDIF. After you enter ENDIF, the PDM immediately executes the IF command.

If you are in the middle of interactively entering an IF statement and want to abort it, type CONTROL C.

if/else/endif — *Conditionally Execute Debugger Commands*

Syntax

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

Menu selection

none

Environments

☑ basic debugger ☐ PDM ☑ profiling

Description

These commands allow you to conditionally execute debugger commands 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 sequence is optional.

You can substitute a keyword for the expression. Keywords evaluate to true (1) or false (0). You can use the following keywords with the IF command:

- **$$EMU$$** (tests for the emulator version of the debugger)
- **$$SIM$$** (tests for the simulator version of the debugger)
- **$$EVM$$** (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 file.
- You can't nest conditional commands within the same batch file.

load — *Load Executable Object File*

Syntax

load *object filename*

Menu selection

Load→ **L**oad

Environments

☑ basic debugger ☐ PDM ☑ 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 *object filename* with an extension of .out. Note that the LOAD command clears the old symbol table and closes the WATCH and DISP windows.

loop/break/ continue/endloop

Loop Through PDM Commands

Syntax

loop *expression*
*PDM command*s
[**break**]
[**continue**]
endloop

Menu selection

none

Environments

☐ basic debugger ☑ PDM ☑ profiling

Description

The LOOP/BREAK/CONTINUE/ENDLOOP commands allow you to set up a looping situation in a batch file or from the command line. Unlike the debugger version of the LOOP/ENDLOOP commands, the PDM version of the LOOP command evaluates only Boolean expressions:

- If the Boolean expression evaluates to true (1), the PDM executes all commands between the LOOP and the BREAK, CONTINUE, or ENDLOOP.
- If the Boolean expression evaluates to false (0), the loop is not entered.

The optional BREAK command allows you to exit the loop without having to reach the ENDLOOP. This is helpful when you are testing a group of processors and want to exit if an error is detected.

The CONTINUE command, which is also optional, acts as a goto and returns command flow to the enclosing LOOP command. CONTINUE is useful when the part of the loop that follows is complicated and returning to the top of the loop avoids further nesting.

The LOOP/BREAK/CONTINUE/ENDLOOP commands can be entered interactively or included in a batch file that is executed by the TAKE command. When you enter LOOP from the PDM command line, a question mark (?) prompts you for the next entry. The PDM continues to prompt you for input using the ? until you enter ENDLOOP. After you enter ENDLOOP, the PDM immediately executes the LOOP command.

If you are in the middle of interactively entering a LOOP statement and want to abort it, type CONTROL C.

loop/endloop — *Loop Through Debugger Commands*

Syntax

loop *expression*
debugger commands
endloop

Menu selection

none

Environments

☑ basic debugger ☐ PDM ☑ profiling

Description

The LOOP/ENDLOOP commands allow you to set up a looping situation in a batch file. These looping commands evaluate in a way similar to the way that the run conditional command expression evaluates:

- 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 file.
- You can't nest LOOP/ENDLOOP commands within the same file.

ma — *Add Block to Memory Map*

Syntax

ma *address, page, length, type*

Menu selection

Memory→**A**dd

Environments

☑ basic debugger ☐ PDM ☑ profiling

Description

The MA command identifies valid ranges of target memory.

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

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

- The *length* parameter defines the length of the range. This parameter can be any C expression.

- The *type* parameter identifies the read/write characteristics of the memory range. The *type* must be one of these keywords:

To identify this kind of memory,	Use this keyword as the *type* parameter
Read-only memory	**R** or **ROM**
Write-only memory	**W** or **WOM**
Read/write memory	**R|W** or **RAM**
Read/write external memory	**RAM|EX** or **R|W|EX**
No-access memory	**PROTECT**
Input port	**P|R**
Output port	**P|W**
Input/output port	**P|R|W**

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.

map — *Enable Memory Mapping*

Syntax **map** {**on** | **off**}

Menu selection **M**emory→**E**nable

Environments ☑ basic debugger ☐ PDM ☑ 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 Simulated I/O Port or Data Memory to a File* — **Simulator**

Syntax **mc** *port address, page, length, filename, fileaccess*

Menu selection **M**emory→**C**onnect

Environments ☑ basic debugger ☐ PDM ☐ profiling

Description The MC command connects P|R, P|W, or P|R|W 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 space or data memory. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. The *address* must be the starting address of a block.
- The *page* parameter is a one-digit number that identifies the type of memory (data or I/O) that the address occupies:

To identify this page,	Use this value as the *page* parameter
Data memory	**1**
I/O space	**2**

Note:

A file cannot be connected to program memory (page 0) locations.

- The *length* parameter defines the length of the range. This parameter can be any C expression.
- The *filename* parameter can be any filename. If you connect a port or memory location to read from a file, the file must exist, or the MC command will fail.

- The *fileaccess* parameter identifies the access characteristics of the I/O memory and data memory. The file access must be one of the keywords identified below:

To identify this file access type	Use this keyword as the *fileaccess* parameter
Input port (I/O space)	**P|R**
Simulator halt at EOF of input space (I/O space)	**R|P|NR**
Output port (I/O space)	**P|W**
Read-only internal memory	**R**
Read-only external memory	**EX|R**
Simulator halt at EOF of input file for internal memory	**R|NR**
Simulator halt at EOF of input file for external memory	**EX|R|NR**
Write-only internal memory	**W**
Write-only external memory	**EX|W**

  For I/O memory locations, the file is accessed during a read or write instruction to the associated port address. You can connect any I/O port to a file. A maximum of one input and one output file can be connected to a single port; however, multiple ports can be connected to a single file.

  For data memory locations, the debugger accesses the data as follows:

  - When you're executing code:
    - If you have specified a file, the debugger reads the data from the file and updates the memory location with that data.
    - If you have specified a file, the debugger writes the data to the memory location, as well as the file.
  - When you're using the debugger:
    - The debugger reads the data value from the memory location, *not* from the connected file.
    - If you have specified a file, the debugger writes the data to the memory location, as well as the file.

If you use the NR parameter, then the simulator halts execution when it reads an EOF. The debugger displays the appropriate message in the COMMAND window display area:

```
<addr> EOF reached – connected at port(I/O_PAGE)
```

or

```
<addr> EOF reached – connected at location (DATA_PAGE)
```

At this point, you can disconnect the file by using the MI command and attach a new file by using the MC command. If you don't do anything, then the input file rewinds automatically, and execution continues until EOF is read.

If you do not specify NR at EOF, execution does not halt, and you are not notified when EOF is reached. The input file rewinds automatically, and the simulator resumes reading from the file.

md — *Delete Block From Memory Map*

Syntax **md** *address, page*

Menu selection **M**emory→**D**elete

Environments ☑ basic debugger ☐ PDM ☑ profiling

Description The MD command deletes a range of memory from the debugger's memory map.

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

  ```
  Specified map not found
  ```

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

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

Note:

If you want to use the MD command to remove a simulated I/O port, you must first disconnect the port with the MI command.

mem — *Modify MEMORY Window Display*

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

Menu selection none

Environments ☑ basic debugger ☐ PDM ☐ profiling

Description

The MEM command identifies a new starting address for the block of memory displayed in the MEMORY window. The optional extension number (#) opens a pop-up 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.

You can display either data or program memory; or, if you are using an emulator or EVM, you can display I/O space:

- By default, the MEMORY window displays data memory. Although it is not necessary, you can explicitly specify data memory by following the *expression* parameter with a suffix of **@data**.
- You can display the contents of program memory by following the *expression* parameter with a suffix of **@prog**. When you do this, the MEMORY window's label changes to MEMORY [PROG] so that there is no confusion about the type of memory being displayed.
- Using an emulator or EVM, you can display the contents of the I/O space by following the *expression* parameter with a suffix of **@io**. When you do this, the MEMORY window's label changes to MEMORY [IO] so that there is no confusion about the type of memory being displayed.

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

Parameter	Result	Parameter	Result
*****	Default for the data type	**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

mi — *Disconnect I/O Port or Data Memory* — **Simulator Only**

Syntax **mi** *portaddress, page, fileaccess*

Menu selection **M**emory→D**i**sConn

Environments ☑ basic debugger ☐ PDM ☐ profiling

Description The MI command disconnects a simulated I/O port or data memory location from its associated system file.

- The *portaddress* parameter identifies the address of the I/O port or data memory, which must have been previously defined with the MC command.
- The *page* parameter is a one-digit number that identifies the type of memory (data or I/O) that the port occupies:

To identify this page,	Use this value as the *page* parameter
Data memory	**1**
I/O space	**2**

The page parameter for the MI command must match the page parameter that was used with the MC command to connect the port.

- The *fileaccess* parameter identifies the access characteristics of the memory/IO space range. The fileaccess must be one of these keywords:

To identify this file access type	Use this keyword as the *fileaccess* parameter
Input port (I/O space)	**P|R**
Simulator halt at EOF of input space (I/O space)	**R|P|NR**
Output port (I/O space)	**P|W**
Read-only internal memory	**R**
Read-only external memory	**EX|R**
Simulator halt at EOF of input file for internal memory	**R|NR**
Simulator halt at EOF of input file for external memory	**EX|R|NR**
Write-only internal memory	**W**
Write-only external memory	**EX|W**

The fileaccess parameter for the MI command must match the parameter that was used with the MC command to connect the port.

mix — *Enter Mixed Mode*

Syntax **mix**

Menu selection Mo**D**e→**M**ixed

Environments ☑ basic debugger ☐ PDM ☐ profiling

Description The MIX command changes 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 ☐ PDM ☑ 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 ☐ PDM ☑ profiling

Description The MOVE command moves the active window to the specified XY position. If you choose to, 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 window 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 ☐ PDM ☑ 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, page, length, filename*

Menu selection **M**emory→**S**ave

Environments ☑ basic debugger ☐ PDM ☑ 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 first address in the block.
- The *page* parameter is a one-digit number that identifies the type of memory (program, data, or I/O) to be saved:

To save this type of memory	Use this value as the *page* parameter
Program memory	**0**
Data memory	**1**
I/O space	**2**

- The *length* parameter defines the length, in words, of the block. This parameter can be any C expression.
- The *filename* parameter 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 ☐ PDM ☐ 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 (*Running code conditionally*, page 7-17, discusses this in detail).

patch — *Patch Assemble*

Syntax **patch** *address, assembly language instruction*

Menu selection none

Environments ☑ basic debugger ☐ PDM ☐ 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*. Notice that patch assembly is available on all debugger platforms except DOS.

pause — *Pause Execution*

Syntax **pause**

Menu selection none

Environments ☑ basic debugger ☑ PDM ☐ profiling

Description The PAUSE command allows you to pause the debugger or PDM while running a batch file or executing a flow control command. Pausing is especially helpful in debugging the commands in a batch file.

When the debugger or PDM reads this command in a batch file or during a flow control command segment, the debugger/PDM stops execution and displays the following message:

```
<< pause – type return >>
```

To continue processing, press ↵.

pesc — *Send ESC Key to Debuggers*

Syntax **pesc** [**–g** {*group* | *processor name*}]

Menu selection none

Environments ☐ basic debugger ☑ PDM ☐ profiling

Description The PESC command sends the ESC key to an individual debugger or to a group of debuggers. This PESC command halts program execution, but the processors don't halt at the same real time. When halting a group of processors, the individual processors are halted in the order in which they were added to the group.

The **–g** option identifies the group or processor that the command should be sent to. If you don't use this option, the ESC key is sent to the default group (dgroup).

pf — *Profile, Full*

Syntax **pf** *starting point* [, *update rate*]

Menu selection **P**rofile→**F**ull

Environments ☐ basic debugger ☐ PDM ☑ 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, and the spinning-wheel character is not displayed.

phalt — *Halt Processors in Parallel*

Syntax **phalt** [{**–g** *group* | *processor name*}]

Menu selection none

Environments ☐ basic debugger ☑ PDM ☐ profiling

Description The PHALT command halts one or more processors. If you send a PRUN or PRUNF command to a group or to an individual processor, you can use PHALT to halt the group or the individual processor. Each processor in a group is halted at the same real time. If you don't use the **–g** option to specify a group or a processor name, the PHALT command will be sent to the default group (dgroup).

pq — *Profile, Quick*

Syntax **pq** *starting point* [, *update rate*]

Menu selection **P**rofile→**Q**uick

Environments ☐ basic debugger ☐ PDM ☑ 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 (see page 14-43).

pr — *Resume Profiling Session*

Syntax **pr** [*clear data* [, *update rate*]]

Menu selection **P**rofile→**R**esume

Environments ☐ basic debugger ☐ PDM ☑ profiling

Description The PR command resumes the last profiling session (initiated by PF or PQ), starting from the current program counter. 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.

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 (see page 14-43).

prompt — *Change Command-Line Prompt*

Syntax **prompt** *new prompt*

Menu selection **C**olor→**P**rompt

Environments ☑ basic debugger ☐ PDM ☑ 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).

prun — *Run Code in Parallel*

Syntax **prun** [**–r**] [**–g** {*group* | *processor name*}]

Menu selection none

Environments ☐ basic debugger ☑ PDM ☐ profiling

Description The PRUN command is the basic command for running an entire program. You enter the command from the PDM command line to begin execution at the same real time for an individual processor or a group of processors.

The **–g** option identifies the group or processor that the command should be sent to. If you don't use this option, then code will run on the default group (dgroup). You can use the PHALT command to stop a global run.

The **–r** (return) option for the PRUN command determines when control returns to the PDM command line:

- **Without –r**, control is not returned to the command line until each debugger in the group finishes running code. If you want to to break out of a synchronous command and regain control of the PDM command line, press CONTROL C in the PDM window. This will return control to the PDM command line. However, no debugger executing the command will be interrupted.
- **With –r**, control is returned to the command line immediately, even if a debugger is still executing a command. You can type new commands, but the processors can't execute the commands until they finish with the current command; however, you can perform PHALT, PESC, and STAT commands when the processors are still executing.

prunf — *Run Free in Parallel*

Syntax **prunf** [**–g** {*group* | *processor name*}]

Menu selection none

Environments ☐ basic debugger ☑ PDM ☐ profiling

Description The PRUNF command starts the processors running free, which means they are disconnected from the emulator. RUNF synchronizes the debuggers to cause the processors to begin execution at the same real time. The **–g** option identifies the group or processor that the command should be sent to. If you don't use this option, then code will run on the default group (dgroup).

The PHALT command stops a PRUNF; note that the debugger automatically executes a PHALT when the debugger is invoked.

pstep — *Single-Step in Parallel*

Syntax **pstep** [**–g** {*group* | *processor name*}] [*count*]

Menu selection none

Environments ☐ basic debugger ☑ PDM ☐ profiling

Description The PSTEP command single-steps synchronously through assembly language code with interrupts disabled. RUNF synchronizes the debuggers to cause the processors to begin execution at the same real time. The **–g** option identifies the group or processor that the command should be sent to. If you don't use this option, then code will run on the default group (dgroup). You can use the PHALT command to stop a global run.

You can use the *count* parameter to specify the number of statements that you want to single-step.

> **Note:**
>
> If the current statement that a processor is pointing to has a breakpoint, that processor will not step synchronously with the other processors when you use the PSTEP command. However, that processor will still single-step.

quit — *Exit Debugger*

Syntax **quit**

Menu selection none

Environments ☑ basic debugger ☑ PDM ☑ profiling

Description The QUIT command exits the debugger and returns to the operating system. If you enter this command from the PDM, the PDM and all debuggers running under the PDM are exited.

reload — *Reload Object Code*

Syntax **reload** [*object filename*]

Menu selection **L**oad→**R**eload

Environments ☑ basic debugger ☐ PDM ☑ 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 ☐ PDM ☑ 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.

If you are using the simulator and execute the RESET command, the simulator simulates the ’C5xx processor and peripheral reset operation, putting the processor in a known state.

restart — *Reset PC to Program Entry Point*

Syntax
restart
rest

Menu selection
Load→R**E**start

Environments
☑ basic debugger ☐ PDM ☑ 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 ☐ PDM ☐ 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 ☐ PDM ☐ profiling

Description
The RUN command is the basic command for running an entire program. The command's behavior depends on the type of *expression* 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*, the run becomes conditional (conditional runs are described in detail on page 7-17).
- 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 ☐ PDM ☐ 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 7.7, *Benchmarking*, on page 7-19.

runf — *Run Free* — **Emulator & EVM Only**

Syntax **runf**

Menu selection none

Environments ☑ basic debugger ☐ PDM ☐ 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 Stoppoint*

Syntax **sa** *address*

Menu selection **S**top-points→**A**dd

Environments ☐ basic debugger ☐ PDM ☑ 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 ☐ PDM ☐ 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 areas of the display that are affected. The *attributes* identify how the area is affected. The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Valid values for the *attribute* parameters include:

black	blue	green	cyan
red	magenta	yellow	white
bright		blink	

Valid values for the *area name* parameters include:

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

You don't have to type an entire *attribute* or *area name*; you need 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 in which 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 ☐ PDM ☑ 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 Stoppoint*

Syntax **sd** *address*

Menu selection **S**top-points→**D**elete

Environments ☐ basic debugger ☐ PDM ☑ profiling

Description The SD command deletes the stopping point at *address*.

send — *Send Debugger Command to Individual Debugger(s)*

Syntax **send** [**–r**] [**–g** {*group* | *processor name*}] *debugger command*

Menu selection none

Environments ☐ basic debugger ☑ PDM ☐ profiling

Description The SEND command sends any debugger command to an individual debugger or to a group of debuggers. If the command produces a message, it will be displayed in the COMMAND window for the appropriate debugger(s) and also in the PDM window.

- The **–g** option specifies the group or processor that the debugger command should be sent to. If you don't use this option, the command is sent to the default group (dgroup).
- The **–r** (return) option determines when control returns to the PDM command line:
  - **Without –r**, control is not returned to the command line until each debugger in the group finishes running code. Any results that would be printed in the COMMAND window of the individual debuggers will also be echoed in the PDM command window. These results will be displayed by processor.

If you want to break out of a synchronous command and regain control of the PDM command line, press CONTROL C in the PDM window. This will return control to the PDM command line. However, no debugger executing the command will be interrupted.

- **With –r**, control is returned to the command line immediately, even if a debugger is still executing a command. When you use –r, you *do not* see the results of the commands that the debuggers are executing.

set — *Set a Variable to a String*

Syntax

set [*group name* [**=** *list of processor names*]]
set [*variable* [**=** *string value*]]

Menu selection

none

Environments

☐ basic debugger ☑ PDM ☐ profiling

Description

The SET command allows you to create groups of processors to which you can send commands. With the SET command you can:

- **Define a group of processors.** It is useful to define a group when you plan to send commands to the same set of processors. The commands are sent to the processors in the same order in which you added the processors to the group. To define a group, specify a *group name* and then list the processors you want in the group.
- **Set the default group.** Defining a default group provides you with a shorthand method of maintaining members in a group or of sending commands to the same group. To set up the default group, use the SET command with a special *group name* called dgroup.
- **Modify an existing group or create a group based on another group.** Once you've created a group, you can add processors to it by using the SET command and preceding the existing *group name* with a dollar sign ($) in the list of processors. You can also use a group as part of another group by preceding the existing group's name with a dollar sign. The dollar sign tells the PDM to use the processors listed previously in the group as part of the new list of processors.
- **List all groups of processors.** You can use the SET command without any parameters to list all of the processors that belong to a group, in the order in which they were added to the group.

You can also use the SET command with system-defined variables to:

- **Change the prompt for the PDM.** To change the PDM prompt, use the SET command with the system variable called prompt. For example, to change the PDM prompt to 3PROCs, enter:

  ```
  set prompt = 3PROCs ↵
  ```

- **Check the execution status of the processors.** In addition to displaying the execution status of a processor or group of processors, the STAT command (described on page 14-57) sets a system variable called status. If *all* of the processors in the specified group are running, the status variable is set to 1. If one or more of the processors in the group is halted, the status variable is set to 0.

  You can use this variable when you want an instruction loop to execute until a processor halts (the LOOP/ENDLOOP command is described on page 14-32):

- **Create your own system variables.** You can use the SET command to create your own system variables that you can use with PDM commands. For more information about creating your own system variables, see page 2-18.

setf — *Set Default Data-Display Format*

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

Menu selection none

Environments ☑ basic debugger ☐ PDM ☐ profiling

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

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

char	short	uint	ulong	double
uchar	int	long	float	ptr

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

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

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

```
setf * ↵
```

size — *Size Active Window*

Syntax **size** [*width*, *length*]

Menu selection none

Environments ☑ basic debugger ☐ PDM ☑ 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 size 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 4-24.

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 Stoppoints*

Syntax **sl**

Menu selection **S**top-points→**L**ist

Environments ☐ basic debugger ☐ PDM ☑ 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 ☐ PDM ☑ 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 Beeping*

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

Menu selection none

Environments ☑ basic debugger ☐ PDM ☐ 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.

spawn — *Invoke a 'C5xx Debugger*

Syntax **spawn emu5xx –n** *processor name* [*invocation options*]

Menu selection none

Environments ☐ basic debugger ☑ PDM ☐ profiling

Description You must invoke a debugger for each processor that you want the PDM to control. To invoke a debugger, use the SPAWN command.

- **emu5xx** is the executable command that invokes the debugger. The PDM associates the *processor name* with the actual processor according to which executable file you use. In order to invoke a debugger, the PDM must be able to find the executable file for that debugger. The PDM will first search the current directory and then search the directories listed with the PATH statement.

- **–n** *processor name* supplies a processor name. You *must* use the –n option since the PDM uses processor names to identify the various debuggers that are running. The processor name can consist of up to eight alphanumeric characters or underscore characters and must begin with an alphabetic character. Note that the name is not case sensitive. The processor name must match one of the names defined in your board configuration file (see Appendix C).

sr — *Reset Stoppoints*

Syntax **sr**

Menu selection **S**top-points→**R**eset

Environments ☐ basic debugger ☐ PDM ☑ 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 ☐ PDM ☐ 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.

stat — *Find the Execution Status of Processors*

Syntax **stat** [{**–g** *group* | *processor name*}]

Menu selection none

Environments ☐ basic debugger ☑ PDM ☐ profiling

Description The STAT command tells you whether a processor is running or halted. If a processor is halted when you execute this command, then the PDM also lists the current PC value for that processor. If you don't use the **–g** option, the PDM displays the status of the processors in the default group (dgroup).

step — *Single-Step*

Syntax **step** [*expression*]

Menu selection Step=**F8** (in disassembly)

Environments ☑ basic debugger ☐ PDM ☐ 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 (*Running code conditionally*, page 7-17, discusses this in detail).

system *Enter Operating-System Command*

Syntax

Basic debugger: **system** [*DOS command* [, *flag*]]
PDM: **system** *operating-system command*

Menu selection

none

Environments

☑ basic debugger ☑ PDM ☑ profiling

Description

You can use the SYSTEM command to enter operating-system commands:

- The debugger version of 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.)

- The PDM version of the SYSTEM command allows you to enter a single operating-system command without explicitly exiting the PDM environment. You cannot enter more than one operating-system command with the PDM version of the SYSTEM command.

take — *Execute Batch File*

Syntax

Basic debugger: **take** *batch filename* [, *suppress echo flag*]
PDM: **take** *batch filename*

Menu selection

none

Environments

☑ basic debugger ☑ PDM ☑ profiling

Description

The TAKE command tells the debugger or the PDM to read and execute commands from a batch file. The *batch filename* parameter identifies the file that contains commands. If you don't supply a pathname as part of the filename, the PDM first looks in the current directory and then searches directories named with the D_DIR environment variable.

The *batch filename* for the PDM version of this command must have a .pdm extension, or the PDM will not be able to read the file. In addition, the batch file that the PDM reads can contain only PDM commands.

By default, the debugger echoes the commands to the output area of the COMMAND window and updates the display as it reads the commands from the batch file. For the debugger, you can change this behavior:

- ❏ 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 ☑ PDM ☑ 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.

unset — *Delete Group*

Syntax

unset *group name*
unset ***

Menu selection

none

Environments

☐ basic debugger ☑ PDM ☐ profiling

Description

The UNSET command deletes a group of processors. You can use this command in conjunction with the SET command to remove a particular processor from a group.

To delete all groups, enter an asterisk instead of a group name:

```
unset *
```

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

> **Note:**
>
> When you use UNSET * to delete all of your system variables and processor groups, variables such as prompt, status, and dgroup are also deleted.

use — *Use New Directory*

Syntax

use *directory name*

Menu selection

none

Environments

☑ basic debugger ☐ PDM ☑ 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

View→S**a**ve→**A**ll views

Environments

☐ basic debugger ☐ PDM ☑ profiling

Description

The VAA command saves all statistics collected during the current profiling session. The data is stored in a system file.

vac — *Save Displayed Profile Data to a File*

Syntax **vac** *filename*

Menu selection **V**iew→S**a**ve→**C**urrent view

Environments ☐ basic debugger ☐ PDM ☑ 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 Current Debugger Version*

Syntax **version**

Menu selection none

Environments ☑ basic debugger ☐ PDM ☑ profiling

Description The VERSION command displays the debugger's copyright date and the current version number of the debugger, silicon, XDS, and EVM.

vr — *Reset PROFILE Window Display*

Syntax **vr**

Menu selection **V**iew→**R**eset

Environments ☐ basic debugger ☐ PDM ☑ 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 default sort order.

wa — *Add Item to WATCH Window*

Syntax **wa** *expression* [**@prog** | **@data**][,[*label*] [*, display format*]]

Menu selection **W**atch→**A**dd

Environments ☑ basic debugger ☐ PDM ☐ 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. If the *expression* identifies an address, you can follow it with @prog to identify program memory or with @data to identify data memory. If you are using an emulator or EVM, you can follow an address with @io to identify I/O space. Without the suffix, the debugger treats an address expression as a program-memory location.

WA is most useful for watching an expression whose value changes over time; constant expressions 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 ☐ PDM ☐ 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 ☐ PDM ☐ 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 ☐ PDM ☑ 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 ☐ PDM ☐ 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 ☐ PDM ☑ 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.

14.4 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 14–1. 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 14–2. 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 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 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 14–3. 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 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 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 14–4. 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 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 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 14–5. 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 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 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**

14.5 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

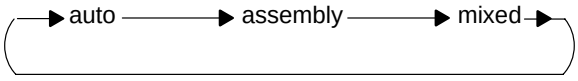
Editing text on the command line

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	DEL
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: 	F3

Halting or escaping from an action

The escape key acts as an end or undo key in several situations.

To do this	Use this function key
Halt program execution	ESC
Close a pulldown menu	
Undo an edit of the active field in a data-display window (pressing this key leaves the field unchanged)	
Halt the display of a long list of data in the COMMAND window display area	

Displaying pulldown menus

To do this	Use these function keys
Display the Load menu	ALT L
Display the Break menu	ALT B
Display the Watch menu	ALT W
Display the Memory menu	ALT M
Display the Color menu	ALT C
Display the MoDe menu	ALT D
Display the Analysis menu	ALT A
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 (which is equivalent to the RUN command without an *expression* parameter)	F5
Single-step code from the current PC (which is equivalent to the STEP command without an *expression* parameter)	F8
Single-step code from the current PC; step over function calls (which is 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 or DISP 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 descriptions/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 eight characters at a time	←
❑ *Other windows:* move the field cursor left one 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 eight characters at a time	→
❑ *Other windows:* move the field cursor right one field; at the last field on a line, wrap back 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 this function key
FILE or DISASSEMBLY window: set or clear a breakpoint	F9
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	

Chapter 15

Basic Information About C Expressions

Many of the debugger commands take C expressions as parameters. This allows the debugger to have a relatively small, yet powerful, instruction set. Because C expressions can have side effects—that is, the evaluation of some types of expressions can affect existing values—you can use the same command to display or to change a value. This reduces the number of commands in the command set.

This chapter contains basic information that you'll need to know in order to use C expressions as debugger command parameters.

Topic **Page**

15.1 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.

15.2 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.

Restrictions

The following restrictions apply to the debugger's expression analysis features.

- The sizeof operator is not supported.
- The comma operator (,) is not supported (commas are used to separate parameter values for the debugger commands).
- Function calls and string constants are currently not supported in expressions.
- The debugger supports a limited capability of type casts; the following forms are allowed:

  (*basic type* **)**
  (*basic type* ***** *...***)**
  ([*structure/union/enum***]** *structure/union/enum tag* **)**
  ([*structure/union/enum***]** *structure/union/enum tag* ***** *...* **)**

  Note that you can use up to six *****s in a cast.

Additional features

- All floating-point operations are performed in double precision using standard widening. (This is transparent.) Floats are represented in IEEE floating-point format.
- All registers can be referenced by name. The TMS320C5xx's auxiliary registers are treated as integers and/or pointers.
- Void expressions are legal (treated like integers).
- The specification of variables and functions can be qualified with context information. Local variables (including local statics) can be referenced with the expression format:

  *function name***.***local name*

This expression format is useful for examining the automatic variables of a function that is not currently being executed. Unless the variable is static, however, the function must be somewhere in the current call stack. Note that if you want to see local variables from the currently executing function, you need not use this form; you can simply specify the variable name (just as in your C source).

File-scoped variables (such as statics or functions) can be referenced with the following expression format:

filename.function name
or *filename.variable name*

This expression format is useful for accessing a file-scoped static variable (or function) that may share its name with variables in other files.

Note that in this expression, *filename* **does not include** the file extension; the debugger searches the object symbol table for any source filename that matches the input name, disregarding any extension. Thus, if the variable ABC is in file source.c, you can specify it as source.ABC.

Note that these expression formats 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
*AR5
*(AR2 + 123)
*(I*J)
```

  By default, the values are treated as integers (that is, these expressions point to integer values).

- Any expression can be typecast to a pointer to a specific type (overriding the default of pointing to an integer, as described above).

  Hint: You can use casting with the WA and DISP commands to display data in a desired format.

  For example, the expression:

```
*(float *)10
```

  treats 10 as a pointer to a floating-point value at location 10 in memory. In this case, the debugger fetches the contents of memory location 10 and treats the contents as a floating-point value. If you use this expression as a parameter for the DISP command, the debugger displays memory contents as an array of floating-point values within the DISP window, beginning with memory location 10 as array member [0].

Note how the first expression differs from the expression:

```
(float)*10
```

In this case, the debugger fetches an integer from address 10 and converts the integer to a floating-point value.

You can also typecast to user-defined types such as structures. For example, in the expression:

```
((struct STR *)10)–>field
```

the debugger treats memory location 10 as a pointer to a structure of type STR (assuming that a structure is at address 10) and accesses a field from that structure.

Appendix A

Customizing the Emulator Analysis Interface

The interface to the ’C5xx emulator analysis module is register based. In most cases, the Analysis break events dialog box provides a sufficient means of setting hardware breakpoints. In some cases, however, you may want to define more complex conditions for the processor to detect. Or, you may want to write a batch file that defines breakpoint conditions. In either case, you can accomplish these tasks by accessing the analysis registers through the debugger. This appendix explains how to access these registers.

Topic **Page**

A.1 Summary of Aliased Commands

A basic set of analysis commands is defined in the analysis.cmd file supplied in your ’C5xx debugger package. These commands, like the analysis dialog boxes, load the analysis registers with your specified values. You must TAKE the analysis.cmd file before you can use any of these commands. To do this, enter:

```
take analysis.cmd
```

By default, the debugger echoes the file to the output area of the COMMAND window. However, you can view the entire file by using the FILE command to display its contents in the FILE window. Table A–1 shows the predefined commands along with their menu equivalents.

These aliased commands, created in the analysis.cmd file, are provided to help you familiarize yourself with the analysis registers and how they work. These aliases are simply a starting point for you to build upon to create your own commands.

Table A–1. The Analysis Commands Found in the analysis.cmd File

Command	Menu → Dialog Box	Description	Page
asys_emu0out	Emulator pins → EMU0 trigger out	Set EMU0 pin to output	A-5
asys_emu1out	Emulator pins → EMU0 trigger out	Set EMU1 pin to output	A-5
asys_extcnt	Emulator pins → External clock	Use the external counter on the emulator	A-5
asys_off	Analysis → Enable/Disable	Turn off the analysis interface	A-4
asys_on	Analysis → Disable/Enable	Turn on the analysis interface	A-4
asys_reset	None	Reset the analysis interface	A-9
cnt_br	Count → Branch taken	Count any branches detected	A-6
cnt_call	Count → Call taken	Count any calls detected	A-6
cnt_clock	Count → CPU clock	Count CPU clock cycles	A-6
cnt_data	Count → Data bus	Count any data accesses	A-6
cnt_ins	Count → Instruction fetch	Count any instructions fetched	A-6
cnt_intr	Count → Interrupt/trap taken	Count any interrupts/traps detected	A-6

Table A–1 The Analysis Commands Found in the analysis.cmd File (Continued)

Command	Menu → Dialog Box	Description	Page
cnt_load *value*	Count → Event counter	Load the analysis counter	A-5
cnt_pclk	Count → Pipeline count	Count CPU pipeline execution clocks	A-6
cnt_prog1 cnt_prog2	Count → Program bus	Count any program address accesses	A-6
cnt_ret	Count → Return taken	Count any returns from an interrupt/trap detected	A-6
data_brk_add *address* or *symbol name*	Break → Data bus: Address field	Set a breakpoint on a data address	A-7
data_brk_msk *value*	Break → Data bus	Set a breakpoint on data value with mask	A-7
data_brk_val *value*	Break → Data bus	Set a breakpoint on data value	A-7
data_qual_r	Count/Break → Data bus: Read	Data read qualifier	A-8
data_qual_rw	Count/Break → Data bus: Access	Data read/write qualifier	A-8
data_qual_w	Count/Break → Data bus: Write	Data write qualifier	A-8
prog1_brk_add *address or function name*	Break → Program bus 1: Address field	Set a breakpoint on a program bus 1 address	A-7
prog2_brk_add *address or function name*	Break → Program bus 2: Address field	Set a breakpoint on a program bus 2 address	A-7
prog1_qual_iaq prog2_qual_iaq	Break → Program bus: Fetch	Program instruction acquisition	A-8
prog1_qual_r prog2_qual_r	Count/Break → Program bus: Read	Program read qualifier	A-8
prog1_qual_rw prog2_qual_rw	Count/Break → Program bus: Access	Program read/write qualifier	A-9
prog1_qual_w prog2_qual_w	Count/Break → Program bus: Write	Program write qualifier	A-9
prog_win_on	Count/Break → Program window	Program windowing enabled	A-5
prog_win_off	Count/Break → Program window	Program windowing disabled	A-5
stop_br	Break → Branch taken	Halt the processor when a branch is detected	A-7
stop_call	Break → Call taken	Halt the processor when a call is detected	A-7

Table A–1 The Analysis Commands Found in the analysis.cmd File (Continued)

Command	Menu → Dialog Box	Description	Page
stop_cnt	Count → Break when < 0	Halt the processor when the counter passes 0	A-5
stop_data	Break → Data bus	Halt the processor on a data bus access	A-7
stop_disc	Break → Discontinuity	Halt any discontinuity	A-7
stop_emu0	Break → EMU0 driven low	Halt the processor when the EMU0 pin is low	A-7
stop_emu1	Break → EMU1 driven low	Halt the processor when the EMU1 pin is low	A-7
stop_intr	Break → Interrupt/trap taken	Halt the processor when an interrupt/trap is detected	A-8
stop_off	None	Disable break events	A-8
stop_pclk	Break → Pipeline clock	Halt the processor on a pipe-clock (instruction fetched)	A-8
stop_prog1 stop_prog2	Break → Program bus	Halt the processor on a program bus access	A-7
stop_ret	Break → Return taken	Halt the processor when a return from an interrupt/trap is detected	A-8

In addition to these predefined commands, you can create your own by using the ALIAS and EVAL commands. Refer to Sections 5.5, page 5-21, and 8.2, page 8-2, for more information on ALIAS and EVAL. The following subsections briefly describe the use of the analysis commands.

Enabling the analysis interface

Enabling the analysis interface is simply a matter of typing in a command. The basic syntax for this command is:

asys_on

The syntax for the command to disable the analysis interface is:

asys_off

Enabling the program window

Enabling the analysis interface is simply a matter of typing in a command. The basic syntax for this command is:

prog_win_on

The syntax for the command to disable the analysis interface is:

prog_win_off

Enabling the EMU0/1 pins

To set the EMU0/1 pins to output or to use the external counter, enter the appropriate command:

To do this...	Enter this...
Set the EMU0 pin to output	**asys_emu0out**
Set the EMU1 pin to output	**asys_emu1out**
Use the external counter on the emulator	**asys_extcnt**

Enabling event counting

The syntax for the command to load or reset the event counter is:

cnt_load *value*

Load *value* with a 1s complement of the number of times you want to count the specified event. For example, to stop the processor after ten instruction fetches have occurred, enter:

```
cnt_load –10        Set the counter to count ten events and then stop.
cnt_ins                                  Count instruction fetches.
stop_cnt          Stop the processor when the counter reaches 0.
```

In this example, you must load the counter with a negative value because the event counter register represents a 1s complement of the loaded value.

To reset the internal event counter and count the number of instruction fetches detected, enter:

```
cnt_load 0                                          Reset the counter.
cnt_ins          Count the number of instruction fetches detected.
```

You can count only one event at a time. To count any of the other events, simply type in the appropriate command. Table A–2 shows the command for counting each of the nine events.

Table A–2. The Analysis Commands

Command	Menu Selection	Description
cnt_br	Branch taken	Count the number of branches detected
cnt_call	Call taken	Count the number of calls detected
cnt_clock	CPU clock	Count the number of CPU clock cycles
cnt_data	Data bus	Count the number of data cycles
cnt_ins	Instruction fetch	Count the number of instruction fetches
cnt_intr	Interrupt/trap taken	Count the number of interrupts/traps detected
cnt_pclk	Pipeline clock	Count the CPU pipeline execution clocks
cnt_prog1 cnt_prog2	Program bus	Count the number of program address accesses
cnt_ret	Return taken	Count the number of returns from interrupts, traps, or subroutine calls

Setting breakpoints on a single program or data address

The simplest events to detect identify a single address. To define this type of event, follow the command with a C expression. For example, to set a program address breakpoint, enter:

asys_on	*Turn the analysis interface on.*
prog1_brk_add *main* **prog2_brk_add *main***	*Set a program address breakpoint on function_name.*
`stop_prog`	*Enable the processor to stop on the breakpoint condition.*
`run`	*Run the program.*
prog1_brk_add *My_Function* **prog2_brk_add *My_Function***	*Set a new program address breakpoint on function_name2.*
`run`	*Run to the new breakpoint.*

The commands shown in bold represent the actual breakpoint commands used. *Main* and *My_Function* represent the addresses on which the processor will break. These function names can be replaced by specific address locations. Table A–3 shows the breakpoint commands for setting single address breakpoints; their respective menu selections can be found in the Analysis break events dialog box. You can set breakpoints on any combination of these events.

Table A–3. Breakpoint Commands for Program and Data Addresses

Command	Dialog Box Selection	Description
data_brk_add *address* or *symbol name*	Data bus	Set a data breakpoint address
data_brk_val *value*	Data bus	Set a data value
data_brk_msk *value*	Data bus	Set a data value mask
prog1_brk_add *address* or *function name*	Program bus 1	Set a program breakpoint address
prog2_brk_add *address* or *function name*	Program bus 2	Set a program breakpoint address
stop_data	Data bus	Stop the processor when the data breakpoint condition executes
stop_prog1 stop_prog2	Program bus	Stop the processor when the program breakpoint condition executes

Breaking on event occurrences

You can also set conditions on various types of processor operations. To define these conditions or events, simply enter the command. For example, to stop the processor when it detects an interrupt or a call taken, enter:

```
asys_on          Turn the analysis interface on.
stop_intr        Enable the processor to stop when it detects an interrupt.
stop_call        Enable the processor to stop when it detects a call taken.
```

Table A–4 shows the commands for stopping the processor when an event occurs. You can set breakpoints on any combination of these events.

Table A–4. Breakpoint Commands for Event Occurrences

Command	Menu Selection	Description
stop_br	Branch taken	Stop the processor when a branch is taken
stop_call	Call taken	Stop the processor when a call is taken
stop_disc	Discontinuity	Stop the processor with any discontinuity
stop_emu0	EMU0 driven low	Stop the processor when the EMU pin reaches a logic low of 0
stop_emu1	EMU1 driven low	Stop the processor when the EMU pin reaches a logic low of 1

Table A–4. Breakpoint Commands for Event Occurrences (Continued)

Command	Menu Selection	Description
stop_intr	Interrupt/trap taken	Stop the processor when an interrupt is detected
stop_off	None	Disable break events
stop_pclk	Pipeline clock	Stop the processor on a pipeline clock (instruction fetched)
stop_ret	Return taken	Stop the processor when a return from an interrupt, branch, or call occurs

Qualifying on a read or a write

Data and program accesses can be qualified, depending on whether the memory cycle is a read or write:

`go` *`function_name`*	*Run to the beginning of the function function_name.*
`data_qual_w`	*Look only at writes.*
`data_brk_add` *`data_symbol`*	*Set a data address breakpoint on data_symbol.*
`cnt_data`	*Enable the processor to count any writes to the specified data access.*
`run`	*Count the number of any writes to data_symbol.*

This example sets a data address breakpoint that counts only when a write is detected. Table A–5 shows the qualifier commands for data and program break events. You can use only one of these commands at a time.

Table A–5. Read and Write Qualifying Commands for Data and Program Accesses

Command	Menu Selection	Description
data_qual_r	Data bus: Read	Look only at data reads
data_qual_rw	Data bus: Access	Look at both data reads and writes
data_qual_w	Data bus: Write	Look only at data writes
prog1_qual_iaq prog2_qual_iaq	Program bus: Fetch	Program instruction acquisition
prog1_qual_r prog2_qual_r	Program bus: Read	Look only at data reads

Table A–5. Read and Write Qualifying Commands for Data and Program Accesses (Continued)

Command	Menu Selection	Description
prog1_qual_rw prog2_qual_rw	Program bus: Access	Look at both data reads and writes
prog1_qual_w prog2_qual_w	Program bus: Write	Look only at data writes

Resetting the analysis interface

Whenever you begin a new analysis session, you may want to define new parameters or qualifier expressions. You can do this without having to manually deselect each defined condition. Just enter the ASYS_RESET command. To reset the analysis interface, type:

```
asys_reset ↵
```

Note:

To clear conditions or qualifier expressions previously defined via the Analysis menu, you must open the *Analysis count events* and *Analysis break events* dialog boxes and deselect each defined condition.

A.2 Using the Analysis Registers

By manipulating the analysis registers, you can customize commands for more complex instructions that do not exist on the Break or Count dialog boxes. Use the alias and evaluate commands to create your own commands. The basic syntax for creating customized analysis commands is:

alias *command_name*, ”**eval** *register name* = *code*”

For example, to create a new command for turning on the analysis interface, enter:

```
alias analysis_on, ”eval anaenbl = 1”
```

To create a new command for counting branches detected, enter:

```
alias cb, ”e evtselt = 12”
```

To create your own analysis commands, you must familiarize yourself with the thirteen analysis registers and how they work. The following subsections discuss the analysis registers briefly. (The registers are in alphabetical order.)

anaenbl (enable analysis)

You can enable and disable the analysis module by using the anaenbl register. Set the bit to 1 to enable or to 0 to disable.

Bit Number	Description
0	enable analysis module
1	reserved (set to 0)
2	reserved (set to 0)
3	enable external counter
4	enable EMU0 output
5	enable EMU1 output

When you disable analysis, all registers except anaenbl retain their previous state.

anastat (analysis status)

The anastat register records the occurrence of enabled events. The status bits are defined below:

Bit Number	Definition
0	call taken
1	return from interrupt/trap/subroutine
2	interrupt/trap taken
3	branch taken
4	pipe clock
5	program 1 address
6	data address
7	discontinuity
8	event counter passed 0
9	EMU0 detected low
10	EMU1 detected low
11	program 2 address

Run commands will not interfere with the status bits because they are cleared before command execution.

datbrkp (data breakpoint address)

You can specify a breakpoint address for each of the major buses in the 'C5xx path. When a valid bus cycle occurs and the bus value matches the breakpoint address, then a breakpoint condition can occur.

datdval (data breakpoint data value)

You can specify a data value that is qualified by the data breakpoint address. The debugger halts the processor when the entered data bus address contains the entered data value.

datmval (data breakpoint mask value)

You can mask bits of the data value. If the mask value is 0, then any value in the given data address will be trapped, providing the access type matches. If the mask value is 0xffff, then the mask value is not used; when the data address value matches the specified data value, the data breakpoint is trapped. Otherwise, the mask value and the data value will be used to continue checking for the desired data pattern according to the following algorithm:

```
if ( NOT( (data_bus XOR data_val) AND mask_val) )
  {
   data_break_point_found = TRUE ;
  }
```

datqual (data breakpoint qualifier)

The data breakpoint register has three qualifier bits. The qualifier definitions are shown below.

Qualifier Code	Definition
0	read
1	write
2	reserved
3	read/write

evtcntr (event counter)

This register represents a true value in the ’C5xx analysis module, which provides a 16-bit decrementing event counter. For convenience, the pseudo-register, EVT_cntr, provides a 1s complement of the evtcntr value.

You can use the event counter in one of two ways:

- Count the number of events detected.
- Stop after *n* events have occurred.

To count the number of events detected, load the counter with its maximum value –1, or 0xFFFF. The following example loads the counter and counts the instructions.

```
cnt_load 0          Reset the counter.
cnt_ins             Count the number of instruction fetches detected.
```

The EVT_cntr register will display the number of events detected after reaching a stop condition.

To stop after a certain number of events, load the counter with the number of events you want to occur before setting a breakpoint. The following example counts ten events and then stops.

```
cnt_load –10        Set the counter to count ten events and then stop.
cnt_ins             Count instruction fetches.
stop_cnt            Stop the processor when the counter reaches 0.
```

If a software breakpoint happens to halt the processor before the counter reaches zero, then the CNT_valu (displayed in the WATCH window) will contain the number of events remaining.

Note:

When CPU clock cycles are counted, the event counter includes start-up and latency cycles.

evtselt (select the event for counting)

The ’C5xx can count nine types of events; however, only one event can be counted at a time. The count select codes are defined below.

Select Code	Definition
0	CPU clocks
1	pipeline clocks
2–7	not used
8	instruction fetched
9	call taken
10	return from interrupt/trap/subroutine
11	interrupt/trap taken
12	branch taken
13	program 1 address breakpoints
14	program 2 address breakpoints
15	data address breakpoints

hbpenbl (select hardware breakpoints)

By setting the appropriate enable bit to 1 in the hbpenbl register, the ’C5xx can break on multiple events. Setting the bit to 0 disables the breakpoint and clears the register. The breakpoint enable bits are defined below.

Bit Number	Definition
0	call taken
1	return from interrupt/trap/subroutine
2	interrupt/trap taken
3	branch taken
4	pipe clock
5	program 1 address
6	data address
7	discontinuity
8	counter passing 0
9	EMU0 detected low
10	EMU1 detected low
11	program 2 address

pgabrkp1, pgabrkp2 (program address breakpoint)

You can specify a breakpoint address for each of the major buses in the 'C5xx path. When a valid bus cycle occurs and the bus value matches the breakpoint address, then a breakpoint condition can occur.

pgaqual1, pgaqual2 (program breakpoint qualifier)

The data breakpoint register has three qualifier bits. The qualifier definitions are shown below.

Qualifier Code	Definition
0	read
1	write
2	program instruction acquisition
3	read/write

progwin (program window enable)

You can enable data breakpoints in the program window between program address 1 and program address2. Set the bit to 1 to enable or to 0 to disable.

ptrace0/ptrace1/ptrace2 (discontinuity trace samples 0–2)

A program discontinuity occurs when the program addresses fetched by the processor become nonsequential as a result of branches, interrupts, and similar events. The ’C5xx provides three levels of discontinuity trace to aide in program flow analysis:

Register	Name	Description
ptrace0	discontinuity trace sample 0	traces the current code segment
ptrace1	discontinuity trace sample 1	traces the previous code segment
ptrace2	discontinuity trace sample 2	traces the oldest code segment

Example A–1. Program Discontinuity

Address		Code		Comment
010A		nop		
010B		nop		
010C		b	dcon2	*Discontinuity occurs.*
010E		nop		*branch from*
010F		nop		
0110		nop		
0111	dcon2:	nop		*branch to*
0112		nop		
0113		nop		
0114		nop		
0115		b	dcon1	*Discontinuity occurs.*
0117		nop		*branch from*
0118		nop		
0119		nop		
011A	dcon1:	nop		*branch to*
011B		nop		
011C		nop		

Stepping through the code starting at address 0x010A with ptrace0/1/2 initialized to 0, the trace buffer will show the following:

- Following the first branch (b dcon2), the trace contains the following values:

PTRACE2	0x0000	
PTRACE1	0x010F	*last instruction fetched before the branch*
PTRACE0	0x0111	*branch to address (dcon2)*

- Following the second branch (b dcon1), the trace contains the following values:

PTRACE2	0x0111	*discontinuity from the oldest code segment*
PTRACE1	0x011E	*discontinuity from the previous code segment*
PTRACE0	0x011A	*current code segment*

Appendix B

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. Note that the PDM executes the first step. (For more information on the environment variables mentioned below, refer to the appropriate installation guide.)

1) Establishes the connection between the processor name that you provide and the actual processor.
2) Reads options from the command line.
3) Reads any information specified with the D_OPTIONS environment variable.
4) Reads information from the D_DIR and D_SRC environment variables.
5) Looks for the init.clr screen configuration file in directories named with D_DIR.
6) Initializes the debugger screen and windows but initially displays only the COMMAND window.
7) 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 have not used the –t option, the debugger looks for the default initialization batch file. The batch file name differs for each version of the debugger:
      - For the emulator, this file is named *emuinit.cmd.*
      - For the EVM, this file is named *evminit.cmd.*
      - For the simulator, this file is named *siminit.cmd.*

      If the debugger finds the file corresponding to your tool, it reads and executes the file.

c) If the debugger does not find the –t option or the initialization batch file, it looks for a file called *init.cmd*. This allows you to have one initialization batch file for more than one debugger tool. To set up this file, you can use the IF/ELSE/ENDIF commands (see page 5-18 for more information) to indicate which memory map applies to each tool.

8) Loads any object filenames specified with D_OPTIONS or specified on the command line during invocation.

9) 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 C

Describing Your Target System to the Debugger

In order for the debugger to understand how you have configured your target system, you must supply a file for the debugger to read.

- If you're using an emulation scan path that contains only one 'C5xx and no other devices, you can use the *board.dat* file that comes with the 'C5xx emulator kit. This file describes to the debugger the single 'C5xx in the scan path and gives the 'C5xx the name CPU_A. Since the debugger automatically looks for a file called board.dat in the current directory and in the directories specified with the D_DIR environment variable, you can skip this appendix.
- If you plan to use a different target system, you must follow these steps:

  Step 1: Create the board configuration text file.

  Step 2: Translate the board configuration text file to a binary, structured format so that the debugger can read it.

  Step 3: Specify the formatted configuration file when invoking the debugger.

  These steps are described in this appendix.

Topic **Page**

C.1 Step 1: Create the Board Configuration Text File

To describe the emulation scan path of your target system to the debugger, you must create a board configuration file. The file consists of a series of entries, each describing one device on your scan path. For the debugger to work, you must list, in order, the individual devices on your system in the board configuration file. The text version of the configuration file will be referred to as *board.cfg* in this book.

Example C–1 shows a board.cfg file that describes a possible 'C5xx device chain. It lists six octals named A1–A6, followed by five 'C5xx devices named CPU_A, CPU_B, CPU_C, CPU_D, and CPU_E.

Example C–1. A Sample 'C5xx Device Chain

(a) A sample board.cfg file

Device Name	Device Type	Comments
"A1"	BYPASS08	;the first device nearest TDO ;(test data out)
"A2"	BYPASS08	;the next device nearest TDO
"A3"	BYPASS08	
"A4"	BYPASS08	
"A5"	BYPASS08	
"A6"	BYPASS08	
"CPU_A"	TI320C5xx	;the first 'C5xx
"CPU_B"	TI320C5xx	
"CPU_C"	TI320C5xx	
"CPU_D"	TI320C5xx	
"CPU_E"	TI320C5xx	;the last 'C5xx nearest TDI ;(test data in)

(b) A sample 'C5xx device chain

The order in which you list each device is important. The emulator scans the devices, assuming that the data from one device is followed by the data of the next device on the chain. Data from the device that is closest to the emulation header's TDO (test data out) reaches the emulator first. Moreover, in the board.cfg file, the devices should be listed in the order in which their data reaches the emulator. For example, the device whose data reaches the emulator first is listed first in the board.cfg file; the device whose data reaches the emulator last is listed last in the board.cfg file.

The board.cfg file can have any number of each of the three types of entries:

- **Debugger devices** such as the 'C5xx. These are the only devices that the debugger can recognize.
- The **TI ACT8997 scan path linker**, or **SPL**. The SPL allows you to have up to four secondary scan paths that can each contain debugger devices ('C5xxs) and other devices.
- **Other devices**. These are any other devices in the scan path. These devices cannot be debugged and must be worked around or bypassed when trying to access the 'C5xxs.

Each entry in the board.cfg file consists of at least two pieces of data:

- **The name of the device.** The device name always appears first and is enclosed in double quotes:

  "device name"

  This is the same name that you use with the –n debugger option, which tells the debugger the name of the 'C5xx. The *device name* can consist of up to eight alphanumeric characters or underscore characters and must begin with an alphabetic character.

- **The type of the device.** The debugger supports the following device types:

  - **TI320C5xx** is an example of a debugger-device type. TI320C5xx describes the 'C5xx. TI320C4x describes the 'C4x.
  - **SPL** specifies the scan path linker and must be followed by four subpaths, as in this syntax:

    "*device name*" **SPL** {*subpath0*} {*subpath1*} {*subpath2*} {*subpath3*}

    Each *subpath* can contain any number of devices. However, an SPL subpath *cannot* contain another SPL.

Example C–2 shows a file that contains an SPL.

Example C–2. A board.cfg File Containing an SPL

Device Name	Device Type	Comments
″A1″	BYPASS08	;the first device nearest TDO
″A2″	BYPASS08	
″CPU_A″	TI320C5xx	;the first ’C5xx
″HUB″	SPL	;the scan path linker
{		;the first subpath
″B1″	BYPASS08	
″B2″	BYPASS08	
″CPU_B″	TI320C5xx	;the second ’C5xx
}		
{		;the second subpath
″C1″	BYPASS08	
″C2″	BYPASS08	
″CPU_C″	TI320C5xx	;the third ’C5xx
}		
{		;the third subpath (contains nothing)
}		
{		;the fourth subpath
″D1″	BYPASS08	
″D2″	BYPASS08	
″CPU_D″	TI320C5xx	;the fourth ’C5xx
}		
″CPU_E″	TI320C5xx	;the last ’C5xx nearest TDI

Note: The indentation in the file is for readability only.

C.2 Step 2: Translate the Configuration File to a Debugger-Readable Format

After you have created the board.cfg file, you must translate it from text to a binary, conditioned format so that the debugger can understand it. To translate the file, use the composer utility that is included with the emulator kit. At the system prompt, enter the following command:

composer [*input file* [*output file*]]

- The *input file* is the name of the board.cfg file that you created in step 1; if the file isn't in the current directory, you must supply the entire pathname. If you omit the input filename, the composer utility looks for a file called board.cfg in your current directory.
- The *output file* is the name that you can specify for the resulting binary file; ideally, use the name board.dat. If you want the output file to reside in a directory other than the current directory, you must supply the entire pathname. If you omit an output filename, the composer utility creates a file called board.dat and places it in the current directory.

To avoid confusion, use a .cfg extension for your text filenames and a .dat extension for your binary filenames. If you enter only one filename on the command line, the composer utility assumes that it is an input filename.

C.3 Step 3: Specify the Configuration File When Invoking the Debugger

When you invoke a debugger (either from the PDM or at the system prompt), the debugger must be able to find the board.dat file so that it knows how you have set up your scan path. The debugger looks for the board.dat file in the current directory and in the directories named with the D_DIR environment variable.

If you used a name other than board.dat or if the board.dat file is not in the current directory or in a directory named with D_DIR, you must use the –f option when you invoke the debugger. The –f option allows you to specify a board configuration file (and pathname) that will be used instead of board.dat. The format for this option is:

–f *filename*

Appendix D

Debugger and PDM Messages

This appendix contains an alphabetical listing of the progress and error messages that the debugger might display in the display area of the COMMAND window. Each message contains both a description of the situation that causes the message and an action to take if the message indicates a problem or error.

Topic **Page**

D.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.

D.2 Alphabetical Summary of Debugger Messages

Symbols

‘]’ expected

Description This is an expression error—it means that the parameter contained an opening bracket symbol but didn’t contain a closing bracket symbol.

Action See Section D.4, page D-28.

‘)’ expected

Description This is an expression error—it means that the parameter contained an opening parenthesis symbol but didn’t contain a closing parenthesis symbol.

Action See Section D.4, page D-28.

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 at a location 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 reenter 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 breakpoints are set in your program. Use the BR command to delete all 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.

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, reenter the CD command and specify the entire pathname for that directory (for example, specify C:\c5xxhll, not just c5xxhll).

Cannot connect file to MMR (0h–01fh) area

Description An attempt has been made to connect a file to a data memory location between 0x0 and 0x1f .

Action You cannot connect a file to any location between 0x0 and 0x1f of data memory using the MC command.

Cannot connect file to program memory

Description An attempt has been made to connect a file to program memory.

Action You cannot connect a file to any location in program memory using the MC command.

CANNOT DETECT TARGET POWER

Description This hardware error occurs after resetting the emurst command.

Action Follow the steps described below and then restart your emulator.

- ❑ Check the emulator board to be sure it is installed snugly.
- ❑ Check the cable connecting your emulator and target system to be sure it is not loose.
- ❑ Check your target board to be sure it is getting the correct voltage.
- ❑ Check your emulator scan path to be sure it is uninterrupted.
- ❑ Ensure that your port address is set correctly:
  - ■ Check to be sure that the –p option used with the D_OPTIONS environment variable matches the I/O address defined by your switch settings. (Refer to the *TMS320C5xx Emulator Installation Guide* for more information.)

- Check to see if you have a conflict in address space with another bus setting. If you have a conflict, change the switches on your board to one of the alternate settings listed in the installation guide. Modify the –p option of the D_OPTIONS environment variable to reflect the change in your switch settings.

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 be updated automatically.

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 the autoexec or initdb.bat file; then invoke the debugger again.

CANNOT INITIALIZE TARGET SYSTEM

Description This error occurs while you are invoking the debugger with the emulator. Any combination of events may cause this error to occur.

Action

- Check the cable connecting the emulator to the target system to be sure it is not loose.
- Ensure that your port address is set correctly:
  - Check to be sure that the –p option used with the D_OPTIONS environment variable matches the I/O address defined by your switch settings.

- Check to see if you have a conflict in address space with another bus setting. If you have a conflict, change the switches on your board to one of the alternate settings listed in the installation guide. Modify the –p option of the D_OPTIONS environment variable to reflect the change in your switch settings.

❑ Check the end of your autoexec.bat or initdb.bat file for the emurst.exe command. Execute this command *after* powering up the target board.

For more details, refer to the *TMS320C5xx Emulator Installation Guide.*

CANNOT INITIALIZE THE EVM

Description This error occurs while you are invoking the debugger with the EVM. Any combination of events may cause this error to occur.

Action

❑ Check the EVM board to be sure it is installed snugly.

❑ Ensure that your port address is set correctly:

- Check to be sure the –p option used with the D_OPTIONS environment variable matches the I/O address defined by your switch settings.
- Check to see if you have a conflict in address space with another bus setting. If you have a conflict, change the switches on your board to one of the alternate settings listed in the installation guide. Modify the –p option of the D_OPTIONS environment variable to reflect the change in your switch settings.

For more details, refer to the *TMS320C5xx Evaluation Module Installation Guide.*

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 Be sure that you are connecting to or disconnecting from an address that is mapped in as an input, output, or I/O port.

Cannot open config file

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, reenter the command with the correct name. If it was, reenter 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 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 open object file: "*filename*"

Description The file specified with the LOAD, SLOAD, or RELOAD command is not an object file that the debugger can load.

Action Be sure that you're loading an actual object file. Be sure that the file was linked (you may want to run dspcl again to create an executable object file).

Cannot read processor status

Description This is a fatal error—for some reason, pressing ESC didn't halt program execution.

Action Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger again. If you are using the emulator, check the cable connections, also.

Cannot reset the processor

Description This is a fatal error—for some reason, pressing ESC didn't halt program execution.

Action Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger again. If you are using the emulator, there may be a problem with the target system; check the cable connections.

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 D.5, page D-28.

Cannot step

Description There is a problem with the target system.

Action See Section D.5, page D-28.

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 D.4, page D-28.

Command "*command*" not found

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

Action Reenter the correct command. Refer to Chapter 14 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 D.5, page D-28.

Conflicting map range

Description A block of memory specified with the MA command overlaps an existing memory map entry. Blocks cannot overlap.

Action Use the ML command to list the existing memory map; this will help you find the existing block that the new block would overlap. If the existing block is not necessary, delete it with the MD command and reenter the MA command. If the existing block is necessary, reenter the MA command with parameters that will not overlap the existing block.

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 because 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.

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 *TMS320C5xx Emulator Installation Guide*.

EOF reached –connected at port(DATA_PAGE): <*memory addr*>

Description The last data of the input file has been read.

Action You can disconnect the file with the MI command and connect a new file with the MC command. If you do not do anything and resume execution, then the input file automatically rewinds, and input data is read from the beginning of the file.

EOF reached –connected at port(IO_PAGE): <*port addr*>

Description The last data of the input file has been read.

Action You can disconnect the file with the MI command and connect a new file with the MC command. If you do not do anything and resume execution, then the input file automatically rewinds, and input data is read from the beginning of the file.

Error in expression

Description This is an expression error.

Action See Section D.4, page D-28.

Execution error

Description There is a problem with the target system.

Action See Section D.5, page D-28.

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 does not exist

Description The port file could not be opened for reading.

Action Be sure that the file exists as named. If it does, enter the USE command to identify the file's directory.

Files must be disconnected from ports

Description You attempted to delete a memory map that has files connected to it.

Action You must disconnect a port with the MI command before you can delete it from the memory map.

File not found

Description The filename specified for the FILE command was not found in the current directory or any of the directories identified with D_SRC.

Action Be sure that the filename was typed correctly. If it was, reenter the FILE command and specify full path information with the filename.

File not found : "*filename*"

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, reenter the command and specify full path information with the filename.

File range is overlapped

Description Files cannot be connected to overlapping data memory or I/O memory ranges.

Action Specify addresses which do not overlap when connecting files to memory location ranges.

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 dsplnk 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 D.4, page D-28.

Function required

Description The parameter for the FUNC command must be the name of a function in the program that is loaded.

Action Reenter the FUNC command with a valid function name.

I

Illegal addressing mode

Description An illegal addressing mode was encountered.

Action Correct your programming to use a valid addressing mode.

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 D.4, page D-28.

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 left-hand side of an assignment expression doesn't meet C language assignment rules.

Action See Section D.4, page D-28.

Illegal memory access

Description Your program tried to access unmapped memory.

Action Modify your source code.

Illegal opcode

Description An invalid 'C5xx 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 D.4, page D-28.

Illegal pointer math

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

Action See Section D.4, page D-28.

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 D.4, page D-28.

Illegal port access(IO_PAGE) at: <*port address*>

Description An attempt was made during simultator execution to read or write to an I/O page location which is not defined.

Action Define the I/O page location with the MA command.

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 D.4, page D-28.

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 D.4, page D-28.

Illegal use of void expression

Description This is an expression error—the expression parameter does not meet the C language rules.

Action See Section D.4, page D-28.

Integer not allowed

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

Action See Section D.4, page D-28.

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 Reenter the command with valid parameters. Refer to the appropriate command description in Chapter 14.

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 Reenter the COLOR or SCOLOR command with a valid area name parameter. Valid area names are listed in Table 10–2, page 10-3.

Invalid color name

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 Reenter the COLOR or SCOLOR command with a valid color parameter. Valid color attributes are listed in Table 10–1, page 10-2.

Invalid memory attribute

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 Reenter the MA command. Use one of the following valid parameters to identify the memory type:

R, EX|R	(read-only memory)
W, EX|W	(write-only memory)
RW, EX|R|W	(read/write memory)
PROTECT	(no-access memory)
P|R	(I/O memory)
P|R|W	(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 cl500 again to create an executable object file). If the file you attempted to load was a valid executable object file, then it was probably corrupted; recompile, assemble, and link with dspcl.

Invalid register

Description This is an internal error.

Action Shutdown the debugger and restart it. If the problem recurs, call the hotline.

Invalid watch delete

Description The debugger can't use the WD command to delete the parameter supplied. Usually, this is because the watch index doesn't exist or because a symbol name was typed in instead of a watch index.

Action Reenter 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 is 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 a 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 D.5, page D-28.

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 D.5, page D-28.

Lval required

Description This is an expression error—an assignment expression was entered that requires a legal left-hand side.

Action See Section D.4, page D-28.

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 D.5, page D-28.

Memory access outside valid range: *address*

Description The debugger attempted to access unmapped memory or memory that overlaps multiple ranges, or the debugger tried to perform an action on a memory range that doesn't have the attributes required for the action.

Action Modify your memory map.

Memory location is not configured

Description An attempt was made to connect a file to a location that has not been configured.

Action Use the MA command to configure the memory location first.

Memory map table full

Description Too many blocks have been added to the memory map. This will rarely happen 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, reenter 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 handle more than 100 pending interrupts.

Action Ensure that the value of the ST register enables this interrupt level during program execution. If it does, disconnect the file that contains the error, modify the file, then use the PINC command to reconnect the file. Next, use the debugger to reset the value of the CLK pseudoregister (if necessary) and restart your program.

No breakpoint at address

Description This is an internal error.

Action Shut down the debugger and restart it. If the problem recurs, call the hotline.

No file is connected at port: <*port address*>

Description An attempt was made during simulator execution to read from or write to an I/O page location with no file connected.

Action Connect an appropriate input or output file to the I/O page location.

Non-repeatable instruction

Description The instruction following the RPT instruction is not a repeatable instruction.

Action Modify your code.

P

Pointer not allowed

Description This is an expression error.

Action See Section D.4, page D-28.

R

Read not allowed for port

Description An attempt was made to connect a file for input operation to an address which is not configured for read.

Action Remap the port or correct the access in your source code.

Read not allowed for port: <*port address*>

Description An attempt has been made to read from an output port of the I/O page.

Action Remap the port or 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 D.5, page D-28.

S

Specified map not found

Description The MD command was entered with an address or block that is not in the memory map.

Action Use the ML command to verify the current memory map. When using MD, it is possible to specify only the first address of a defined block.

Structure member name required

Description This is an expression error—a symbol name followed by a period but no member name.

Action See Section D.4, page D-28.

Structure member not found

Description This is an expression error—an expression references a non-existent structure member.

Action See Section D.4, page D-28.

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 D.4, page D-28.

T

Take file stack too deep

Description Batch files can be nested up to ten 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 ten levels.

Action Edit the batch file that caused the error. Instead of calling another batch file from within the offending file, you may want to copy the contents of the second file into the first. This will remove a level of nesting.

Too few instruction words in RPTB

Description The length of the repeat block was less than three instruction words.

Action Modify your source 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 breakpoints are set in your program. Use the BR command to delete all breakpoints or use the BD command to delete individual software breakpoints.

Too many memory maps for simulator

Description You attempted to add a new memory map range that causes the total number of map ranges to exceed the maximum number that the simulator can support (20 ranges).

Action Check your memory map and reorganize it if necessary.

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 connect to or disconnect from an address that isn't declared as a port.

User halt

Description The debugger halted program execution because you pressed the ESC key.

Action None required; this is normal debugger behavior.

W

Window not found

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

Action Reenter 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 Remap the port or correct the access in your source code.

Write not allowed for port: <*port address*>

Description An attempt has been made to write to an input port of the I/O page.

Action Remap the port or correct the access in your source code.

D.3 Alphabetical Summary of PDM Messages

This section contains an alphabetical listing of the error messages that the PDM might display. Each message contains both a description of the situation that causes the message and an action to be taken.

Note:

If errors are detected in a TAKE file, the PDM aborts the batch file execution, and the file line number of the invalid command is displayed along with the error message.

C

Cannot communicate with *"name"*

Description The PDM cannot communicate with the named debugger, because the debugger either crashed or was exited.

Action Spawn the debugger again.

Cannot communicate with the child debugger

Description This error occurs when you are spawning a debugger. The PDM was able to find the debugger executable file, but the debugger could not be invoked for some reason, and the communication between the debugger and the PDM was never established. This usually occurs when you have a problem with your target system.

Action Exit the PDM and go back though the installation instructions in the installation guide. Reinvoke the PDM and try to spawn the debugger again.

Cannot create mailbox

Description The PDM was unable to create a mailbox for the new debugger that you were trying to spawn; the PDM must be able to create a mailbox in order to communicate with each debugger. This message usually indicates a resource limitation (you have more debuggers invoked than your system can handle).

Action If you have numerous debuggers invoked and you're not using all of them, close some. If you are under a UNIX environment, use the ipcs command to check your message queues; use ipcrm to clean up the message queues.

Cannot open log file

Description The PDM cannot find the filename that you supplied when you entered the DLOG command.

Action

- ❑ Be sure that the file resides in the current directory or in one of the directories specified by the D_DIR environment variable.
- ❑ Check to see if you mistyped the filename.

Cannot open take file

Description The PDM cannot find the batch filename supplied for the TAKE command. You will also see this message if you try to execute a batch file that does not have a .pdm extension.

Action

- ❑ Be sure that the file resides in the current directory or in one of the directories specified by the D_DIR environment variable.
- ❑ Check to see if you mistyped the filename.
- ❑ Be sure that the batch filename has a .pdm extension.
- ❑ Be sure that the file has executable rights.

Cannot open temporary file

Description The PDM is unable create a temporary file in the current directory.

Action Change the permissions of the current directory.

Cannot seek in file

Description While the PDM was reading a file, the file was deleted or modified.

Action Be sure that files the PDM reads are not deleted or modified during the read.

Cannot spawn child debugger

Description The PDM couldn't spawn the debugger that you specified, because the PDM couldn't find the debugger executable file (emu5xx). The PDM will first search for the file in the current directory and then search the directories listed with the PATH statement.

Action Check to see if the executable file is in the current directory or in a directory that is specified by the PATH statement. Modify the PATH statement if necessary, or change the current directory.

Command error

Description The syntax for the command that you entered was invalid (for example, you used the wrong options or arguments).

Action Reenter the command with valid parameters.

D

Debugger spawn limit reached

Description The PDM spawned the maximum number of debuggers that it can keep track of in its internal tables. The maximum number of debuggers that the PDM can track is 2,048. However, your system may not have enough resources to support that many debuggers.

Action Before trying to spawn an additional debugger, close any debuggers that you don't need to run.

I

Illegal flow control

Description One of the flow control commands (IF/ELIF/ELSE/ENDIF or LOOP/BREAK/CONTINUE/ENDLOOP) has an error. This error usually occurs when there is some type of imbalance in one of these commands.

Action Check the flow command construct for such problems as an IF without an ENDIF, a LOOP without an ENDLOOP, or a BREAK that does not appear between a LOOP and an ENDLOOP. Edit the batch file that contains the problem flow command, or interactively reenter the correct command.

Input buffer overflow

Description The PDM is trying to execute or manipulate an alias or shell variable that has been recursively defined.

Action Use the SET and/or ALIAS commands to check the definitions of your aliases and system variables. Modify them as necessary.

Invalid command

Description The command that you entered was not valid.

Action Refer to the command summary in Chapter 14, *Summary of Commands and Special Keys,* for a complete list of commands and their syntax.

Invalid expression

Description The expression that you used with a flow control command or the @ command is invalid. You may see specific messages before this one that provide more information about the problem with the expression. The most common problem is the failure to use the $ character when evaluating the contents of a system variable.

Action Check the expression that you used. Refer to Section 2.7, page 2-17, for more information about expression analysis.

Invalid shell variable name

Description The system variable name that you used the SET command to assign is invalid. Variable names can contain any alphanumeric characters or underscore characters.

Action Use a different name.

M

Maximum loop depth exceeded

Description The LOOP/ENDLOOP command that you tried to execute had more than 10 nested LOOP/ENDLOOP constructs. LOOP/ENDLOOP constructs can be nested up to ten deep.

Action Edit the batch file that contains the LOOP/ENDLOOP construct, or reenter the LOOP/ENDLOOP command interactively.

Maximum take file depth exceeded

Description The batch file that you tried to execute with the TAKE command called or nested more than ten other batch files. The TAKE command can handle only batch files that are nested up to ten deep.

Action Edit the batch file.

U

Unknown processor name *"name"*

Description The processor name that you specified with the –g option or a processor name within a group that you specified with the –g option does not match any of the names of the debuggers that were spawned under the PDM.

Action Be sure that you've correctly entered the processor name.

D.4 Additional Instructions for Expression Errors

Whenever you receive an expression error, you should reenter the command and edit the expression so that it follows the C language expression rules. If necessary, refer to a C language manual such as *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

D.5 Additional Instructions for Hardware Errors

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

- If a bus fault occurs, the emulator may not be able to access memory.
- The ’C5xx must be reset before you can use the emulator. Most target systems reset the ’C5xx at power-up; your target system may not be doing this.

Appendix E

Glossary

A

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

aggregate type: A C data type such as a structure or array in which a variable is composed of multiple variables, called members.

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

ANSI C: A version of the C programming language that conforms to the C standards defined by the American National Standards Institute.

assembly mode: A debugging mode that shows assembly language code in the DISASSEMBLY window and doesn't show the FILE window, no matter what type of code is currently running.

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.

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

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.

break event: An event that causes the processor to halt.

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.

cl500: A shell utility that invokes the TMS320C5xx 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 TMS320C5xx 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 'C5xx 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 'C5xx programs running on a 'C5xx 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.

discontinuity: A state in which the addresses fetched by the debugger become nonsequential as a result of instructions that load the PC with new values, such as branches, calls, and returns.

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 that provides direct control over the 'C5xx 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.

event: An operation performed in hardware such as branches, calls, and return instructions.

EVM: *Evaluation Module.* A development tool that lets you execute and debug applications programs by using the 'C5xx 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 primarily intended 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

masking: Ignoring the last 16 bits of a specified address. The processor detects all reads from and writes to addresses containing the first 16 bits of the specified address. This allows you to follow the progress of events occurring on a range of addresses.

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.

O

open-collector output: An output circuit that actively drives logic levels both high and low.

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.

PDM: *Parallel Debug Manager.* A program used for creating and controlling multiple debuggers for the purpose of debugging code in a parallel-processing environment.

point: To move the mouse cursor until it overlays the desired object on the screen.

port address: The PC I/O memory space that the debugger uses for communicating with the emulator 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.

R

ripple-carry output signal: An output signal from a counter indicating that the counter has reached its maximum value.

S

scalar type: A C type in which the variable is a single variable, not composed of other variables.

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 'C5xx and lets you execute and debug applications programs by using the 'C5xx 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 'C5xx 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 'C5xx board that works with the emulator; the emulator doesn't contain a 'C5xx device, so it must use a 'C5xx target board. Usually, the target system is a board that you have designed; you use the emulator and the debugger to help you debug your design.

totem-pole output: An output circuit that actively drives both high and low logic levels.

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

Sy

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

IMPORTANT NOTICE

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

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

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

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

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

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

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