TMS320C54x
C Source Debugger
User’s Guide

Literature Number: SPRU099D
July1998

PRINTED WITH SOY INK™

TEXAS INSTRUMENTS

Printed on Recycled Paper

IMPORTANT NOTICE

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

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

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

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

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

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

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

Preface

Read This First

About This Manual

This book tells you how to use the TMS320C54x C source debugger with the following debugging tools to test and refine your code:

❑ Emulator
❑ Simulator

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

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

❑ The basic debugger environment is a general-purpose debugging environment. You can use standard data-management commands and run-type commands to test and evaluate your code.

❑ The profiling environment is a special environment for collecting statistics about code execution. You can use the profiling environment to identify areas in your code where you want to improve performance.

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

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

This book is meant to be used with the online help included with the C source debugger. The online help provides you with information about the windows, menu items, icons, and dialog boxes of the debugger interface. For information on how to access the online help, see section 1.6 on page 1-11.

Notational Conventions

This document uses the following conventions.

- The TMS320C54x family of devices is referred to as 'C54x.
- Debugger commands are not case sensitive; you can enter them in lowercase, uppercase, or a combination. To emphasize this fact, commands are shown throughout this user's guide in both uppercase and lowercase.
- Program listings and examples are shown in a `special font`. Some examples use a **`bold version`** to identify code, commands, or portions of an example that *you* enter. Here is an example:

Command	Result Displayed in the Command Window
`whatis aai`	`int aai[10][5];`

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

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

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

  load *object filename*

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

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

  run [*expression*]

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

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

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

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

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

Related Documentation From Texas Instruments

The following books describe the TMS320C54x 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.

TMS320C54x 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 'C54x generation of devices.

TMS320C54x Optimizing C Compiler User's Guide (literature number SPRU103) describes the 'C54x C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C54x generation of devices.

TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals (literature number SPRU131) describes the TMS320C54x 16-bit, fixed-point, general-purpose digital signal processors. Covered are its architecture, internal register structure, data and program addressing, the instruction pipeline, and on-chip peripherals. Also includes development support information, parts lists, and design considerations for using the XDS510 emulator.

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

Related Documentation

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

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

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

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

FCC Warning

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

Trademarks

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

HP-UX is a trademark of Hewlett-Packard Company.

PC is a trademark of International Business Machines Corporation.

Solaris, SunOS and OpenWindows are trademarks of Sun Microsystems, Inc.

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

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

Windows and Windows NT are registered trademarks of Microsoft Corporation.

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

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

If You Need Assistance . . .

❑ World-Wide Web Sites

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

❑ North America, South America, Central America

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

DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/pub/tms320bbs

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

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

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	

Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/

❑ Japan

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

❑ Documentation

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

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251-1443

Email: dsph@ti.com

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

Contents

Many of the debugger commands accept C expressions as parameters. This chapter provides general information about the rules governing C expressions and describes specific implementation features related to using C expressions as command parameters.

In some circumstances, you may find it helpful to know the steps that the debugger goes through during the invocation process; this appendix lists these steps.

Describes the analysis registers and explains how to use them to create customized analysis commands.

Figures

Tables

Examples

Chapter 1

Overview of the Code Development and Debugging System

The C source debugger is an advanced programmer's interface that helps you to develop, test, and refine ’C54x C programs (compiled with the ’C54x optimizing ANSI C compiler) and assembly language programs. The debugger is the interface to the ’C54x simulator and the scan-based emulator.

This chapter gives an overview of the C source debugger, describes the code development environment, and explains how you must prepare your program for debugging. This chapter also describes the parallel debug manager (PDM) for use with the ’C54x emulator.

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

Topic **Page**

1.1 Key Features of the Debugger

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

- ❑ **Fully configurable graphical user interface.** The C source debugger separates code, data, and commands into manageable portions. The graphical user interface is intuitive and follows the conventions used by your windowing system.

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

- ❑ **On-screen editing.** You can change any data value displayed in any window—just click and type.

- ❑ **Automatic update.** The debugger automatically updates information on the screen, highlighting changed values.

- ❑ **Dynamic profiling.** In addition to the basic debugging environment, a second environment—the *profiling environment*—is available for ICECrusher/ARM7TDMIE-based devices. The profiling environment provides a method for collecting execution statistics about specific areas in your code. This gives you immediate feedback on your application's performance and helps you identify bottlenecks within the code.

- ❑ **Analysis module.** In addition to the basic debugger features, the 'C54x has an analysis module on the chip that allows the emulator to monitor the operations of your target system. This expands your debugging capabilities beyond simple software breakpoints.

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

1.2 About the C Source Debugger Interface

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

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

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

Figure 1–1. The Basic Debugger Display

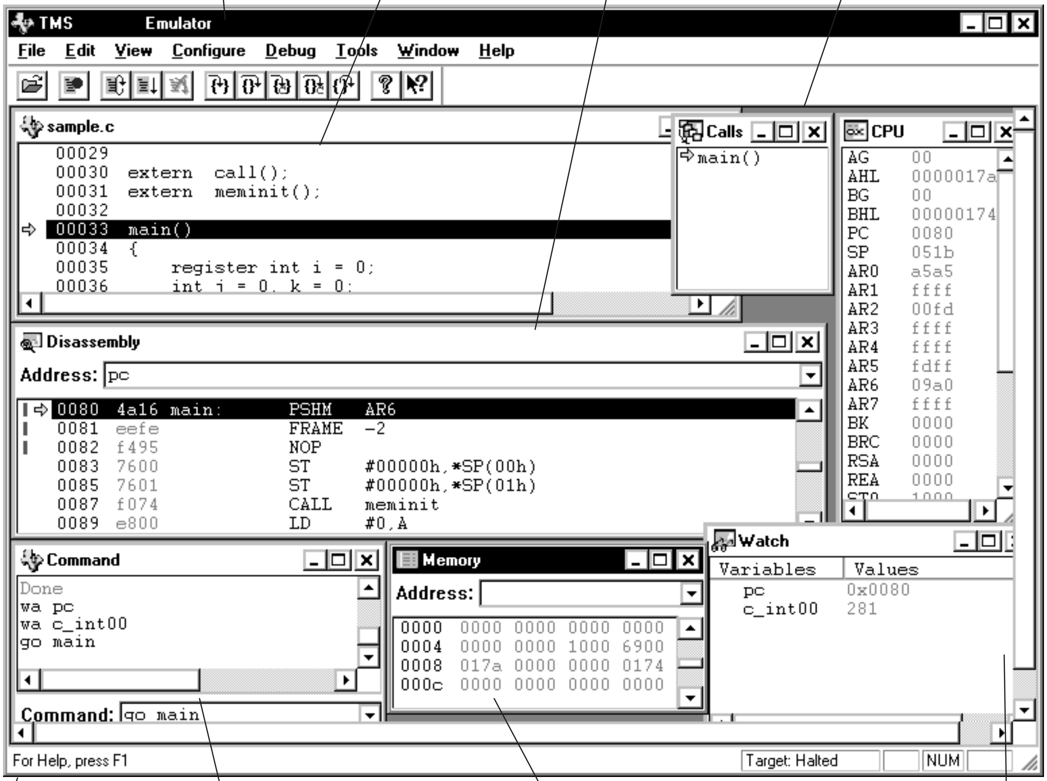
Pulldown menus are available for most tasks.

Toolbar icons let you select commands quickly.

C source display

Disassembly display

Natural-format data displays



```
00029
00030  extern  call();
00031  extern  meminit();
00032
00033  main()
00034  {
00035      register int i = 0;
00036      int j = 0, k = 0;
```

```
main()
```

```
AG   00
AHL  0000017a
BG   00
BHL  00000174
PC   0080
SP   051b
AR0  a5a5
AR1  ffff
AR2  00fd
AR3  ffff
AR4  ffff
AR5  fdff
AR6  09a0
AR7  ffff
BK   0000
BRC  0000
RSA  0000
REA  0000
```

Address: pc

```
0080 4a16 main:     PSHM   AR6
0081 eefe           FRAME  -2
0082 f495           NOP
0083 7600           ST     #00000h,*SP(00h)
0085 7601           ST     #00000h,*SP(01h)
0087 f074           CALL   meminit
0089 e800           LD     #0,A
```

```
Done
wa pc
wa c_int00
go main
```

Command: go main

Address:

```
0000  0000 0000 0000 0000
0004  0000 0000 1000 6900
0008  017a 0000 0000 0174
000c  0000 0000 0000 0000
```

Variables	Values
pc	0x0080
c_int00	281

Context-sensitive status bar

Interactive command entry with command history

Scrolling data displays with on-screen editing

Function call traceback

Descriptions of the debugger windows and their contents

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

- Code-display windows display assembly language or C code. These code-display windows are:
  - A File window displays any text file that you want to display; its main purpose, however, is to display C source code. You can display multiple File windows at one time.
  - The Disassembly window displays the disassembly (assembly language version) of memory contents.
  - The Calls window displays the currently active stack of function calls and previous function calls if you are debugging a C program. When you call a function, it is pushed onto the stack. When the function returns, it is popped off the stack.
- The Profile window displays statistics about code execution
- Data-display windows are for observing and modifying various types of data. There are five data-display windows:
  - A Memory window displays the contents of a range of memory. You can display multiple Memory windows to allow you to view different sections of memory at one time.
  - The CPU window displays the contents of ’C54x registers.
  - A Watch window displays selected data such as variables, specific registers, or memory locations. You can display multiple Watch windows to allow you to view multiple variables, register, or memory locations at one time.
  - A Variable window displays all variables declared in the program’s current function (through the Local tab), as well as variables on the currently executing line of C code and in the previous C statement (through the Auto tab).
- The Command window provides an area for typing in commands and reentering commands and an area for displaying various types of information, such as progress messages, error messages, or command output.

Table 1–1 summarizes the purpose of each window, how each window is created, and in which debugging mode each window is visible. (See Chapter 13, *Summary of Commands*, for speed key and pull-down menu commands).

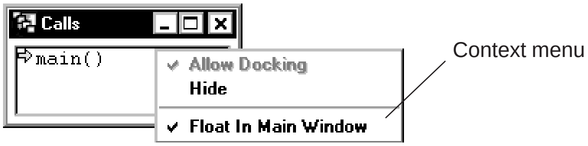
Table 1–1. Summary of Debugger Window Descriptions

Window	Purpose	Created	Mode
Calls	Lists the current function, its caller, and the caller's caller, etc. for C functions	❑ Automatically when you are displaying C code ❑ With the CALLS command if you previously closed the Calls window	❑ Auto ❑ Mixed
Command	❑ Provides a command line for entering commands ❑ Provides a display area for echoing commands and displaying command output, errors, and messages	Automatically	All
CPU	Shows the contents of the 'C54x registers	Automatically	All
Disassembly	Displays the disassembly (or reverse assembly) of memory contents	Automatically	All
File	❑ Displays C source files ❑ Displays assembly source files ❑ Displays text files	❑ With the File→Open menu option ❑ Automatically when your program executes C code, assembly code, or serial assembly code assembled with the –g assembler option	❑ Auto ❑ Mixed
Memory	Displays the contents of memory. Reference addresses, determined by the size of the window, are listed in the first column.	❑ Automatically for the default Memory window only ❑ With the MEM command and a unique *window name* for additional Memory windows	All
Profile	Displays statistics collected during a profiling session	By entering the profiling environment: Tools→Profile→Profile Mode	Mixed
Watch	Displays the values of selected expressions, structures, arrays, or pointers	❑ With the Configure→Watch Add menu option ❑ With the WA and DISP commands	All
Variable	Displays variables and their associated values for the current function, as well as variables on the currently executing line of C code and variables from the previous statement	❑ Automatically when you are displaying C code ❑ With the CALLS command if you have previously closed the Calls window ❑ With the View→Variable Window menu option	❑ Auto ❑ Mixed

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

1) Move your pointer over a debugger window.

2) Click the right mouse button. This displays a context menu like the following example:

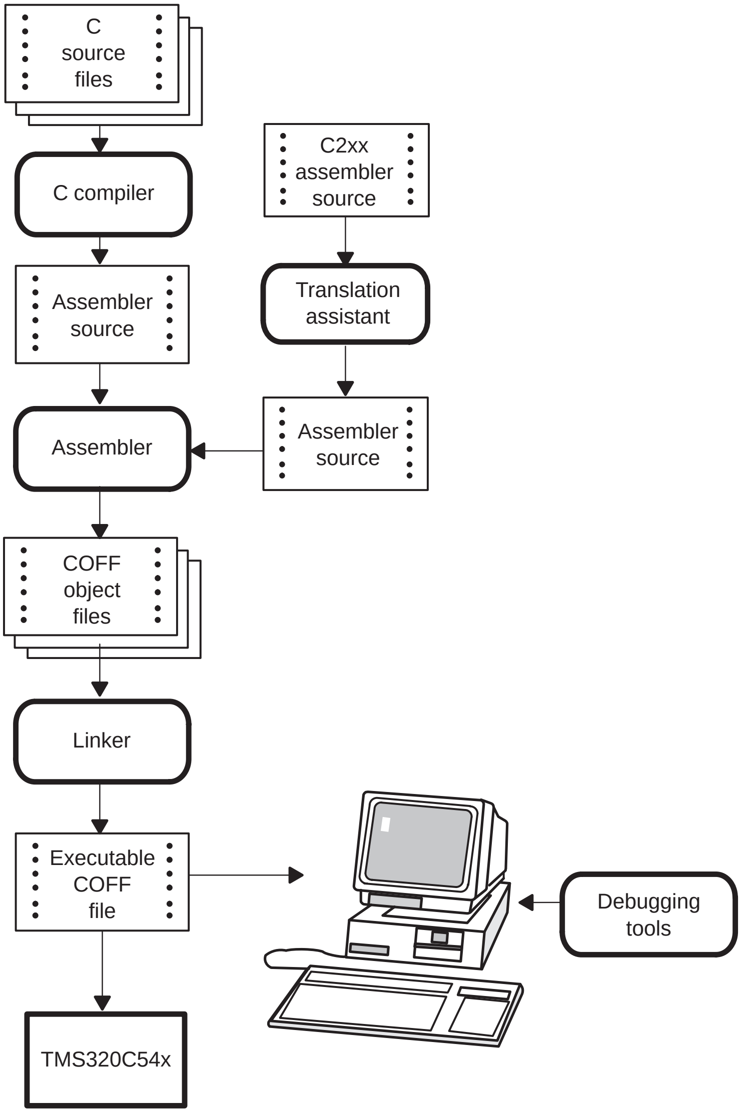


Each context menu option that is currently selected has a check mark (✓) preceding it, and those that are unselected do not. Clicking an option toggles between selected and unselected.

1.3 Developing Code for the TMS320C54x

The 'C54x is well supported by a complete set of hardware and software development tools, including a C compiler, an assembler, and a linker. Figure 1–2 illustrates the basic 'C54x code development flow.

Figure 1–2. TMS320C54x Software Development Flow



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

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

- ❏ The **C compiler** accepts C source code and produces TMS320C54x assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are included in the compiler package:

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

  See the *TMS320C54x Optimizing C Compiler User's Guide* for more information.

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

  See the *TMS320C54x Assembly Language Tools User's Guide* for more information.

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

  See the *TMS320C54x Assembly Language Tools User's Guide* for more information.

- ❏ The main product of this development process is a module that can be executed in a **TMS320C54x target system**.

- ❏ You can use debugging tools to refine and correct your code. Available products include:

  - An instruction-accurate and clock-accurate software simulator
  - An XDS™ emulator

1.4 About the Parallel Debug Manager (Emulator Only)

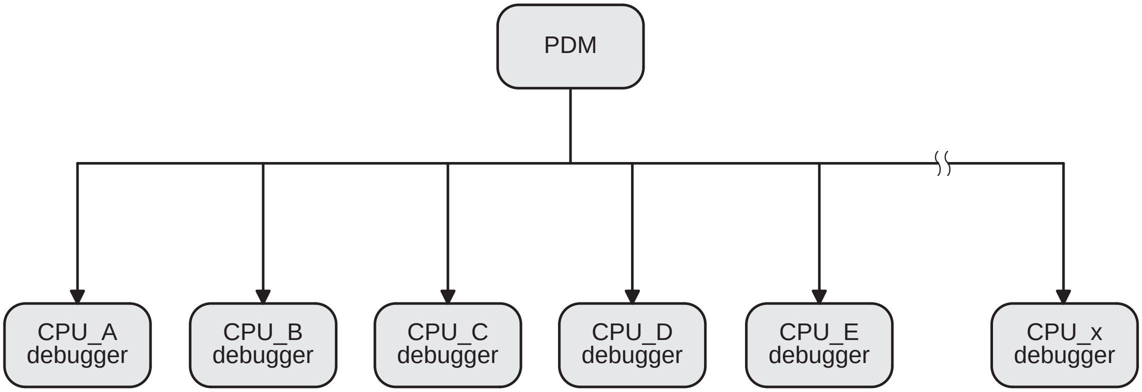
The TMS320C54x emulation system is a true multiprocessing debugging system. It allows you to debug your entire application by using the parallel debug manager (PDM). The PDM is a command shell that controls and coordinates multiple debuggers, providing you with the ability to:

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

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

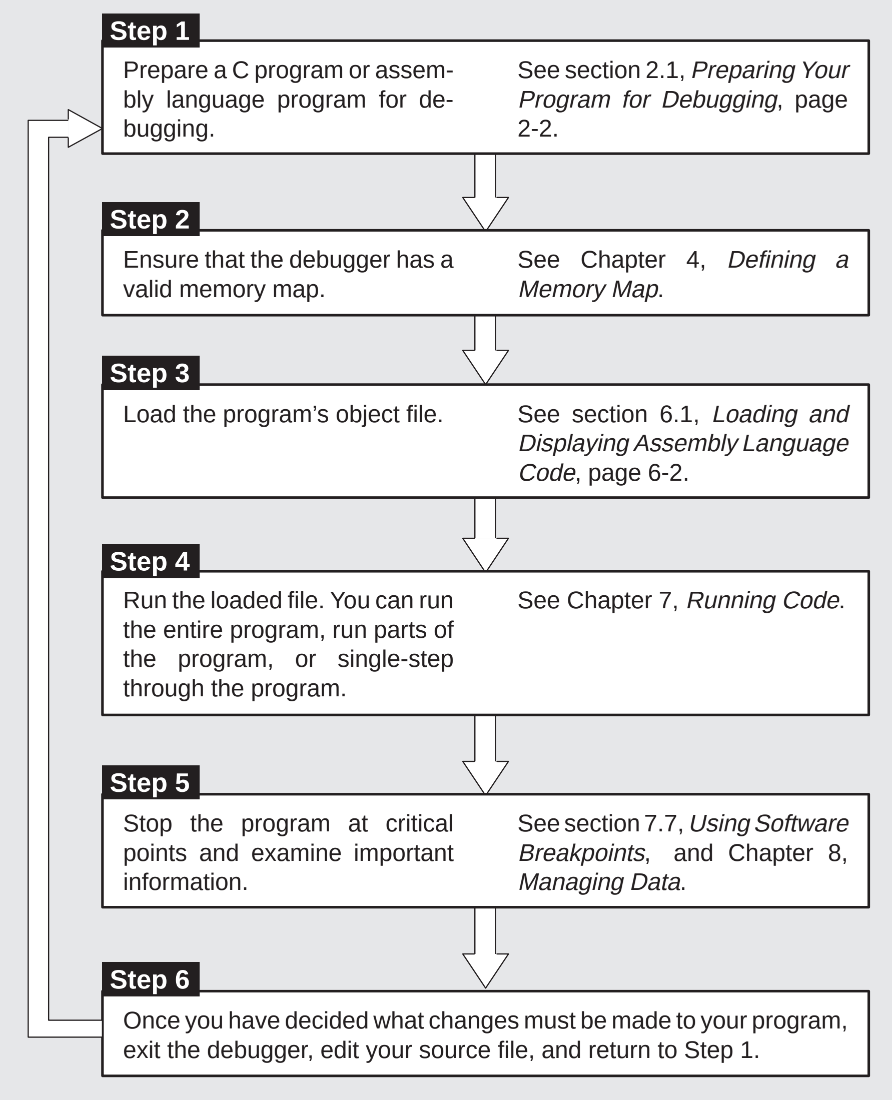
As Figure 1–3 shows, you can run multiple debuggers under the control of the PDM.

Figure 1–3. The PDM Environment



1.5 Overview of the Debugging Process

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



Step 1

Prepare a C program or assembly language program for debugging.

See section 2.1, *Preparing Your Program for Debugging*, page 2-2.

Step 2

Ensure that the debugger has a valid memory map.

See Chapter 4, *Defining a Memory Map*.

Step 3

Load the program's object file.

See section 6.1, *Loading and Displaying Assembly Language Code*, page 6-2.

Step 4

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

See Chapter 7, *Running Code*.

Step 5

Stop the program at critical points and examine important information.

See section 7.7, *Using Software Breakpoints*, and Chapter 8, *Managing Data*.

Step 6

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

1.6 Accessing Online Help

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

Accessing a list of help topics

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

1) Open the list of help topics by using one of these methods:
   - ❑ Click the Help Contents icon on the toolbar:

     ?

   - ❑ From the Help menu, select Help Topics.
   - ❑ From the command line, enter:

     `help` ↵

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

Accessing context-sensitive help

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

- ❑ To find out about an item in the debugger display, follow these steps:
  1) Click the Help icon on the toolbar:

     ↖?

     This changes the pointer to a question mark.

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

- ❑ To find out about a dialog box or a window, follow these steps:
  1) Make the window or the dialog box active.
  2) Press F1 .

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

  Help

Accessing help for debugger commands

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

help *debugger command*

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

Chapter 2

Getting Started With the Debugger

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

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

Topic **Page**

2.1 Preparing Your Program for Debugging

Before you use the debugger, you must create an executable object file. To do so, start with C source and/or assembly language code. You can use the cl500 shell program to compile, assemble, and link your source code, creating an executable object file. To be able to debug the object file, you must use the –g shell option. The –g option generates symbolic debugging directives that are used by the debugger.

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

For more information about the cl500 shell program and its options and about creating an executable object file for use with the debugger, see the *TMS320C54x Optimizing C Compiler User's Guide*.

Debugging optimized code

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

Profiling optimized code

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

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

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

Setting up D_DIR for Windows operating systems

To set the D_DIR environment variable for Windows™ 95 and Windows NT™ operating systems, use this syntax:

SET D_DIR=*pathname*$_1$ [;*pathname*$_2$ . . .]

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

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

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

Setting up D_DIR for SPARC and HP-UX operating systems

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

setenv D_DIR ”*pathname***”**

If you are using SunOS™:

- ❑ For C shells:

  setenv D_DIR ”*pathname***”**

- ❑ For Bourne or Korn shells:

  D_DIR=”*pathname***”**
  export D_DIR

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

2.3 Identifying Directories That Contain Program Source Files (D_SRC)

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

Setting up D_SRC for Windows operating systems

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

SET D_SRC=*pathname*$_1$[;*pathname*$_2$ . . .]

For example, if your ’C54x programs were in a directory named source on drive C, the D_SRC setup would be:

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

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

Setting up D_SRC for SPARC and HP-UX operating systems

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

setenv D_SRC ”*pathname1*[*;pathname2;...*]**”**

If you are using SunOS:

- ❑ For C shells:

  setenv D_SRC ”*pathname*$_1$[**;***pathname*$_2$ . . .]**”**

- ❑ For Bourne or Korn shells:

  D_SRC=”*pathname***”**
  export D_SRC

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

2.4 Setting Up Default Debugger Options (D_OPTIONS)

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

Setting up D_OPTIONS for Windows operating systems

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

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

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

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

Setting up D_OPTIONS for SPARC and HP-UX operating systems

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

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

If you are using SunOS:

- ❑ For C shells:

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

- ❑ For Bourne or Korn shells:

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

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

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

2.5 Resetting the Emulator

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

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

The –x option tells the emurst utility to ignore any options specified with the D_OPTIONS environment variable. For more information about the –x option, see page 2-15.

The –p option *number* identifies the I/O port address that the debugger uses for communicating with the emulator. For more information about the –p option, see page 2-13.

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

```
CANNOT DETECT TARGET POWER
```

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

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

2.6 Invoking the Debuggers and the PDM

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

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

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

Invoking a stand-alone debugger

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

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

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

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

sim54x | emu54x [*filename*] *options*

sim54x invokes the debugger for the simulator.

emu54x invokes the debugger for the emulator.

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

options supply the debugger with information on how to handle files, manage the display, and input information.

Invoking multiple debuggers (emulator only)

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

```
pdm [–t filename]
```

- ❑ **pdm** is the command used to invoke the debugger.

- **–t** option allows you to specify your own customized initialization command file to use intead ofsiminit.cmd or emuinit.cmd or or init.cmd. The format for the –t option is:

  –t *filename.cmd*

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

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

  If you do not supply an extension for the filename, the debugger assumes that the extension is .out.

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

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

```
Cannot open take file.
```

Note:

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

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

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

spawn **emu54x** **–n** *processor_name* [*filename*] [*options*]

- **emu54x** is the executable that invokes the debugger.

  To invoke a debugger, the PDM must be able to find the executable file for that debugger. The PDM first searches the current directory and then searches the directories listed with the PATH statement or the D_DIR environment variable. To set up this environment variable, see section 2.2, *Identifying Alternate Directories for the Debugger to Search (D_DIR)*, on page 2-3.

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

  The processor name must match one of the names defined in your board configuration file (see Appendix C, *Describing Your Target System to the Debugger*). For example, to invoke a debugger for a ’C54x that you defined as CPU_A, you would enter:

  ```
  spawn emu54x –n CPU_A ↵
  ```

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

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

  If you do not supply an extension for the filename, the debugger assumes that the extension is .out.

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

2.7 Summary of Debugger Options

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

Table 2–1. Summary of Debugger Options

Option	**Brief Description**	**Debugger Tools**
–c	Clear the .bss section	All
–d *machine name*	Display the debugger on a different machine	All (X Window System™ only)
–f *filename*	Identify a new board configuration file	Emulator
–i *pathname*	Identify additional directories	All
–l	Enables pipeline conflict detection	Simulator
–mv	Select the device version to simulate	Simulator
–n *device_name*	Identify device for debugging	Emulator
–p *port_address*	Identify the port address	Emulator
–profile	Enter the profiling environment	All
–s *filename*	Load the symbol table only	All
–t *filename*	Identify a new initialization file	All
–v	Load without the symbol table	All
–w	Writes pipeline conflict warning to a file (must be used with –l option).	Simulator
–x	Ignore D_OPTIONS	All

Clearing the .bss section (–c option)

The –c option clears the .bss section when the debugger loads code. Use this option when you have C programs that use the RAM initialization model (specified with the –cr linker option described in the *TMS320C54x Assembly Language Tools User's Guide*).

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

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

–d *machine_name***:0**

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

You can also specify a different machine by using the DISPLAY environment variable . If you use both the DISPLAY environment variable and the –d option, –d overrides DISPLAY.

Identifying a new configuration file (–f option)

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

–f *filename***.dat**

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

Identifying additional directories (–i option)

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

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

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

Enabling pipeline conflict detection (–l option)

The –l option enables pipeline conflicts to be detected when debugging code. When a pipeline conflict is detected, a warning message is written to the command window, stating which register bit or field name is associated with the conflict. Also listed are address and opcode of the second instruction in the instruction pair causing the conflict.

Along with providing a warning of the pipeline conflict, the –l option, when used, also halts execution of code.

The –w option can be used in conjunction with the –l option. For more information on the –w option, refer to page 2-15.

Selecting the device version (–mv option)

The –mv option allows you to determine which specific ’C54x device the simulator wil be simulating. It also defines the peripherals that can be simulated with this device, and turns on any of the device-specific features not available on all ’C54x derivatives, like extended addressing for the ’C548, etc. The different options are described in the following table:

Option	Device Simulated	Init File	Peripherals Simulated
–mv540	’C540	sim540.cmd	Serial port 0, serial port 1, timer
–mv541	’C541	sim541.cmd	Serial port 0, serial port 1, timer
–mv542	’C542	sim542.cmd	Buffered serial port, TDM serial port, timer, HPI
–mv543	’C543	sim543.cmd	Buffered serial port, TDM serial port, timer
–mv544	’C544	sim544.cmd	Serial port 0, serial port 1, timer
–mv545	’C545	sim545.cmd	Buffered serial port, serial port 1, timer, HPI
–mv545lp	’C545lp	sim545lp.cmd	Buffered serial port, serial port 1, timer, HPI
–mv546	’C546	sim546.cmd	Buffered serial port, serial port 1, timer
–mv547	’C547	sim547.cmd	Buffered serial port, serial port 1, timer, HPI
–mv548	’C548	sim548.cmd	Buffered serial port, 1, buffered serial port 2, TDM serial port, timer, HPI
–mv549	’C549	sim549.cmd	Buffered serial port 1, buffered serial port 2, TDM serial port, timer, HPI

Note: If the –mv option is not specified, ’C540 is simulated.

Identifying the processor to be debugged (–n option)

The –n option is valid only when you are using the emulator. The –n option allows you to specify which particular ’C54x to debug when you are using the SPAWN command to invoke multiple debuggers. The processor name must match one of the names defined in your board.cfg file. The format for this option is:

–n *device_name*

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

Identifying the I/O port address (–p option)

The –p option specifies which I/O port the debugger uses to communicate with the emulator. The format for the –p option is:

–p *port_address*

If you use the default switch settings, you do not need to use the –p option. **If you use nondefault switch settings in your installation of the XDS51x system, you must use –p**. For information on switch settings, see the *XDS51x Installation Guide*; determine your switch settings, and replace *port address* with one of these values:

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

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

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

Entering the profiling environment (–profile option)

This option is valid only when you are using the simulator. The –profile option allows you to bring up the debugger in a profiling environment so that you can collect statistics about code execution. Only a subset of the basic debugger features is available in the profiling environment. For more information about the profiling environment, see Chapter 9, *Profiling Code Execution*.

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

Loading the symbol table only (–s option)

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

–s *filename***.out**

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

Identifying a new initialization file (–t option)

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

–t *filename.cmd*

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

Loading without the symbol table (–v option)

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

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

Writing pipeline conflict warnings to a file (–w option)

The –w option can only be used in conjunction with the –l option. The –l option allows pipeline conflicts to be detected when executing code with the debugger. The –w option allows code execution to continue without interruption. The –w option allows the pipeline conflict messages to be written to a file, instead of halting code execution and writing a warning message to the command window, as the –l option alone does.

You can specify a file in your current directory to be used when the debugger writes pipeline conflict warning messages. However, if no file is specified, the debugger automatically creates a file called latency.msg when the first pipeline conflict is detected.

Ignoring D_OPTIONS (–x option)

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

2.8 Debugging Modes

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

Auto mode

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

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

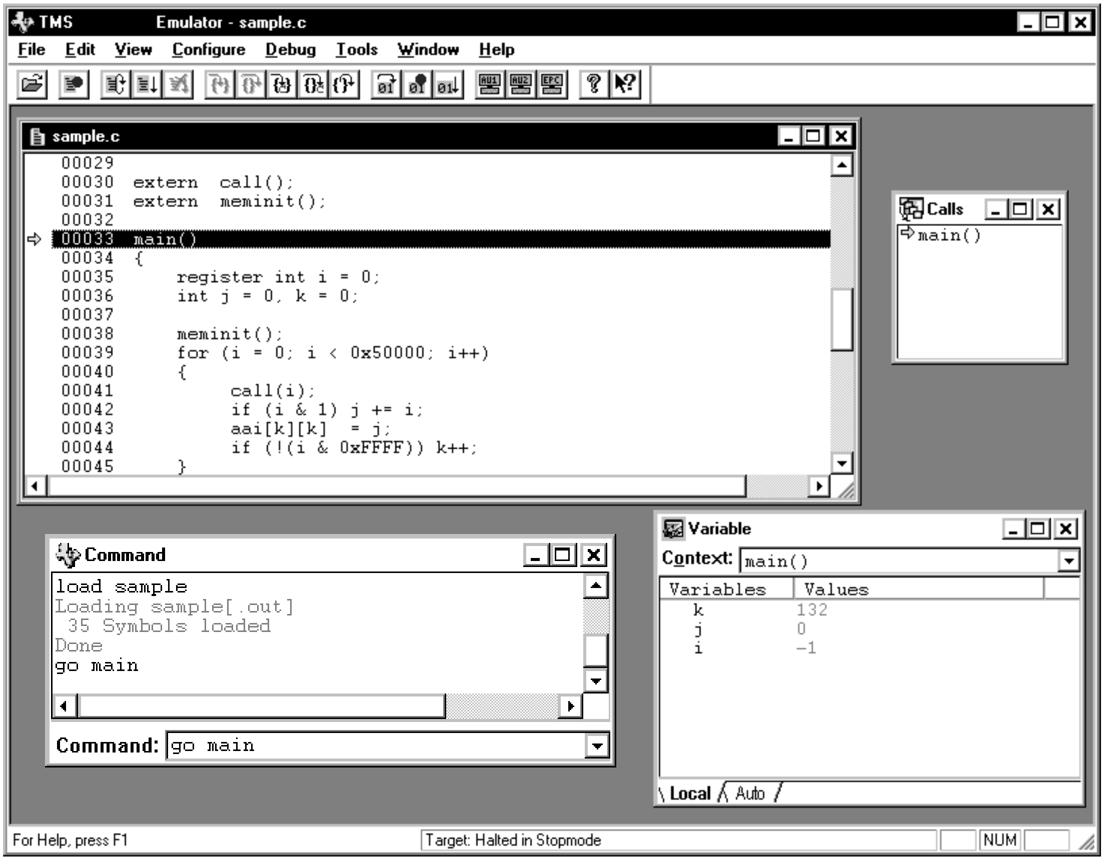
When you first invoke the debugger, you see a display similar to this.

❑ When the debugger is running C code, you see a C display similar to the one in Figure 2–1. (This assumes that the debugger can find your C source file to display in the File window. If the debugger cannot find your source, it displays the disassembly code only.)

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

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

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



```

extern  call();
extern  meminit();

main()
{
    register int i = 0;
    int j = 0, k = 0;

    meminit();
    for (i = 0; i < 0x50000; i++)
    {
         call(i);
         if (i & 1) j += i;
         aai[k][k]  = j;
         if (!(i & 0xFFFF)) k++;
    }
```

Calls:

```
main()
```

Command window:

```
load sample
Loading sample[.out]
 35 Symbols loaded
Done
go main
```

Command: go main

Variable window — Context: main()

Variables	Values
k	132
j	0
i	–1

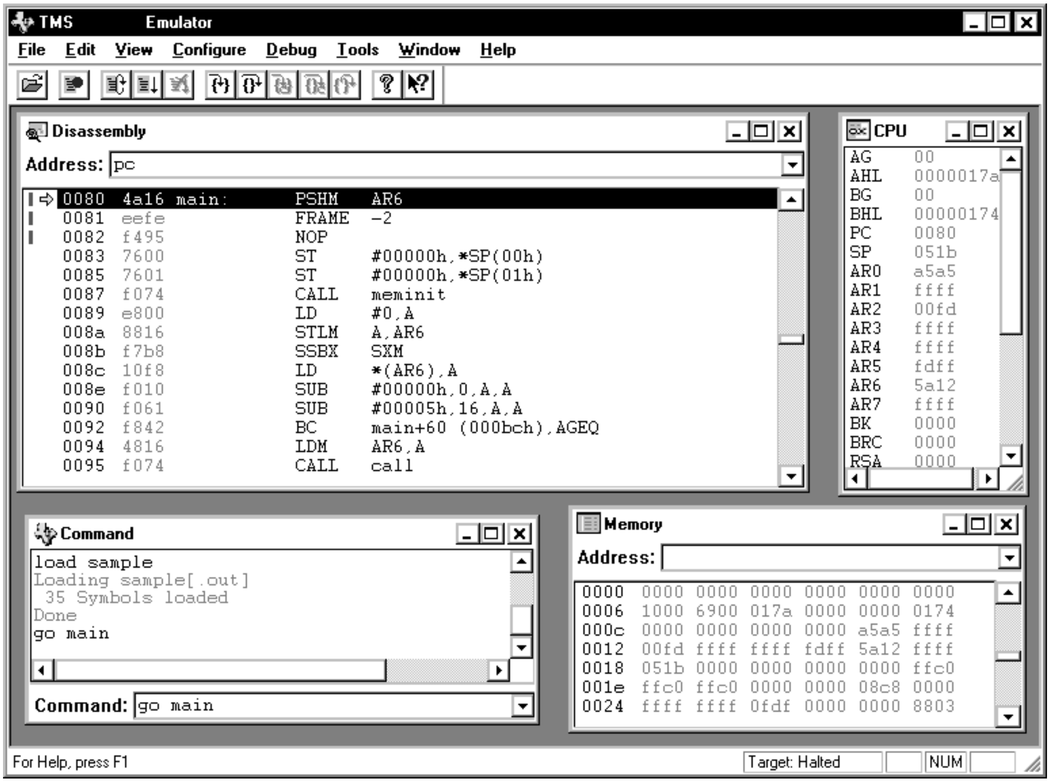
For Help, press F1 — Target: Halted in Stopmode — NUM

Assembly mode

Assembly mode is for viewing assembly language programs only. In this mode, you see a display similar to the one shown in Figure 2–2. When you are in assembly mode, you always see the assembly display, regardless of whether C or assembly language is currently running.

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

Figure 2–2. Typical Assembly Display (for Auto Mode and Assembly Mode)



Disassembly

Address: pc

```
0080  4a16 main:   PSHM   AR6
0081  eefe         FRAME  -2
0082  f495         NOP
0083  7600         ST     #00000h,*SP(00h)
0085  7601         ST     #00000h,*SP(01h)
0087  f074         CALL   meminit
0089  e800         LD     #0,A
008a  8816         STLM   A,AR6
008b  f7b8         SSBX   SXM
008c  10f8         LD     *(AR6),A
008e  f010         SUB    #00000h,0,A,A
0090  f061         SUB    #00005h,16,A,A
0092  f842         BC     main+60 (000bch),AGEQ
0094  4816         LDM    AR6,A
0095  f074         CALL   call
```

CPU

```
AG    00
AHL   0000017a
BG    00
BHL   00000174
PC    0080
SP    051b
AR0   a5a5
AR1   ffff
AR2   00fd
AR3   ffff
AR4   ffff
AR5   fdff
AR6   5a12
AR7   ffff
BK    0000
BRC   0000
RSA   0000
```

Command

```
load sample
Loading sample[.out]
 35 Symbols loaded
Done
go main
```

Command: go main

Memory

Address:

```
0000  0000 0000 0000 0000 0000 0000
0006  1000 6900 017a 0000 0000 0174
000c  0000 0000 0000 0000 a5a5 ffff
0012  00fd ffff ffff fdff 5a12 ffff
0018  051b 0000 0000 0000 0000 ffc0
001e  ffc0 ffc0 0000 0000 08c8 0000
0024  ffff ffff 0fdf 0000 0000 8803
```

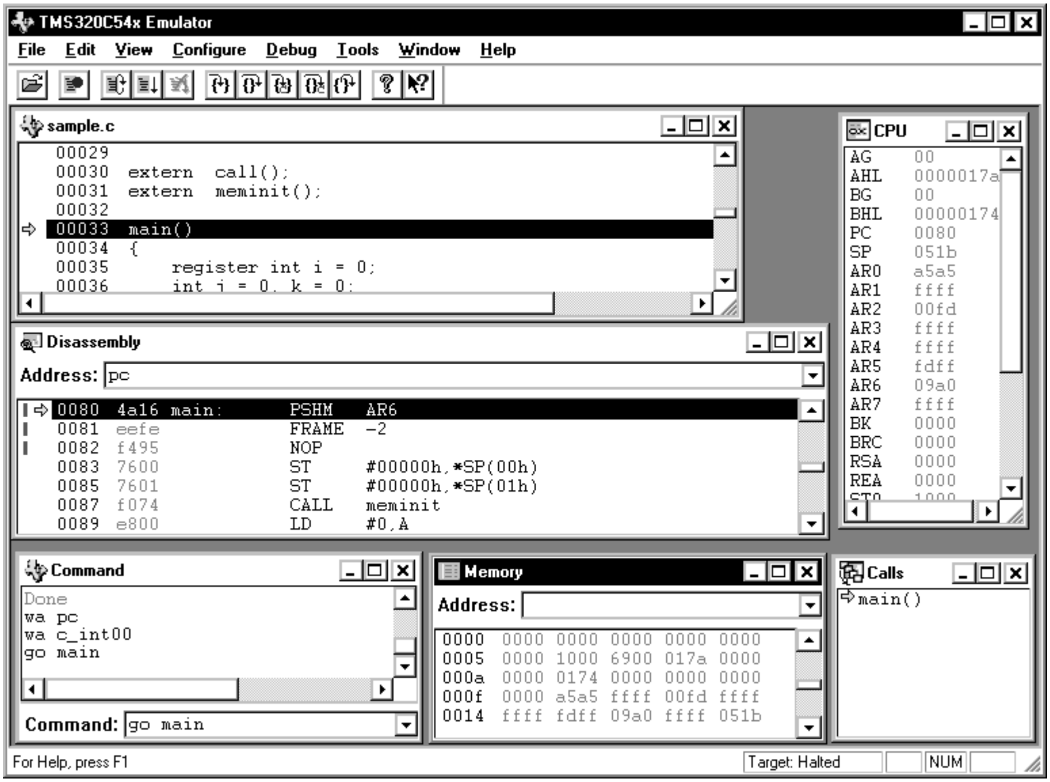
Mixed mode

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

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

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

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



```
00029
00030  extern  call();
00031  extern  meminit();
00032
00033  main()
00034  {
00035      register int i = 0;
00036      int j = 0, k = 0;
```

Address: pc

```
0080  4a16 main:      PSHM   AR6
0081  eefe            FRAME  -2
0082  f495            NOP
0083  7600            ST     #00000h,*SP(00h)
0085  7601            ST     #00000h,*SP(01h)
0087  f074            CALL   meminit
0089  e800            LD     #0,A
```

```
AG     00
AHL    0000017a
BG     00
BHL    00000174
PC     0080
SP     051b
AR0    a5a5
AR1    ffff
AR2    00fd
AR3    ffff
AR4    ffff
AR5    fdff
AR6    09a0
AR7    ffff
BK     0000
BRC    0000
RSA    0000
REA    0000
ST0    1000
```

```
Done
wa pc
wa c_int00
go main
```

Command: go main

Address:

```
0000  0000 0000 0000 0000 0000
0005  0000 1000 6900 017a 0000
000a  0000 0174 0000 0000 0000
000f  0000 a5a5 ffff 00fd ffff
0014  ffff fdff 09a0 ffff 051b
```

```
main()
```

For Help, press F1 | Target: Halted | NUM

Restrictions associated with debugging modes

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

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

❑ The CALLS, DISP, FUNC, and FILE commands are valid only in auto and mixed modes.

❑ The MEM command is valid only in assembly and mixed modes.

2.9 Exiting the Debugger or the PDM

To exit the debugger, use one of these methods:

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

  ```
  quit ↵
  ```

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

Chapter 3

Entering and Using Commands

The debugger provides you with several methods for entering commands:

❑ From the toolbar
❑ From the menu bar
❑ With function keys (see Chapter 13, *Summary of Commands*)
❑ From the command line (see Chapter 13, *Summary of Commands*)
❑ From a batch file

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

Topic **Page**

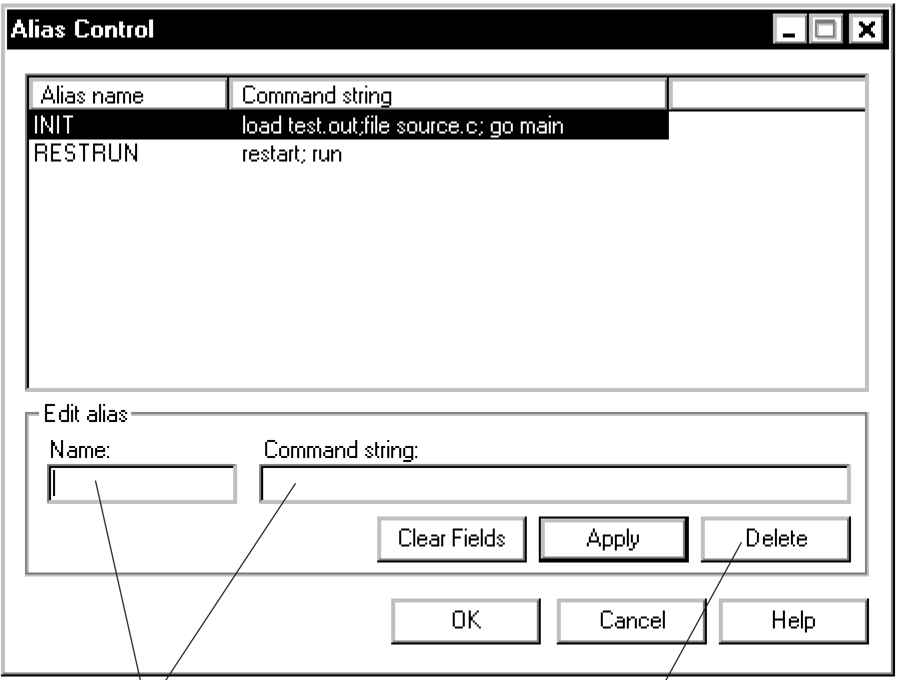
3.1 Defining Your Own Command Strings

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

Note:

Creating aliased commands in PDM is different from creating aliased commands in the debugger. For information about the PDM versions of the ALIAS and UNALIAS commands, see page 12-15.

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



Alias name	Command string
INIT	load test.out;file source.c; go main
RESTRUN	restart; run

List of defined aliases

To define an alias, enter an alias name and command string and click Apply.

To delete an alias, select an alias name and click Delete.

Defining an alias

To define an alias, follow these steps:

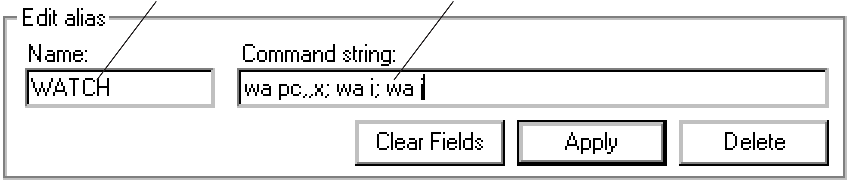
1) From the Configure menu, select Alias Commands. This displays the Alias Control dialog box.

2) In the Name field, enter a name for the alias.

3) In the Command string field, enter the command string that you want to associate with the alias name. If you want to associate multiple commands with the alias, separate the commands with a semicolon.

Enter a name for the alias.

Enter the command string that you want to associate with the alias name.



Edit alias

Name: WATCH

Command string: wa pc,,x; wa i; wa j

4) Click Apply.

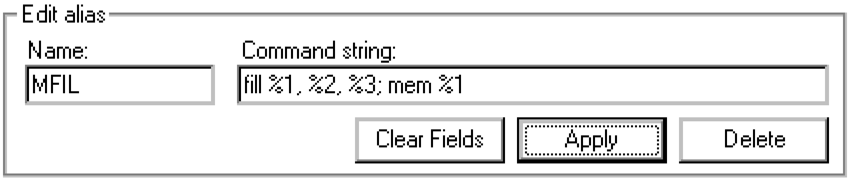
5) Click OK to close the Alias Control dialog box.

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

Defining an alias with parameters

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

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



Edit alias

Name: MFIL

Command string: fill %1, %2, %3; mem %1

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

```
mfil 0x808020,0x18,0x1122
```

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

Editing or redefining an alias

To edit or redefine an alias, follow these steps:

1) From the Configure menu, select Alias Commands. This displays the Alias Control dialog box.

2) From the list of aliases at the top of the dialog box, select the alias that you want to edit or redefine.

3) In the Name and Command string fields, make the appropriate changes.

4) Click Apply.

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

Deleting an alias

To delete an alias, follow these steps:

1) From the Configure menu, select Alias Commands. This displays the Alias Control dialog box.

2) From the list of aliases at the top of the dialog box, select the alias that you want to delete.

3) Click Delete.

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

Considerations for using alias definitions

Alias definitions are lost when you exit the debugger. If you want to reuse aliases, define them in a batch file. Use the ALIAS command, as described on page 13-13.

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

3.2 Entering Operating-System Commands From Within the Debugger

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

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

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

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

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

Entering a single command from the debugger command line

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

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

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

0 The debugger immediately returns to the debugger environment after the last item of information is displayed.

1 The debugger does not return to the debugger environment until you enter:

```
exit ⏎.
```

(This is the default.)

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

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

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

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

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

Entering several commands from a system shell

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

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

```
exit ↵
```

For information about the PDM version of the SYSTEM command, see page 12-16.

3.3 Creating and Executing a Batch File

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

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

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

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

```
echo Loading object code
load testcode.out

echo Loading screen configuration
sconfig myconfig.clr

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

Echoing strings in a batch file

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

echo *string*

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

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

```
echo Creating new memory map
```

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

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

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

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

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

Executing commands conditionally in a batch file

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

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

If the Boolean expression evaluates to true (1), the debugger executes all commands between the IF and ELSE or ENDIF. The ELSE portion of the command is optional. See Chapter 14, *Basic Information About C Expressions*, for more information.

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

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

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

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

```
if $$EMU$$
echo Invoking initialization batch file for emulator.
use .\emu54x
take emuinit.cmd
.
.
.
endif

if $$SIM$$
echo Invoking initialization batch file for simulator.
use .\sim54x
take siminit.cmd
.
.
.
endif
.
.
.
```

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

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

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

See *Controlling PDM command execution*, page 12-10, for more information about the PDM versions of the IF and LOOP commands.

Looping command execution in a batch file

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

loop *expression*
debugger commands
endloop

These looping commands evaluate using the same method as the conditional RUN command expression. (See Chapter 14, *Basic Information About C Expressions*, for more information.)

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

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

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

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

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

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

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

The LOOP/ENDLOOP command works with the following conditions:

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

See *Controlling PDM command execution*, page 12-10, for more information about the PDM versions of the IF and LOOP commands.

Pausing the execution of a batch file

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

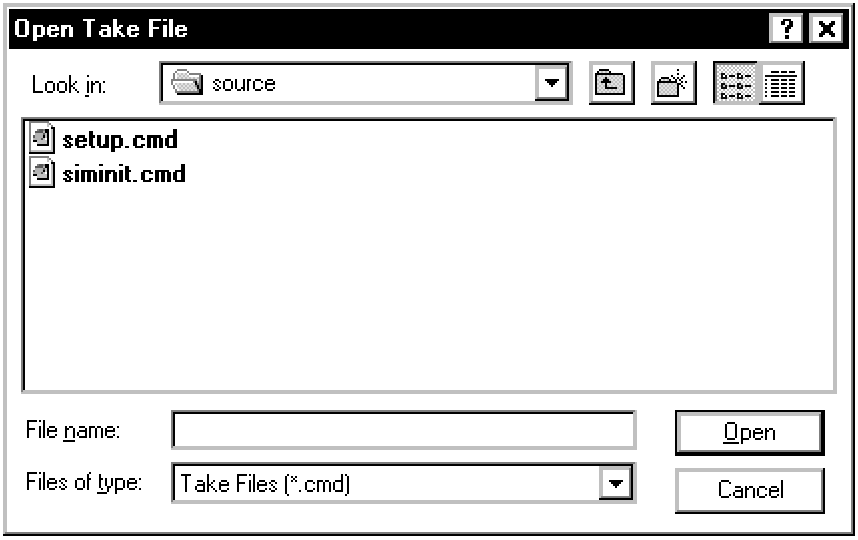
pause

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

Executing a batch file

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

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



Look in: source

setup.cmd
siminit.cmd

File name:

Files of type: Take Files (*.cmd)

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

3) Click Open.

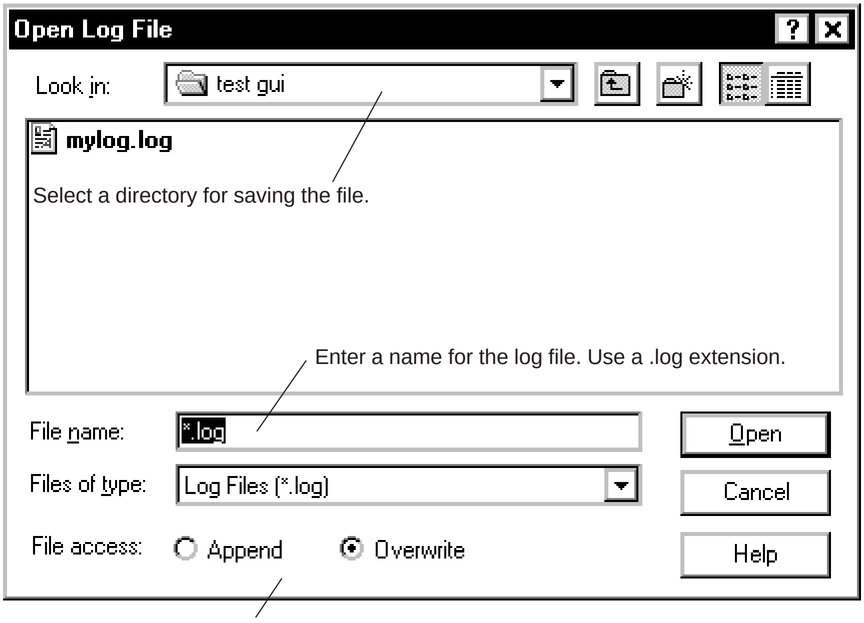
This causes the debugger to read and execute the commands in the batch file. To halt the debugger's execution of a batch file, press ESC .

3.4 Using a Log File to Reexecute a Series of Commands

The information shown in the display area of the Command window can be written to a log file. The log file is a system file that contains commands you have entered from the command line, from the toolbar, from the menus, or with function keys. The log file also contains the results from commands and error or progress messages. The debugger automatically precedes all error or progress messages and command results with a semicolon to turn them into comments. This way, you can reexecute the commands in your log file by using the File→Execute Take File menu option. (For information about creating a log file in PDM, see page 12-10.) You can view the log file with any text editor.

To begin a recording session, follow these steps:

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



Look in: test gui

mylog.log

File name: *.log

Files of type: Log Files (*.log)

File access: ○ Append ● Overwrite

Select a directory for saving the file.

Enter a name for the log file. Use a .log extension.

Select whether to append or overwrite an existing log file.

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

3) Select the name for the log file:

   a) If you are creating a new file, enter a name in the File name field. Use a .log extension to identify the file as a log file.

      Click Open.

   b) If you are using a file that already exists, select the name in the file list.

In the File access field, select one of the following actions:

- ❑ Append to add the log information to an existing file
- ❑ Overwrite to write over the contents of an existing file

Click Open.

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

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

Chapter 4

Defining a Memory Map

Before you begin a debugging session, you must supply the debugger with a memory map. The memory map tells the debugger which areas of memory it can and cannot access. The memory map is the only means for the ’C54x emulator to distinguish between program and data memory.

Topic **Page**

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

A memory map tells the debugger which areas of memory it can and cannot access. The memory map is the only way that the ’C54x emulator is able to distinguish between program and data memory. Memory maps vary, depending on the application being used. Typically, the memory map matches the MEMORY definition found in your linker command file.

Note:

When the ’C54x debugger compares memory accesses to the memory map, it performs this checking in software, not hardware. The debugger, when using the emulator, cannot prevent your program from attempting to access nonexistent memory. However, the simulator flags all illegal accesses performed by your program.

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 initially begin using the debugger, but may need to be altered at a later time. The debugger enables you to modify the default memory map or define a new memory map either interactively (as described in section 4.2 on page 4-4) or by defining the memory map in a batch file (as described in section 4.5 on page 4-11).

Potential memory map problems

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

❑ **Accessing invalid memory addresses.** If you do not supply a batch file containing memory-map commands, the debugger is initially unable to access any target memory locations. Invalid memory addresses and their contents are displayed in red in the data-display windows by default.

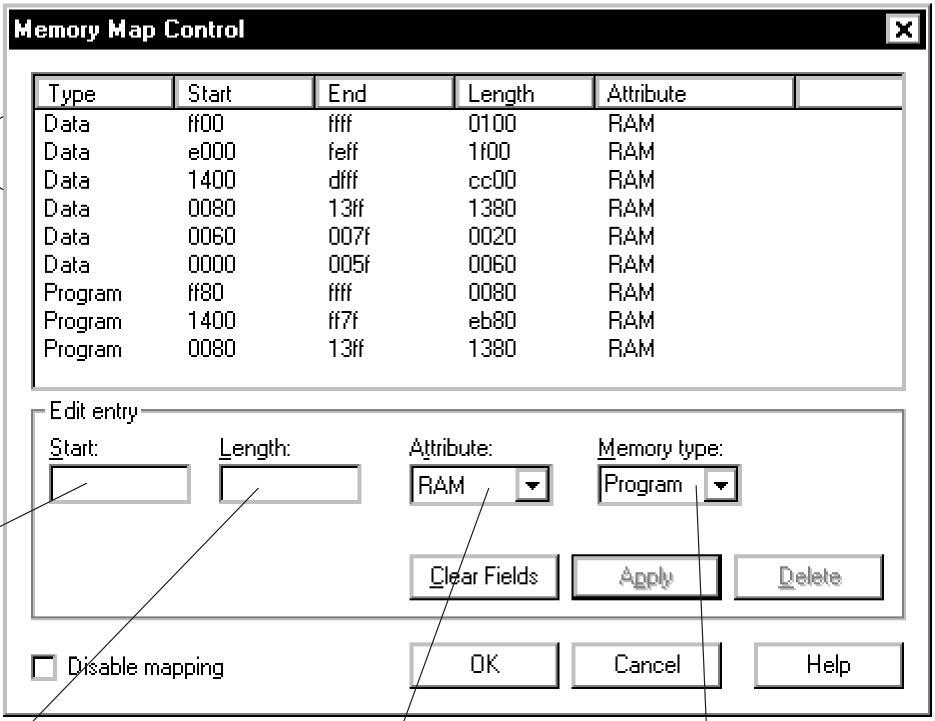
❑ **Accessing an undefined or protected area.** When memory mapping is enabled, the debugger checks each of its memory accesses against the memory map. If you attempt to access an undefined or protected area, the debugger displays an error message.

- **Loading a COFF file with sections that cross a memory range.** Be sure that the map ranges you specify in a COFF file match those that you defined in a batch file or with the Memory Map Control dialog box. Alternatively, you can turn memory mapping off during a load by disabling memory mapping (as described in section 4.3 on page 4-8). When mapping is turned off, you can still access memory locations.

- **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.

4.2 Creating or Modifying the Memory Map

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



Type	Start	End	Length	Attribute
Data	ff00	ffff	0100	RAM
Data	e000	feff	1f00	RAM
Data	1400	dfff	cc00	RAM
Data	0080	13ff	1380	RAM
Data	0060	007f	0020	RAM
Data	0000	005f	0060	RAM
Program	ff80	ffff	0080	RAM
Program	1400	ff7f	eb80	RAM
Program	0080	13ff	1380	RAM

Start:
Length:
Attribute: RAM
Memory type: Program

☐ Disable mapping

List of defined memory ranges

Enter the starting address of a memory range.

Enter the length of the memory range.

Select a memory attribute to identify the read/write characteristics of the memory range.

Select the memory type of the memory range.

Adding a range of memory

To add a range of memory, follow these steps:

1) From the Configure menu, select Memory Maps. This displays the Memory Map Control dialog box.

2) In the Memory Type field, select program or data memory.

3) In the Start field, enter the starting address for a memory range. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.

4) In the Length field, enter the length of the memory range. The length can be any C expression.

5) In the Attribute field, select a memory type to identify the read/write characteristics of the memory range.

6) Click Apply.

7) Click OK.

The following restrictions apply to identifying usable memory ranges:

- A new memory range cannot overlap an existing entry. If you define a range that overlaps an existing range, the debugger ignores the new range.

- The map ranges that you specify in a COFF file must match those that you define with the Memory Map Control dialog box.

- The origin and length values for a range that you define with the MEMORY directive in your linker command file must match the Start and Length values for the same range in the Memory Map Control dialog box.

Creating a customized memory type

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

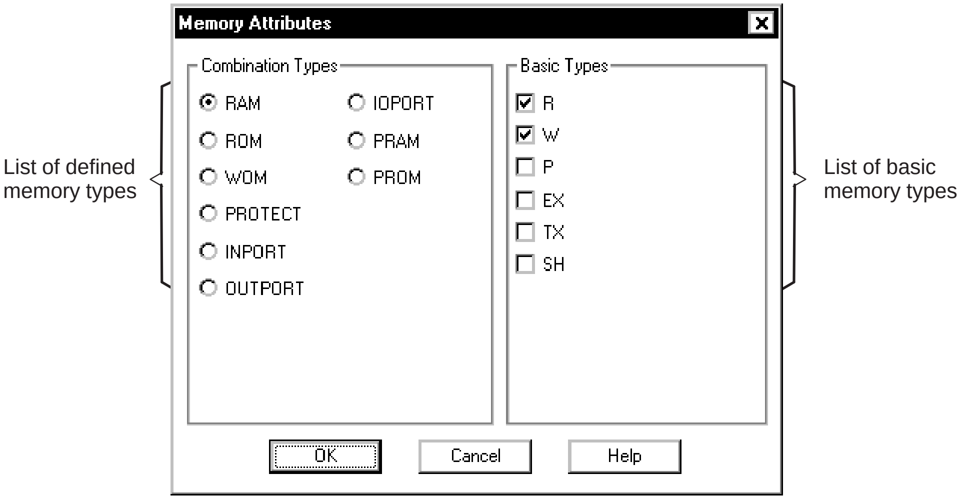
To create a customized memory type, follow these steps:

1) From the Configure menu, select Memory Maps. This displays the Memory Map Control dialog box.

2) In the Start field, enter the starting address for the memory range you want to customize. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.

3) In the Length field, enter the length of the memory range. The length can be any C expression.

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



Memory Attributes

Combination Types

● RAM ○ IOPORT
○ ROM ○ PRAM
○ WOM ○ PROM
○ PROTECT
○ INPORT
○ OUTPORT

Basic Types

☑ R
☑ W
□ P
□ EX
□ TX
□ SH

5) From the Basic Memory Type column, select the individual memory attribute that you want to apply to the memory range that you are adding.

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

6) Click OK. This closes the Memory Attributes dialog box and applies the customized memory attributes to the memory range in the Memory Map Control dialog box.

7) Add other memory ranges as needed, then click OK to close the Memory Map Control dialog box.

Deleting a range of memory

To delete a range of memory, follow these steps:

1) Select Memory Maps from the Configure menu. This displays the Memory Map Control dialog box.

2) From the list of defined ranges at the top of the dialog box, select the range that you want to delete.

3) Click Delete.

4) Click OK.

Before you can delete a memory address used as a simulated I/O port from the memory map, you must disconnect the address. See the *Disconnecting an I/O port* section on page 4-21 for information.

Modifying a defined range of memory

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

1) Select Memory Maps from the Configure menu. This displays the Memory Map Control dialog box.

2) From the list of defined ranges at the top of the dialog box, select the range that you want to modify.

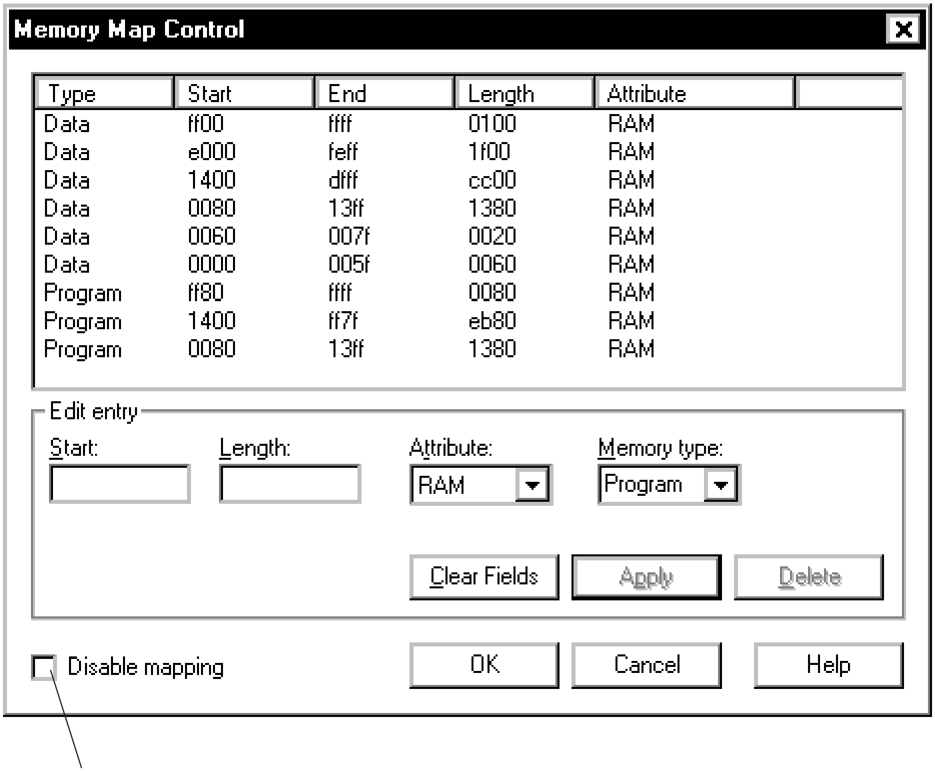
3) In the Memory Type, Start, Length, and/or Attribute fields, make the appropriate changes.

4) Click Apply.

5) Click OK.

4.3 Enabling Memory Mapping

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



Memory Map Control

Type	Start	End	Length	Attribute
Data	ff00	ffff	0100	RAM
Data	e000	feff	1f00	RAM
Data	1400	dfff	cc00	RAM
Data	0080	13ff	1380	RAM
Data	0060	007f	0020	RAM
Data	0000	005f	0060	RAM
Program	ff80	ffff	0080	RAM
Program	1400	ff7f	eb80	RAM
Program	0080	13ff	1380	RAM

Edit entry

Start:

Length:

Attribute: RAM

Memory type: Program

☐ Disable mapping

Click here to enable/disable memory mapping.

- ❑ Memory mapping is enabled when the box is empty:

☐ Disable mapping

- ❑ Memory mapping is disabled when the box is checked:

☑ Disable mapping

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

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

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

Note:

When memory mapping is enabled, you cannot:

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

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

```
Error in expression
```

4.4 A Sample Memory Map

You must define a memory map before you can run any programs, unless the program is run in a *Disable mapping* (map-off) mode. Therefore, it is convenient to define the memory map in the initialization batch files. Figure 4–1 (a) shows the memory map that is defined in the initialization batch file that accompanies the ’C54x simulator. You can use the file as is, edit it, or create your own memory map batch file to match your own configuration. You can also define the memory map after you have invoked the debugger with the Memory Map Control dialog box (see section 4.2 on page 4-4).

If you are defining the memory map in a batch file, you can use MA (map add) commands to define valid memory ranges and identify the read/write characteristics of the memory ranges. (For more information about the MA command, see section 4.5 on page 4-11.) By default, mapping is enabled when you invoke the debugger. Figure 4–1 (b) illustrates the memory map defined by the MA commands in Figure 4–1 (a).

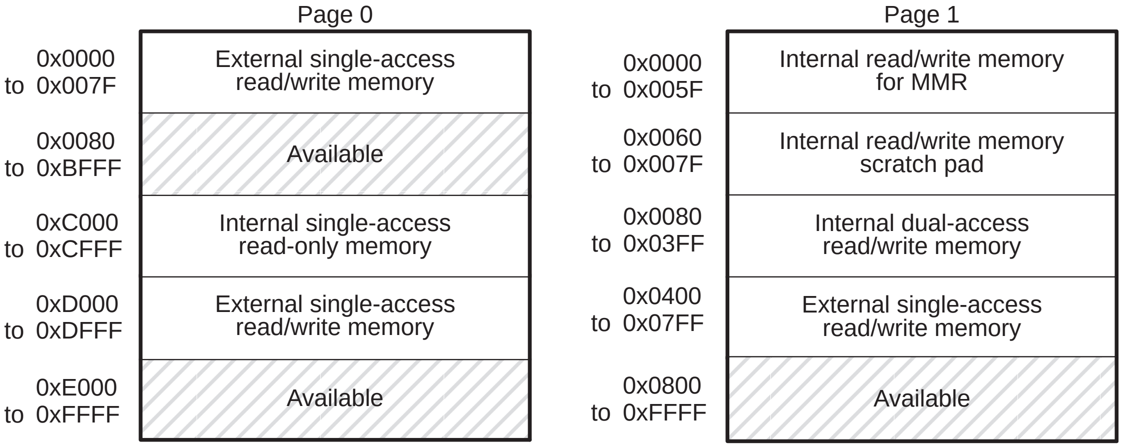
Figure 4–1. Sample Memory Map for Use With a TMS320C54x Simulator

(a) Memory map commands

```
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
```

(b) Memory map for TMS320C54x local memory


Page 0
0x0000 to 0x007F
External single-access read/write memory
0x0080 to 0xBFFF
Available
0xC000 to 0xCFFF
Internal single-access read-only memory
0xD000 to 0xDFFF
External single-access read/write memory
0xE000 to 0xFFFF
Available
Page 1
0x0000 to 0x005F
Internal read/write memory for MMR
0x0060 to 0x007F
Internal read/write memory scratch pad
0x0080 to 0x03FF
Internal dual-access read/write memory
0x0400 to 0x07FF
External single-access read/write memory
0x0800 to 0xFFFF
Available

4.5 Defining and Executing a Memory Map in a Batch File

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

Defining a memory map in a batch file

To define a memory map in a batch file, use the MA command. The syntax for 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. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.
- ❑ The *page* parameter is a one-digit number that identifies the type of memory (program or data) 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**

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

Usage notes

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

- 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 4.9, *Simulating I/O Space.*

- Be sure that the map ranges that you specify in a COFF file match those that you define with the MA command. 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 disabling the memory map (see Section 4.3, page 4-8).

- 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 0xf800, 0, 0x400,  ROM⏎
  ma 0xf800, 0, 0x400,  EX|ROM⏎
  ```

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

- If a range of memory is configured as single-access RAM (using the SA attribute with the MA command), it means only one access (read/write) can be performed on any address in the block in one cycle. You can configure more than one single-access RAM block. Simultaneous accesses to different single-access RAM blocks during the same cycle are permitted.

  For example, the following commands create two single-access RAM blocks. The blocks are 0x100 in size. If an instruction performs two accesses, one in the first block (for example, address 0x110) and another in the second block (for example, address 0x230), the instruction executes in only one cycle.

  ```
  ma 0x0100, 1, 0x0100,  R|W|SA⏎
  ma 0x0200, 1, 0x0200,  R|W|SA⏎
  ```

Contrarily, if the blocks were combined into one block and configured as one single chunk of 0x200 words (as shown in the following commangd), simultaneous accesses to addresses 0x110 and 0x230 would take two cycles to complete.

```
ma 0x0100, 1, 0x0200,  R|W|SA⏎
```

- If a range of memory is configured as dual-access RAM (using the DA attribute with the MA command), it means two simultaneous accesses (read/write) can be performed during the same cycle to the block.

  For example, the following command creates one dual-access RAM as a data page. If an instruction performs two simultaneous accesses to two addresses in this block, both accesses execute in one cycle.

  ```
  ma 0x0100, 1, 0x0100,  R|W|DA⏎
  ```

Defining a memory map using cache capabilities

Unlike the emulator and EVM, the ’C54x 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 ’C54x, do not allocate more than 20K words of memory in your memory map. For example, the following memory map allocates 64K words of ’C54x program memory.

Example 4–1. Sample Memory Map for the TMS320C54x 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 are 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 are not 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.

Executing a memory map batch file

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

- Use the File→Execute Take File... menu option from within the debugger environment.
- Use the –t debugger option to specify the batch file when you invoke the debugger. For more information, see page 2-14.
- Use the TAKE command. For more information, see section 3.3, *Creating and Executing a Batch File*, on page 3-7.

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

1) It checks to see whether you have used the –t debugger option. The –t option allows you to specify a batch file other than the initialization batch file shipped with the debugger. If it finds the –t option, the debugger reads and executes the specified file.

2) If you do not use the –t option, the debugger looks for the default initialization batch file. The batch filename for the simulator is called siminit.cmd. The batch filename for the emulator is called emuinit.cmd. The batch filename for the EVM is called evminit.cmd. If the debugger finds the proper initialization batch file, it reads and executes the file.

3) If the debugger does not find the –t option or the initialization batch file, it looks for a file called init.cmd.

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

4.6 Customizing the Memory Map

The customizable ’C54x (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 RAM
```

During execution or when the debugger performs memory accesses, the block of memory accessed is based on the ’C54x MP/$\overline{\text{MC}}$ 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 preceding 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/$\overline{\text{MC}}$ 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/$\overline{\text{MC}}$ is set to 0, accesses internal memory block, then external memory block; if MP/$\overline{\text{MC}}$ 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

Mapping on-chip dual-access RAM to program memory

The following steps describe how to map a block of memory to program memory. The memory that is mapped is configured as on-chip dual-access RAM in the data memory.

Step 1: Set the OVLY (overlay bit) in the PMST register to 1.

Step 2: Define the data-memory map before the program-memory map. It is essential to define the data-memory map for the overlay mode.

Step 3: Add a dummy program-memory map in the same region as the external memory. To do this, use the EX attribute.

Note:

The size of the data-memory map and the program-memory map must be the same.

The following is an example of mapping the on-chip dual-access RAM to program memory. The example shows the commands to set the mode to overlay.

```
ma 0x0080, 1, 0x0f80, R|W|DA
ma 0x0080, 0, 0x0f80, R|W|EX
?pmst=0xffc0 ; mp/mc=0, ovly=1
```

Simulating data memory (ROM)

With the ’C54x simulator, you can simulate the DROM bit in the ’C541, ’C543, ’C544, ’C545, ’C545LP, ’C546, ’C547, or ’C549 processor. This simulation allows you to map the on-chip program memory (ROM) to the data memory. To map the program memory (ROM) to the data memory, follow these steps:

Step 1: Set the DROM bit in the PMST register to 1.

Step 2: Invoke the simulator with the –mv541, –mv543, –mv544, –mv545, –mv545lp, –mv546, –mv547, or –mv549 option.

The following is an example of simulating data memory:

```
?pmst |=0x10   ; DROM bit is set to 1
```

Programming your memory

Setting up your memory is easiest during the initialization process. However, you can edit your memory map while your program is running.

Use the OVLY and MP/$\overline{\text{MC}}$ 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/$\overline{\text{MC}}$ bits are as follows:

- OVLY bit

  0 = external program memory (RAM)

  1 = internal program memory (RAM)

- MP/$\overline{\text{MC}}$ bit

  0 = internal program memory (ROM)

  1 = external program memory (ROM)

You can edit the the values of the OVLY and MP/$\overline{\text{MC}}$ 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.

4.7 Returning to the Original Memory Map

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

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

1) From the command line, enter **mr** ↵ to reset the memory map.

2) From the File menu, select Execute Take File.

3) From the Open Take File dialog box, select mem.map to reread the default memory map.

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

4.8 Using Multiple Memory Maps for Multiple Target Systems

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

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

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

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

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

3) Invoke the debugger as usual.

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

   This redefines the memory map for the current debugging session.

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

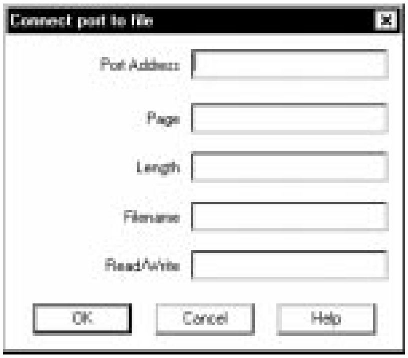
4.9 Simulating I/O Space (Simulator Only)

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

Connecting an I/O port

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

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



2) In the Port Address field, enter the address where you want to simulate an I/O port. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.

3) In the Page field, enter a one-digit number that identifies the type of memory (program or data) that the address occupies:

To identify this page . . .	Use this value as the *page* parameter . . .
Program memory	**0**
Data memory	**1**

4) In the Length field, enter the length of the memory range. The length can be any C expression.

5) If you are connecting a port to be read from a file, in the Filename field, enter the name of the file to which you want to connect. If you connect a port to read from a file, the file must exist, or the MC command will fail.

6) In the Read/Write field, enter how the file will be used (for input or output, respectively). The keywords for the read/write characteristics are available on page 4-11.

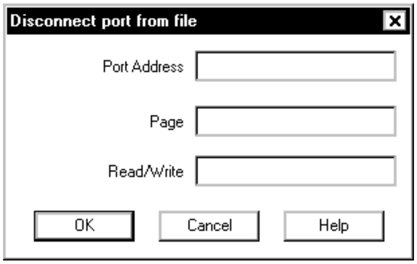
7) Click OK.

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

Disconnecting an I/O port

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

1) In the command line, enter **mi**. This displays the Disconnect port from file dialog box:



2) In the Port Address field, enter I/O port memory address that is to be closed.

3) In the Page field, enter the one-digit number that identifies the type of memory (program or data) that the address occupies.

4) In the Read/Write field, enter the characteristic used when the port was connected.

5) Click OK.

4.10 Simulating External Interrupts (Simulator Only)

The ’C54x simulator allows you to simulate the external interrupts signals $\overline{\text{INT}}$0–$\overline{\text{INT}}$3 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 4 interrupt pins, $\overline{\text{INT}}$0–$\overline{\text{INT}}$3, 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

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 where you want an interrupt to occur.

  You can have two types of CPU clock cycles:

  - **Absolute**. To use an absolute clock cycle, your cycle value must represent the actual CPU clock cycle 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. No operation is performed on the clock cycle value; the interrupt occurs exactly as the clock cycle value is written.

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

    ```
    12 +34 55
    ```

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

- The *logic value* parameter is valid 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] [56,0] [78,1]
  ```

This causes the $\overline{\text{BIO}}$ pin to go high at the 12th cycle, low at the 56th cycle, and high again at the 78th 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 for a fixed number of times. For example:

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

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

    The n is a positive integer value.

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

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

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

Connecting your input file to the interrupt pin

To connect your input file to the interrupt pin, use the PINC command. The syntax for this command is:

pinc *pinname, filename*

- The *pinname* identifies the input pin and must be one of the interrupt pins: $\overline{\text{INT}}$0–$\overline{\text{INT}}$3 or and the $\overline{\text{BIO}}$ pin.
- The *filename* is the name of your input file.

Example 4–2 shows you how to connect your input file using the PINC command.

Example 4–2. Connecting the Input File With the PINC Command

Suppose you want to generate an external interrupt on INT2 at the 12th, 34th, 56th, and 89th clock cycles.

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

```
12 34 56 89
```

To connect the input file to the pin, enter:

```
pinc INT2, myfile                          Connects your data file
                                         to the specific interrupt pin
```

This command connects myfile to theINT2 pin. As a result, the simulator generates an external interrupt on INT2at the 12th, 34th, 56th, and 89th clock cycles.

Disconnecting your input file from the interrupt pin

To end the interrupt simulation, use the PIND command to disconnect the pin. The syntax for this command is:

pind *pinname*

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

Listing the interrupt pins and connecting input files

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

pinl

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

4.11 Simulating Peripherals (Simulator Only)

With the ’C54x simulator, you can simulate the timer, standard serial port, buffered serial port, and TDM serial port. The peripherals simulated depend upon the device that you simulate. You simulate a device by starting the simulator with the appropriate option. Table 4–1 summarizes the options used to simulate the peripherals for each device.

Table 4–1. Debugger options for the Simulator

Option	Device Simulated	Init File	Peripherals Simulated
–mv540	’C540	sim540.cmd	Serial port 0, serial port 1, timer
–mv541	’C541	sim541.cmd	Serial port 0, serial port 1, timer
–mv542	’C542	sim542.cmd	Buffered serial port, TDM serial port, timer, HPI
–mv543	’C543	sim543.cmd	Buffered serial port, TDM serial port, timer
–mv544	’C544	sim544.cmd	Serial port 0, serial port 1, timer
–mv545	’C545	sim545.cmd	Buffered serial port, serial port 1, timer, HPI
–mv545lp	’C545lp	sim545lp.cmd	Buffered serial port, serial port 1, timer, HPI
–mv546	’C546	sim546.cmd	Buffered serial port, serial port 1, timer
–mv547	’C547	sim547.cmd	Buffered serial port, serial port 1, timer, HPI
–mv548	’C548	sim548.cmd	Buffered serial port 1, buffered serial port 2, TDM serial port, timer, HPI
–mv549	’C549	sim549.cmd	Buffered serial port 1, buffered serial port 2, TDM serial port, timer, HPI

Note: If the –mv option is not specified, ’C540 is simulated.

Detailed information about simulating the different peripherals is discussed in the following sections:

Type of Peripheral	See Section. . .
Standard	4.12 on page 4-26
Buffered	4.13 on page 4-30
TDM	4.14 on page 4-34
HPI	4.15 on page 4-37

4.12 Simulating Standard Serial Ports (Simulator Only)

The ’C54x simulator supports standard serial port transmission and reception by reading data from, and writing data to, the files associated with the DXR/TDXR and DRR/TDRR registers, respectively.

The simulator also provides limited support for the simulation of the serial port control pins (frame synchronization pins) with the help of external event simulation capability. Frame synchronization pin values for receive and transmit operations are, at various instants of time, fed through the files associated with the pins.

The ’C54x simulator supports the following operations in the standard serial port simulation:

- Internal clocks (1/4 CPU clock) and external clocks for the transmit and receive operations. External clocks are simulated by using the Divide command in the files connected to the FSX/TFSX and FSR/TFSR pins.
- External frame synchronization pulses (FSX/TFSX transmit and FSR/TFSR receive frame synchronization pulses). Transmit and receive operations are initiated when the signals for these values go high.
- The operations associated with the following memory-mapped registers:

Register	Memory	Bits used	Description
SPC	0x22	FO	Format specifier (8/16 bits)
TSPC	0x32	MCM XRST/RRST XRDY/RRDY XSREMPTY RSRFULL	Internal/external clock Transmit/receive reset Transmit/receive ready Transmit register empty flag Receive register full flag
DXR	0x21	All bits are used	Transmit data register
TDXR	0x31		
DRR	0x20	All bits are used	Receive data register
TDRR	0x30		

Setting up your transmit and receive operations

The ’C54x simulator supports the simulation of the following pins using external event simulation. The pulses occurring on the FSX and FSR pins initiate the standard serial port transmit and receive operations, respectively.

- ❑ FSR/TFSR – Frame synchronization pulses for the receive operation
- ❑ FSX/TFSX – Frame synchronization pulses for the transmit operation

Connect the files to the pins using the PINC (pin connect) command. Use the following command syntax, selecting the appropriate command for the pin you want:

```
pinc FSX, filename
pinc TFSX, filename
pinc FSR, filename
pinc TFSR, filename
```

filename is the name of the file that contains the CPU cycles at which the pin value goes high. Use the following syntax in the files to define clock cycles:

[*clock cycle*] **rpt** {n **|** **EOS**}

For more information about defining clock cycles, see section 4.10, on page 4-22.

Additionally, you can use the Divide command to specify the clock divide ration for the device. Use the following syntax in the files for the Divide command:

DIVIDE *r*

r is a real number or integer specifying the ratio of serial port clock versus the CPU clock. Use the divide ration when the serial port is configured to use the external clock. When you use the Divide command, it must be the first command in the file.

The following example specifies the clock ratio of the transmit clock and the clock cycles for the occurrences of TFSX pulses (if this file is connected to the TFSX pin):

```
DIVIDE 5
100  +200  +100
```

The Divide command specifies the divide-down ratio of the clock against the CPU clock. That is, the CLKX frequency is 1/5 of the CPU clock. The second line indicates that the TFSX should go high at the 100th, 300th (100+200), and the 400th (300+100) CPU cycles. The TFSX pin goes to high in the 20th, 60th, and 80th cycles of the serial port clock.

Connecting input/output files

Input and output files are connected to DRR/TDRR and DXR/TDXR registers for receive and transmit operations, respectively. To simulate the transmit operation, data is written to the file that is connected to the DXR/TDXR register. To simulate the receive operation, data is read from the file that is connected to the DRR/TDRR register.

The input and output file formats for the standard serial port operation requires at least one line containing an hexadecimal number. The following is an acceptable format for an input file:

```
0055
aa55
efef
dead
```

Note:

To simulate the standard serial port 0, use the DXR and DRR registers and the FSX and FSR pins. To simulate the standard serial port 1, use the TDXR and TDRR registers and the TFSX and TFSR pins.

Programming the simulator

To simulate the standard serial port, configure the DXR/TDXR and DRR/TDRR registers as the output port (OPORT) and the input port (IPORT), respectively. Connect these ports to an output file and an input file. Also, connect files to the TFSX/FSX and TFSR/FSR pins to specify the clock cycles during which the frame synchronization pins go high.

To make these connections, use the following commands in the simulator initialization batch file (siminit.cmd):

```
ma DRR, 1, 1, R|P
ma DXR, 1, 1, W|P

mc DRR, 1, 1, receive filename, READ
mc DXR, 1, 1, transmit filename, WRITE

pinc FSX, fsx timing filename
pinc FSR, fsr timing filename
```

Variable	Description
receive filename	The file to read data from, which simulates the input port
transmit filename	The file to write data to, which simulates the output port
fsx timing filename	The file that contains the CPU cycles at which the FSX frame synchronization pin goes high
fsr timing filename	The file that contains the CPU cycles at which the FSR frame synchronization pin goes high

4.13 Simulating Buffered Serial Ports (Simulator Only)

The ’C54x simulator supports buffered serial port transmission and reception by reading data from and writing data to the files associated with the DXR and DRR registers, respectively.

The simulator also provides limited support for the simulation of the serial port control pins (frame synchronization pins) with the help of external event simulation capability. Frame synchronization pin values for receive and transmit operations are, at various instants of time, fed through the files associated with the pins. The ’C54x simulator supports the following operations in the standard serial port simulation:

❑ Automatic buffering and standard serial port modes

❑ Internal clocks (1/(CLKDV + 1) CPU clock) and external clocks for the transmit and receive operations.

❑ External frame synchronization pulses (FSX transmit and FSR receive frame synchronization pulses). Transmit and receive operations are initiated when the signals for these values go high.

❑ The operations associated with the following memory-mapped registers:

Register	Memory	Bits used	Description
SPC	0x22	FO MCM XRST/RRST XRDY/RRDY XSREMPTY RSRFULL	Format specifier (8/16 bits) Internal/external clock Transmit/receive reset Transmit/receive ready Transmit register empty flag Receive register full flag
DXR	0x21	All bits are used	Transmit data register
DXR1	0x41	All bits are used	Transmit data register
DRR	0x20	All bits are used	Receive data register
DRR1	0x40	All bits are used	Receive data register
SPCE	0x23	CLKDV FE RH/TH BXE/BRE HALTX/HALTR	Clock divide ratio Extended format specifier Buffer half received or transmitted Enable/disable automatic buffering Switch to standalone mode after the current half is transmitted/received
AXR	0x38	All bits are used	Address register for transmit
AXR1	0x3c	All bits are used	Address register for transmit

Register	Memory	Bits used	Description
ARR	ox3a	All bits are used	Address register for receive
ARR1	0x3e	All bits are used	Address register for receive
BKX	0x39	All bits are used	Block size register for transmit
BKX1	0x3d	All bits are used	Block size register for transmit
BKR	0x3b	All bits are used	Block size register for receive
BKR1	0x3f	All bits are used	Block size register for receive

Setting up your transmit and receive operations

The ’C54x simulator supports the simulation of the following pins using external event simulation. The pulses occurring on the FSX and FSX1, and FSR and FSR1 pins initiate the standard serial port transmit and receive operations, respectively.

❑ FSR – Frame synchronization pulses for the receive operation
❑ FSR1 – Frame synchronization pulses for the receive operation
❑ FSX – Frame synchronization pulses for the transmit operation
❑ FSX1 – Frame synchronization pulses for the transmit operation
❑ Connect the files to the pins using the PINC (pin connect) command. Use the following command syntax, selecting the appropriate command for the pin you want:

```
pinc FSX, filename
pinc FSX1, filename
pinc FSR, filename
pinc FSR1, filename
```

filename is the name of the file that contains the CPU cycles at which the pin value goes high. Use the following syntax in the files to define clock cycles:

[*clock cycle*] **rpt** {n **|** **EOS**}

For more information about defining clock cycles, see section 4.10, on page 4-22.

Additionally, you can use the Divide command to specify the clock divide ration for the device. Use the following syntax in the files for the Divide command:

DIVIDE *r*

r is a real number or integer specifying the ratio of serial port clock versus the CPU clock. Use the divide ration when the serial port is configured to use the external clock. When you use the Divide command, it must be the first command in the file.

The following example specifies the clock ratio of the transmit clock and the clock cycles for the occurrences of FSX pulses (if this file is connected to the FSX pin):

```
DIVIDE 5
100  +200  +100
```

The Divide command specifies the divide-down ratio of the clock against the CPU clock. That is, the CLKX frequency is 1/5 of the CPU clock. The second line indicates that the TFSX should go high at the 100th, 300th (100+200), and the 400th (300+100) CPU cycles. The TFSX pin goes to high in the 20th, 60th, and 80th cycles of the serial port clock.

Connecting input/output files

Input and output files are connected to DRR/DRR1 and DXR/DXR1 registers for receive and transmit operations, respectively. To simulate the transmit operation, data is written to the file that is connected to the DXR register. To simulate the receive operation, data is read from the file that is connected to the DRR register.

The input and output file formats for the buffered serial port operation requires at least one line containing an hexadecimal number. The following example shows an acceptable format for an input file:

```
0055
aa55
efef
dead
```

Note:

To simulate the buffered serial port 0, use the DXR, DRR, AXR, ARR, BKY, and BKR registers and the FSX and FSR pins. To simulator the buffered serial port 2, use DXR1, DRR1, AXR1, ARR1, BKX1, and BKR1 registers and the FSX1 and FSR1 pins.

Programming the simulator

To simulate the standard serial port, configure the DXR and DRR registers as the output port (OPORT) and the input port (IPORT), respectively. Connect these ports to an output file and an input file. Also, connect files to the TFSX/FSX and TFSR/FSR pins to specify the clock cycles during which the frame synchronization pins go high.

To make these connections, use the following commands in the simulator initialization batch file (siminit.cmd):

```
ma DRR, 1, 1, R|P
ma DXR, 1, 1, W|P

mc DRR, 1, 1, receive filename, READ
mc DXR, 1, 1, transmit filename, WRITE

pinc FSX, fsx timing filename
pinc FSR, fsr timing filename
```

Variable	Description
receive filename	The file to read data from, which simulates the input port
transmit filename	The file to write data to, which simulates the output port
fsx timing filename	The file that contains the CPU cycles at which the FSX frame synchronization pin goes high
fsr timing filename	The file that contains the CPU cycles at which the FSR frame synchronization pin goes high

4.14 Simulating TDM Serial Ports (Simulator Only)

The ’C54x simulator supports TDM serial port transmission and reception by reading data from and writing data to the files associated with the TDXR and TDRR registers, respectively.

The simulator also provides limited support for the simulation of the TDM port control pins (frame synchronization pins) with the help of external event simulation capability. Frame synchronization pin values for receive and transmit operations are, at various instants of time, fed through the files associated with the pins.

The ’C54x simulator supports the following operations in the standard serial port simulation:

❑ TDM and standard serial port modes

❑ Internal clocks (1/4 CPU clock) and external clocks for the transmit and receive operations. External clocks are simulated by using the Divide command in the files connected to the TFSX and TFSR pins.

❑ External frame synchronization pulses (FSX transmit and FSR receive frame synchronization pulses). Transmit and receive operations are initiated when the signals for these values go high.

❑ The operations associated with the following memory-mapped registers:

Register	Memory	Bits used	Description
TSPC	0x32	TDM MCM XRST/RRST XRDY/RRDY XSREMPTY RSRFULL	Multiprocessor/normal mode Internal/external clock Transmit/receive reset Transmit/receive ready Transmit register empty flag Receive register full flag
TCSR	0x33	All bits are used	Channel select register
TRTA	0x34	All bits are used	Receive/transmit address register
TRAD	0x35	All bits are used	Receive address register
TDXR	0x31	All bits are used	Transmit data register
TDRR	0x30	All bits are used	Receive data register

Setting up your transmit and receive operations

The ’C54x simulator supports the simulation of the following pins using external event simulation. The pulses occurring on the TFSX and TFSR pins initiate the standard serial port transmit and receive operations, respectively.

❑ TFSR – Frame synchronization pulses for the receive operation

❑ TFSX – Frame synchronization pulses for the transmit operation

Connect the files to the pins using the PINC (pin connect) command. Use the following command syntax, selecting the appropriate command for the pin you want:

```
pinc TFSX, filename
pinc TFSR, filename
```

filename is the name of the file that contains the CPU cycles at which the pin value goes high. Use the following syntax in the files to define clock cycles:

[*clock cycle*] **rpt** {n **|** **EOS**}

For more information about defining clock cycles, see section 4.10, on page 4-22.

Additionally, you can use the Divide command to specify the clock divide ration for the device. Use the following syntax in the files for the Divide command:

DIVIDE *r*

r is a real number or integer specifying the ratio of serial port clock versus the CPU clock. Use the divide ration when the serial port is configured to use the external clock. When you use the Divide command, it must be the first command in the file.

The following example specifies the clock ratio of the transmit clock and the clock cycles for the occurrences of TFSX pulses (if this file is connected to the TFSX pin):

```
DIVIDE 5
100  +200  +100
```

The Divide command specifies the divide-down ratio of the clock against the CPU clock. That is, the CLKX frequency is 1/5 of the CPU clock. The second line indicates that the TFSX should go high at the 100th, 300th (100+200), and the 400th (300+100) CPU cycles. The TFSX pin goes to high in the 20th, 60th, and 80th cycles of the serial port clock.

Connecting input/output files

Input and output files are connected to TDRR and TDXR registers for receive and transmit operations, respectively. To simulate the transmit operation, data is written to the file that is connected to the DXR register. To simulate the receive operation, data is read from the file that is connected to the TDRR register. Use the following syntax to create the files:

channel-address data

channel–address specifies the TDM channel in which transmission/reception takes place. *data* specifies the value that is written or read from the file. Each field is in hexadecimal format separated by spaces. The following is an acceptable format for an input file:

```
0055
aa55
efef
dead
```

Programming the simulator

To simulate the standard serial port, configure the DXR and DRR registers as the output port (OPORT) and the input port (IPORT), respectively. Connect these ports to an output file and an input file. Also, connect files to the TFSX/ FSX and TFSR/FSR pins to specify the clock cycles during which the frame synchronization pins go high.

To make these connections, use the following commands in the simulator initialization batch file (siminit.cmd):

```
ma TDRR, 1, 1, R|P
ma TDXR, 1, 1, W|P

mc TDRR, 1, 1, receive filename, READ
mc TDXR, 1, 1, transmit filename, WRITE

pinc TFSX, fsx timing filename
pinc TFSR, fsr timing filename
```

Variable	Description
receive filename	The file to read data from, which simulates the input port
transmit filename	The file to write data to, which simulates the output port
fsx timing filename	The file that contains the CPU cycles at which the FSX frame synchronization pin goes high
fsr timing filename	The file that contains the CPU cycles at which the FSR frame synchronization pin goes high

4.15 Simulating Host Port Interfaces (Simulator Only)

The ’C54x simulator provides support for the simulation of host port interfaces (HPI). HPI simulation is available in ’C542, ’C547, ’C548, ’C549, and ’C545LP. This simulation is performed using files which specify the value of the control signals and the corresponding address and data values.

When simulating HPI, two files are associated with HPI, one for specifying the input values, and one for storing the output values. The outputs generated by the HPI simulation are stored in the output file called hpi.out; the name of this file cannot be changed.

Setting up your input file

To simulate host port interfaces, you must first set up an input file that lists the HPI function and corresponding data. Entries made in the input file are usually entered using the following format:

clock cycle function-code [*address*] [*data*] *bit_number* [**rpt**{*n* **| EOS**}]:

(The colon should be used as a delimiter after every statement, as shown above.)

- ❑ The *clock cycle* parameter specifies the clock cycle during which the HPI function is to be performed. The clock values can be specified in decimal format (using 0x or 0X prefix format) or in hexadecimal format (using h or H suffix format).

  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 where you want to simulate an interrupt. For example:

    ```
    12 34 56
    ```

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

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

    ```
    12 +34 55
    ```

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

- ❑ The *function-code* parameter specifies the type of HPI operation to be performed. The *function-code* parameter can be any one of the following values:

  - ■ DATA_READ (data read). This parameter value reads the specified byte (LSB/MSB) from the address specified in the HPIA register. The presence of the ‘++’ indicates post increment of the HPIA register. The syntaxes are:

    clock cycle DATA_READ LSB [++]:
    clock cycle DATA_READ MSB [++]:

  - ■ DATA_WRITE (data write). This parameter value writes data to the specified byte (LSB/MSB) at the address specified in the HPIA register. The presence of the ‘++’ indicates the preincrement of the HPIA register. The syntaxes are:

    clock cycle DATA_WRITE [*data*] [++] LSB :
    clock cycle DATA_WRITE [*data*] [++] MSB :

  - ■ CTRL_READ (control register read). This parameter value reads the specified byte (LSB/MSB) from the control register HPIC. The syntaxes are:

    clock cycle CTRL_READ LSB :
    clock cycle CTRL_READ MSB :

  - ■ CTRL_WRITE (control register write). This parameter value writes the specified byte (LSB/MSB) of control register HPIC. The syntaxes are:

    clock cycle CTRL_WRITE [*data*] LSB :
    clock cycle CTRL_WRITE [*data*] MSB :

  - ■ LOAD (initialize HPI RAM). This parameter value initializes the HPI RAM (starting from address 0x1000) from the specified file. The syntax is:

    clock cycle LOAD [*filename*] :

    (*filename*, for this function, is a specific form of the *data* parameter.)

  - ■ HPIA_WRITE (HPIA register write). This parameter value writes the specified byte to the HPIA register. To write 10-bit address into HPIA, 8 bits are required to write at LSB, and the other 2 bits are required to write at MSB. The syntaxes are:

    clock cycle HPIA_WRITE [*address*] LSB :
    clock cycle HPIA_WRITE [*address*] MSB :

- ❑ The *address* parameter represents a 10-bit address used to specify the values of address pins during a data read or write operation from the host. This parameter may or may not be necessary, depending on the function code.

- ❑ The *data* parameter represents an 8-bit data field used to specify the value of the data pins during a control or data write. Data can be specified either in decimal or in hexadecimal format. This parameter may or may not be necessary, depending on the function code.
- ❑ The *bit_number* parameter represents the bit number being manipulated.
- ❑ The **rpt**{n **|** **EOS**} parameter is optional and represents a repetition value.

  You can have two forms of repetition to simulate interrupts:

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

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

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

    The n is a positive integer value.

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

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

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

Connecting your input file to the interrupt pin

To connect your input file to the interrupt pin, use the PINC command. The syntax for this command is:

pinc *HPI*, *filename*

- ❑ *HPI* is the name of the interrupt pin to which the file will be connected.
- ❑ The *filename* is the name of your input file.

Disconnecting your input file from the interrupt pin

To end the HPI simulation, use the PIND command to disconnect the pin. The syntax for this command is:

pind *HPI*

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

Chapter 5

Using the Debugger With Extended Addressing

The TMS320C54x is limited to 64K bytes of address space in program, data, and I/O space. Certain ’C54x processors are designed to support a much larger program space address reach than 64K bytes. For the processors that support extended addressing, the program address bus (PAB) has been extended to 23 bits and can support program memory of 128 pages of 64K words.

If you use extended addressing, the debugger supports access to instructions and data stored in extended memory.

This chapter defines extended addressing and describes what you need to do in your debugger to enable extended addressing for the ’C54x.

Topic **Page**

5.1 Understanding the Use of Extended Addressing

With the extended addressing feature, you have the ability to add additional memory to your target system.

About extended addressing

Once you modify your target system to include extended memory, you can use the debugger to access the extended memory. To determine which memory location you want to access, the debugger must have a unique address.

In an extended memory system, you logically split the 16-bit address space into two pieces. The low-ordered addresses are common or *unmapped* memory. The high-ordered addresses are extended or *mapped* memory. The address that defines the boundary between unmapped and mapped memory, the *mapped start address*, is defined based on the external registers and memory that you add to your target system.

You can add and define multiple banks of physical memory that overlay each other in a single, mapped address range. The use of extended pages of memory extends the native 16-bit address space. Because the 16-bit address space is extended, the debugger uses addresses that are 23 bits wide. The 23-bit address is composed of the following:

❑ The 7 most significant bits (MSBs) represent the extended page number. In your source code, you store this number in the extended program mapping register (XPC). The XPC stores the extended page number and is located at 0x1E in data space. The value that you use in the XPC is the same extended-page number that you use in the linker command file.

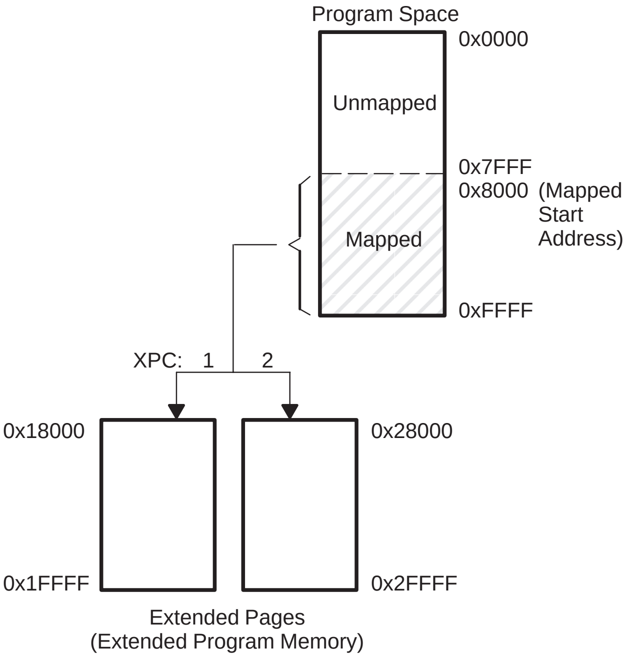
❑ The 16 least significant bits (LSBs) represent the native 16-bit address for the symbol.

Sample extended memory system

Figure 5–1 illustrates a sample extended memory system for TMS320LC548. In this example, the program space is allocated in this manner:

- ❑ The unmapped memory region is from 0x0000 to 0x7FFF.
- ❑ The mapped memory region is from 0x8000 to 0xFFFF.
- ❑ Two extended pages extend the mapped address space.
- ❑ The XPC register qualifies the mapped address, creating a unique address for each location in each extended page.

Figure 5–1. Sample Extended Memory System for the TMS320LC548



5.2 Setting Up Extended Addressing

Before you can use extended memory, you must build a target system that includes extended memory.

Once you have built your extended memory system, you must set up the debugger to use extended addressing by doing the following:

- ❑ Describe your extended memory configuration to the debugger
- ❑ Enable extended addressing
- ❑ Use the MA commands to add the extended memory ranges into the map

This section describes how to perform these tasks for both the emulator and the simulator.

Describing your extended memory configuration to the debugger (emulator)

You must describe to the debugger your extended memory configuration and where to look for the mapper register (XPC). To do so, use the EXT_ADDR_DEF command. The syntax for this command is:

ext_addr_def *map start* [**@prog**] , *reg addr* [**@data**], *mask*

- ❑ The *map start* parameter defines the beginning of the mapped memory range. By default, the *map start* parameter is treated as a program-memory address.

  The map start address is determined by the OVLY bit:

  - ■ If OVLY is 1, map start is 0x8000@prog.
  - ■ If OVLY is 0, map start is 0x0000@prog.

- ❑ The *reg addr* parameter defines the location of the mapping register (XPC). Use the address 0x1E@data.

  By default, the *reg addr* parameter is treated as a data-memory address.

- ❑ The *mask* parameter must be 0x7F since the program address bus (PAB) is 23 bits wide.

For example, when you design an extended memory system where OVLY is 1, you would enter:

```
ext_addr_def 0x8000@prog, 0x001E@data, 0x7f
```

To avoid entering the EXT_ADDR_DEF command each time you invoke the debugger, you can modify your emuinit.cmd file to include the command. The debugger reads and executes the commands in the emuinit.cmd file each time you invoke the debugger.

Describing your extended memory configuration to the debugger (simulator)

You must describe to the debugger your extended memory configuration and where to look for the mapper register (XPC). To do so, use the EXT_ADDR_DEF command. The syntax for this command is:

ext_addr_def *mapped mem start*[@**page**], *mr addr*[@**data**], *mapper mask*

- ❑ The *mapped mem start* parameter defines the beginning of the mapped memory range. By default, the *mapped mem start* parameter is treated as a program-memory address.

  The mapped mem start address is determined by the OVLY bit:

  - ■ If OVLY is 1, mapped mem start is 0x8000@prog.
  - ■ If OVLY is 0, mapped mem start is 0x0000@prog.

- ❑ The *mr addr* parameter defines the location of the mapping register (XPC). Use the address 0x1E@data.

  By default, the *mr addr* parameter is treated as a data-memory address.

- ❑ The *mapper mask* parameter must be 0x7F since the program address bus (PAB) is 23 bits wide.

For example, when you design an extended memory system where OVLY is 1, you would enter:

```
ext_addr_def 0x8000@prog, 0x001E@data, 0x7f
```

To avoid entering the EXT_ADDR_DEF command each time you invoke the debugger, you can modify your siminit.cmd file to include the command. The debugger read and executes the command in the siminit.cmd file each time you invoke the debugger.

Note:

The EXT_ADDR_DEF should only come after the *mapped mem start* and the *mr addr* have been added to the enabled map through earlier MA commands. Otherwise, you see will see the debugger display an invalid address message. In the above example, if 0x8000 in prog is not in the mapped memory, or if 0x1e is not in the mapped data memory, the EXT_ADDR_DEF command will fail.

Enabling extended addressing

Before you load your target code, you must enable extended addressing. To do so, use the EXT_ADDR ON command. The syntax for this command is:

ext_addr {**on** | **off**}

You cannot use this command before you define your extended memory configuration with the EXT_ADDR_DEF command.

To avoid entering the EXT_ADDR command each time you invoke the debugger, you can modify your emuinit.cmd file and/or your siminit.cmd file to include the command.

Mapping the extended memory

Once you have configured the extended memory and enabled extended memory addressing, you can use the MA command to add the extended memory ranges to the memory map.

An example of adding an extended memory range to a memory map is:

```
ma 0x18000, 0, 0x8000, R|W|EX                     ;External
```

5.3 Debugging With Extended Addressing

Once you have set up the debugger for extended addressing (as described in Section 5.2, *Setting Up Extended Memory*, on page 5-4), you can use the debugger to access code stored in the extended memory of your system.

When the debugger loader loads a section of code that contains extended addresses, the loader places the addresses in the proper overlays automatically. The debugger also changes the display of the Disassembly or Memory window to show extended addresses. It uses the XPC value (which contains the extended page number) as a prefix to the address.

Note:

The extended-page number that is shown in the Disassembly or Memory window does not always represent the value currently stored in the XPC. However, while stepping through the code, the register value and the extended-page number are the same.

Registers associated with extended addressing: XPC and EPC

When you turn extended addressing on (using the EXT_ADDR ON command), the debugger adds the following registers to the CPU window:

❑ The extended program page register (XPC). The XPC displays the current value of the XPC that is in your target system.

❑ The extended program counter (EPC). The EPC is 32 bits wide and represents the XPC value concatenated with the PC value.

You can use these registers in expressions. For example, this command sets the XPC to 4 and the program counter (PC) to 0x8000:

```
? EPC= 0x48000
```

This DASM command causes the Disassembly window to display the current extended PC location:

```
DASM EPC
```

When you turn extended addressing off (using the EXT_ADDR OFF command), the CPU window no longer displays the XPC and EPC registers, and you cannot use these registers in expressions.

New expression syntax

When you enter one of the commands listed in Table 5–1, you can use an address as one of the command parameters. You can append a suffix to the address to specify that the address is in program memory (@prog). When extended addressing is enabled, you can use a new suffix with the commands in Table 5–1:

- ❑ The @prog16 suffix identifies an address in extended program memory. The debugger uses the current value in the XPC to qualify the address that you enter.

Table 5–1. Commands That Use the @prog16 Suffixes

Command	**See Page**	**Command**	**See Page**
? (evaluate expression)	13-10	EVAL	13-22
ADDR	13-12	MEM	13-37
DASM	13-18	WA	13-64
DISP	13-18		

How extended addressing affects symbols

When you use the debugger with symbol names, the debugger uses the entire extended address associated with the symbol.

When you use a pointer in an expression, the debugger uses the current XPC value to determine the pointer value. Likewise, if you use the contents of a register as an address, the debugger uses the current XPC value to determine the correct address. For example, if the PC=0x8000 and the XPC=2 and you enter:

```
? *PC
```

The debugger returns the value at location 0x28000.

Table 5–2 shows typical commands that you could use with the debugger and how the results of the commands are affected by extended addressing.

Table 5–2. Sample Commands and Results Using Extended Addressing

If you enter this...	The result is...
`dasm 0x8000`	Disassembly starting at 0:8000, regardless of the XPC value
`dasm 0x8000@prog16`	Disassembly starting at *x*:8000, where *x* represents the current XPC value
`dasm label0`	Where label0 is a label located at 0x8000, the result is disassembly starting at 0:8000, regardless of the XPC value
`dasm label4`	Where label4 is a label located at 0x48000, the result is disassembly starting at 4:8000, regardless of the XPC value
`dasm pc`	Where PC=0x8000, the result is disassembly starting at *x*:8000, where *x* represents the current XPC value
`dasm ptr`	Where ptr is a VOID* pointing to 0x8000, the result is disassembly starting at *x*:8000, where *x* represents the current XPC value
`dasm (ptr+1)`	Where ptr is a VOID* pointing to 0x8000, the result is disassembly starting at *x*:8001, where *x* represents the current XPC value

Note:

In the Disassembly window, addresses associated with symbols are shown as 16-bit addresses. Moreover, if a symbol represents an extended address, you cannot see the entire 23-bit address in the Disassembly window. However, when you use the symbol name in an expression, the debugger accesses the correct address.

Using 16-bit expressions with 23-bit extended addressing

Extended addressing allows you to use 23-bit addresses to reference locations in extended memory. When you use the debugger and specify a 16-bit expression, the debugger uses the following algorithm to determine which memory location to access:

1) If you use the @prog16 suffix, the debugger uses the current XPC value as a prefix to the address that you entered.

2) If you use a pointer in an expression, the debugger uses the current XPC value as a prefix to the pointer value, as applicable.

3) If you use the contents of a register as an address, the debugger uses the XPC value as a prefix to the address, as applicable.

4) If none of the above is true, the debugger uses 0 as a prefix to the address. This applies to constants and program labels.

Chapter 6

Loading and Displaying Code

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

Topic **Page**

6.1 Loading and Displaying Assembly Language Code

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

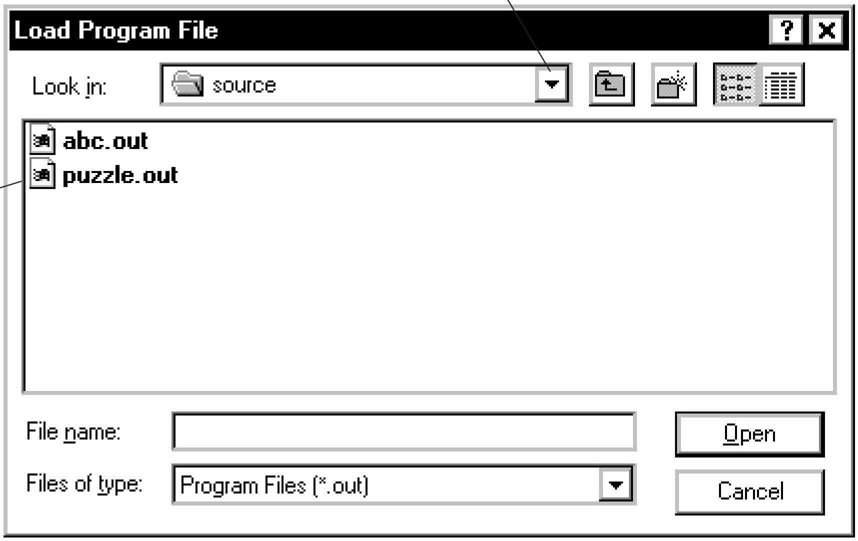
After you invoke the debugger, you can load object code and/or the symbol table associated with an object file.

Loading an object file and its symbol table

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

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

You can change the directory that you want to search.



Select from a list of files.

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

3) Click Open.

Loading an object file without its symbol table

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

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

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

reload *object filename*

Loading a symbol table only

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

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

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

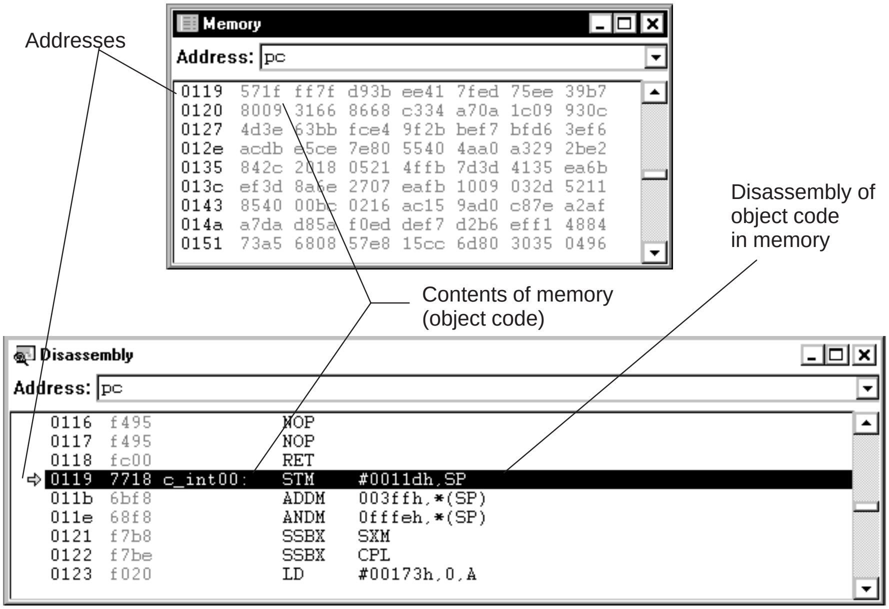
Loading code while invoking the debugger

You can load an object file when you invoke the debugger. (This has the same effect as using the File→Load→Load Program menu option described on page 6-2.) To do this, enter the appropriate debugger-invocation command along with the name of the object file.

If you want to load only a file’s symbol table when you invoke the debugger, use the –s option. (This option has the same effect as using the File→Load→Program Symbols menu option.) To do this, enter the appropriate debugger-invocation command along with the name of the object file and specify –s (see page 2-14 for more information).

Displaying portions of disassembly

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



```
0119  571f ff7f d93b ee41 7fed 75ee 39b7
0120  8009 3166 8668 c334 a70a 1c09 930c
0127  4d3e 63bb fce4 9f2b bef7 bfd6 3ef6
012e  acdb e5ce 7e80 5540 4aa0 a329 2be2
0135  842c 2018 0521 4ffb 7d3d 4135 ea6b
013c  ef3d 8a6e 2707 eafb 1009 032d 5211
0143  8540 00bc 0216 ac15 9ad0 c87e a2af
014a  a7da d85a f0ed def7 d2b6 eff1 4884
0151  73a5 6808 57e8 15cc 6d80 3035 0496
```

```
   0116 f495           NOP
   0117 f495           NOP
   0118 fc00           RET
⇨  0119 7718 c_int00:  STM     #0011dh,SP
   011b 6bf8           ADDM    003ffh,*(SP)
   011e 68f8           ANDM    0fffeh,*(SP)
   0121 f7b8           SSBX    SXM
   0122 f7be           SSBX    CPL
   0123 f020           LD      #00173h,0,A
```

Addresses

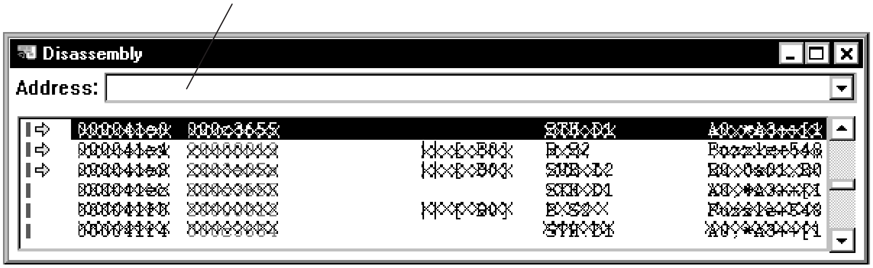
Disassembly of
object code
in memory

Contents of memory
(object code)

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

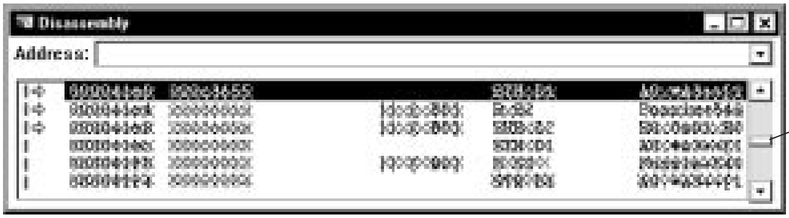
To display code beginning at a specific point, enter a new starting address or a symbol name in the Address field of the Disassembly window:

Enter an address here to display
code at a specific point.



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

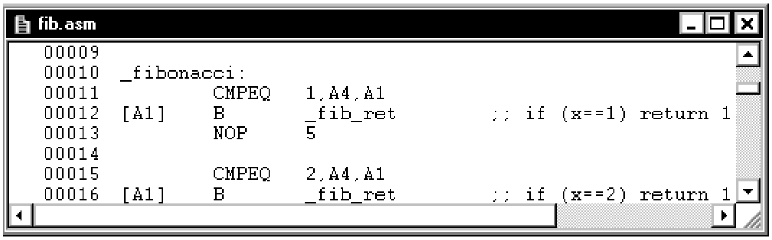
You can also move through the contents of the Disassembly window by using the scroll bar. Because the Disassembly window shows the reverse assembly of memory contents, the scroll-bar handle is displayed in the middle of the scroll bar. The middle of the reverse assembly is defined as the most recent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



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

Displaying assembly source code

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



```
00009
00010  _fibonacci:
00011          CMPEQ   1,A4,A1
00012  [A1]    B       _fib_ret        ;; if (x==1) return 1
00013          NOP     5
00014
00015          CMPEQ   2,A4,A1
00016  [A1]    B       _fib_ret        ;; if (x==2) return 1
```

6.2 Displaying C Code

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

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

Displaying the contents of a text file

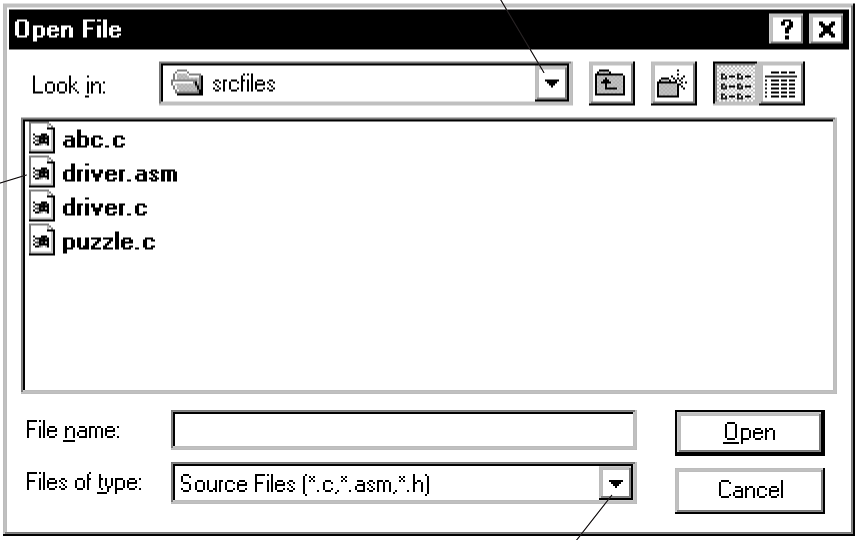
To display the contents of any text file, follow these steps:

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

   - ❑ Click the Open icon on the toolbar:
   - ❑ From the File menu, select Open→Source File.

   This displays the Open File dialog box:

You can change the directory that you want to search.



Look in: srcfiles

abc.c
driver.asm
driver.c
puzzle.c

File name:

Files of type: Source Files (*.c,*.asm,*.h)

Select from a list of files.

Select the type of file you want to open.

2) Select the file that you want to open. To do so, you might need to do one or more of the following actions:

   - ❑ Change the working directory.
   - ❑ Select the type of file that you want to open (for example, .c, .h).

3) Click Open.

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

Displaying a C file *does not* load that file's object code. If you want to be able to run the program, you must load the file's associated object code as described in section 6.1, *Loading and Displaying Assembly Language Code*, on page 6-2.

Displaying a specific C function

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

func {*function name* **|** *address*}

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

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

- ❑ Single-click the name of the C function.
- ❑ Select the name of the C function and press F9 .

Displaying code beginning at a specific point

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

addr {*address* | *function name*}

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

Chapter 7

Running Code

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

❏ Basic *run commands* run your code without updating the display until you explicitly halt execution.

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

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

Topic **Page**

7.1 Defining the Starting Point for Program Execution

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

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

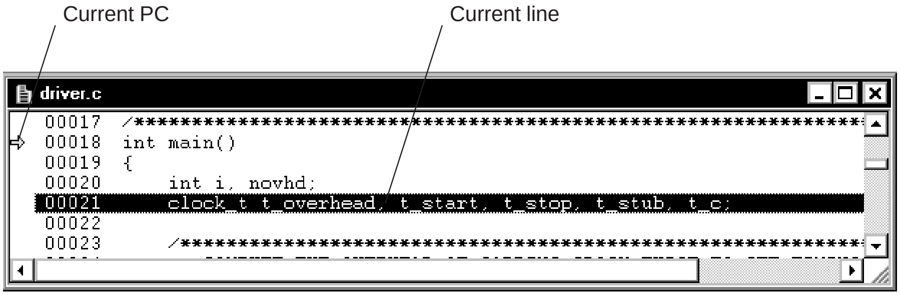
  dasm   PC
  or
  addr   PC

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

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

  Alternatively, you can select Restart from the Debug menu.

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



```
/*********************************************************************
int main()
{
    int i, novhd;
    clock_t t_overhead, t_start, t_stop, t_stub, t_c;

    /*****************************************************************
```

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

1) Open the context menu for the window. (For more information, see page 1-6.)

2) Select Set PC to Cursor from the context menu.

❑ Modify the PC's contents with one of these commands:

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

❑ Modify the value of the PC in the CPU window. (For more information about changing values the displayed in the CPU window, see section 8.3, *Basic Methods for Changing Data Values*, on page 8-5.)

7.2 Using the Basic Run Commands

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

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

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

Running an entire program

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

- Click the Run icon on the toolbar:
- From the Debug menu, select Run.
- Press F5.
- From the command line, enter the RUN command. The format for this command is:

  run [*expression*]

  If you supply a logical or relational *expression*, the RUN command becomes a conditional run (see section 7.4 on page 7-11).

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

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

- The debugger encounters a breakpoint. (For more information about how breakpoints affect a conditional run, see section 7.4 on page 7-11.)
- You click the Halt icon on the toolbar:
- You select Halt! from the Debug menu.
- You press ESC.

Running code up to a specific point in a program

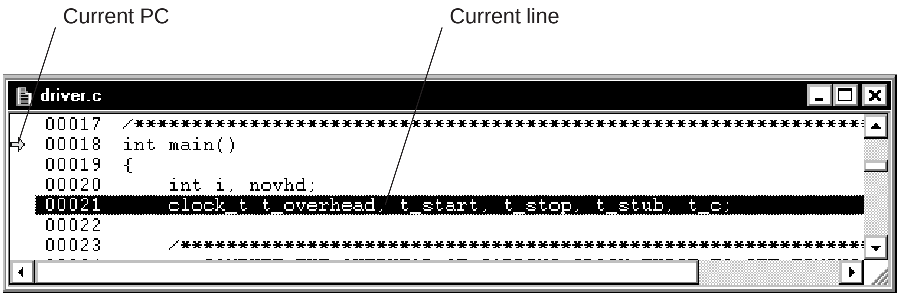
You can execute code up to a specific point in your program by using the GO command. The format for this command is:

go [*address*]

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

- ❑ The debugger encounters a breakpoint.
- ❑ You click the Halt icon on the toolbar:
- ❑ You select Halt! from the Debug menu.
- ❑ You press ESC .

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



```
/******************************************************************
int main()
{
    int i, novhd;
    clock_t t_overhead, t_start, t_stop, t_stub, t_c;

    /**************************************************************
```

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

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

Running the code in the current C function

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

- Click the Return icon on the toolbar:
- From the Debug menu, select Return.

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

- Click the Halt icon on the toolbar:
- From the Debug menu, select Halt!.
- Press ESC .

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

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

When you use the RUNF command, the debugger clears all breakpoints, disconnects the emulator from the target system, and causes the processor to begin execution at the current PC. You can quit the debugger, or you can continue to enter commands. However, any command that causes the debugger to access the target at this time produces and error. To begin entering commands that access the target without creating errors, type HALT in the command line.

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

Running code through breakpoints

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

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

To halt a continuous run, use one of the methods described in section 7.6 on page 7-13.

Resetting the simulator

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

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

If you are using the simulator and you execute a software reset, the simulator simulates the ’C54x processor and peripheral reset operation, putting the processor in a known state.

Resetting the emulator

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

Note:

The Reset Target option does not result in a physical reset of the target. It simply sets your system to the power-on reset value.

7.3 Single-Stepping Through Code

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

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

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

- If you do not supply an *expression*, the program executes a single statement, then halts.
- If you supply a logical or relational *expression*, the program treats the expression as a conditional single-step (see section 7.4 on page 7-11).
- If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger single-steps *count* assembly language statements unless you are currently in C code. If you are currently in C code, the debugger single-steps *count* C statements.

Single-stepping through assembly language or C code

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

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

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

- Click the Step icon on the toolbar:
- From the Debug menu, select Step.
- Press F8 .
- From the command line, enter the STEP command. The format for this command is:

  step [*expression*]

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

For more information about the compiler's –g option, see the *TMS320C54x Optimizing C Compiler User's Guide*.

Single-stepping through C code

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

- Click the Single Step C icon on the toolbar:
- From the Debug menu, select Step C.
- Press CONTROL F8 .
- From the command line, enter the CSTEP command. The format for this command is:

  cstep [*expression*]

Continuously stepping through code

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

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

If no breakpoints are set, you can halt a continuous step by using one of the methods described in section 7.6 on page 7-13.

Single-stepping through code and stepping over C functions

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

- ❑ To use the next single-step command and single-step in terms of assembly language or C statements (similar to the basic single-step command), choose one of these methods:
  - ■ Click the Next Statement icon on the toolbar:
  - ■ From the Debug menu, select Next.
  - ■ Press F10.
  - ■ From the command line, enter the NEXT command. The format for this command is:

    next [*expression*]

- ❑ To use the next single-step command and single-step in terms of C statements (similar to the C single-step command), choose one of these methods:
  - ■ Click the Next C Statement icon on the toolbar:
  - ■ From the Debug menu, select Next C.
  - ■ Press CONTROL F10.
  - ■ From the command line, enter the CNEXT command. The format for this command is:

    cnext [*expression*]

7.4 Running Code Conditionally

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

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

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

You must use software breakpoints with conditional runs; the expression is evaluated each time the debugger encounters a breakpoint. (Breakpoints are described in section 7.7 on page 7-14.) For single-step commands, the expression is evaluated at each statement. Each time the debugger evaluates the conditional expression, it updates the screen.

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

7.5 Benchmarking

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

Benchmarking code is a multiple-step process:

Step 1: Set a software breakpoint at the statement that marks the beginning of the section of code that you want to benchmark. (For more information about setting software breakpoints, see section 7.7 on page 7-14.)

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

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

Step 4: From the Debug menu, select Run Benchmark.

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

Notes:

1) Run Benchmark (or RUNB command) counts CPU clock cycles from the current PC to the breakpoint. This count is not cumulative. You cannot add the number of clock cycles between points A and B to the number of cycles between points B and C to learn the number of cycles between points A and C. This situation occurs because of pipeline filling and flushing.

2) The value in CLK is valid only after using a Run Benchmark command that is terminated by a software breakpoint.

3) When programming in C, avoid using a variable named CLK.

4) The RUNB command accesses the analysis module to count CPU clock cycles. If you have set up an instruction breakpoint, the debugger halts on that breakpoint in addition to your software breakpoints.

7.6 Halting Program Execution

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

- Click the Halt icon on the toolbar:
- From the Debug menu, select Halt!.
- Press ESC.

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

What happens when you halt the emulator

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

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

7.7 Using Software Breakpoints

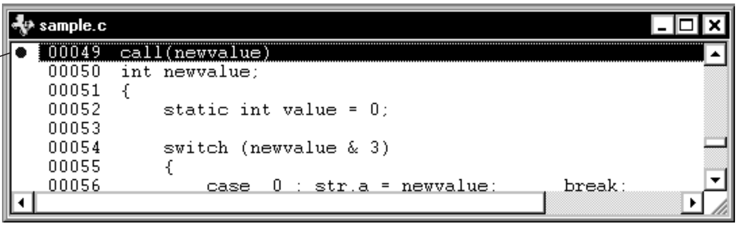
During the debugging process, you may want to halt execution temporarily so that you can examine the contents of selected variables, registers, and memory locations before continuing with program execution. You can do this by setting *software breakpoints* at critical points in your code. You can set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

Software breakpoints are especially useful in combination with conditional execution (described in section 7.4 on page 7-11).

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

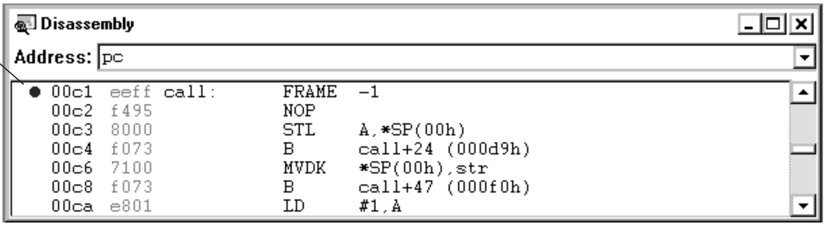
If you set a breakpoint in the disassembly, the debugger also highlights the associated C statement if the debugger has access to the C source. If you set a breakpoint in the C source, the debugger also highlights the associated statement in the disassembly. (If more than one assembly language statement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

A breakpoint is set at this C statement.



```
● 00049  call(newvalue)
  00050  int newvalue;
  00051  {
  00052      static int value = 0;
  00053
  00054      switch (newvalue & 3)
  00055      {
  00056          case  0 : str.a = newvalue;      break;
```

A breakpoint is also set at the associated assembly language statement.



Address: pc

```
● 00c1  eeff call:       FRAME   -1
  00c2  f495             NOP
  00c3  8000             STL     A,*SP(00h)
  00c4  f073             B       call+24 (000d9h)
  00c6  7100             MVDK    *SP(00h),str
  00c8  f073             B       call+47 (000f0h)
  00ca  e801             LD      #1,A
```

Notes:

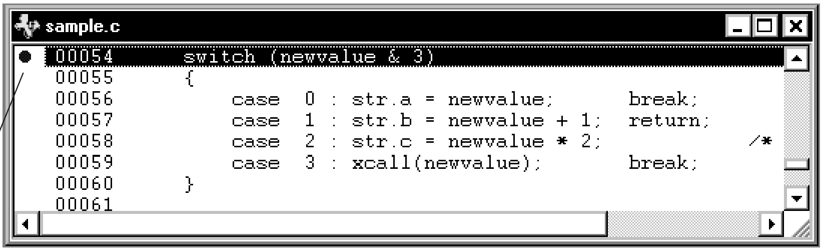
1) After execution is halted by a breakpoint, you can continue program execution by reissuing any of the run or single-step commands.

2) You can set up to 200 breakpoints.

3) You cannot set multiple breakpoints at the same statement.

Setting a software breakpoint

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



```
● 00054      switch (newvalue & 3)
  00055      {
  00056          case  0 : str.a = newvalue;      break;
  00057          case  1 : str.b = newvalue + 1;  return;
  00058          case  2 : str.c = newvalue * 2;          /*
  00059          case  3 : xcall(newvalue);       break;
  00060      }
  00061
```

A breakpoint is set on this statement.

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

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

1) Open the context menu for the window. (For more information, see page 1-6.)

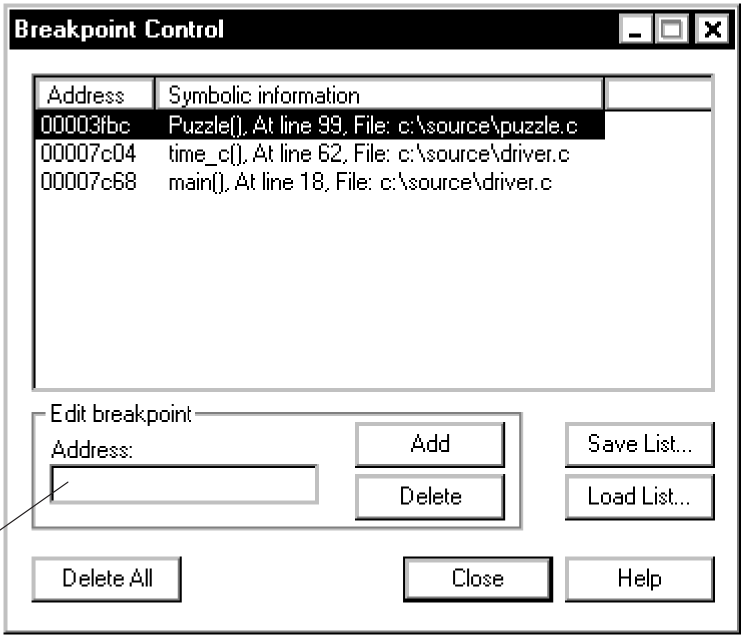
2) Select Toggle Breakpoint from the context menu.

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

❑ Click the Breakpoint Dialog icon on the toolbar:

❑ From the Configure menu, select Breakpoints.

This displays the Breakpoint Control dialog box:



Address	Symbolic information
00003fbc	Puzzle(), At line 99, File: c:\source\puzzle.c
00007c04	time_c(), At line 62, File: c:\source\driver.c
00007c68	main(), At line 18, File: c:\source\driver.c

List of set breakpoints

To set a breakpoint, enter an absolute address, any C expression, the name of a C function, or the name of an assembly language label and click Add.

To set a breakpoint, follow these steps:

3) In the Address field of the Breakpoint Control dialog box, enter an absolute address, any C expression, the name of a C function, or the name of an assembly language label. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.

4) Click Add. The new breakpoint appears in the breakpoint list.

5) Click Close to close the Breakpoint Control dialog box.

Clearing a software breakpoint

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

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

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

1) Select the line in the File or Disassembly window from which you want to remove the breakpoint.

2) From the context menu for the window, select Toggle Breakpoint.

You can also clear a breakpoint by using the Breakpoint Control dialog box (see the illustration on page 7-16):

1) Open the Breakpoint Control dialog box by using one of these methods:
   - Click the Breakpoint Dialog icon on the toolbar:
   - From the Configure menu, select Breakpoints.

2) Select the address of the breakpoint that you want to clear.

3) Click Delete. The breakpoint is removed from the breakpoint list.

4) Click Close to close the Breakpoint Control dialog box.

Clearing all software breakpoints

To clear all software breakpoints, follow these steps:

1) Open the Breakpoint Control dialog box by using one of these methods:
   - Click the Breakpoint Dialog icon on the toolbar:
   - From the Configure menu, select Breakpoints.

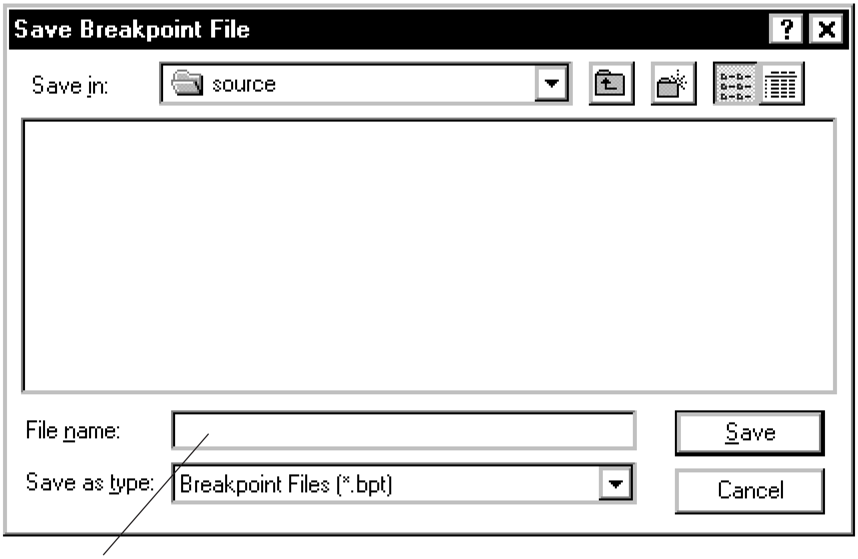
2) Click Delete All.

3) Click Close to close the Breakpoint Control dialog box.

Saving breakpoint settings

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

1) Open the Breakpoint Control dialog box by using one of these methods:
   - ❑ Click the Breakpoint Dialog icon on the toolbar:
   - ❑ From the Configure menu, select Breakpoints.
2) Click Save List. This displays the Save Breakpoint File dialog box:



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

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

Notes:

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

Loading saved breakpoint settings

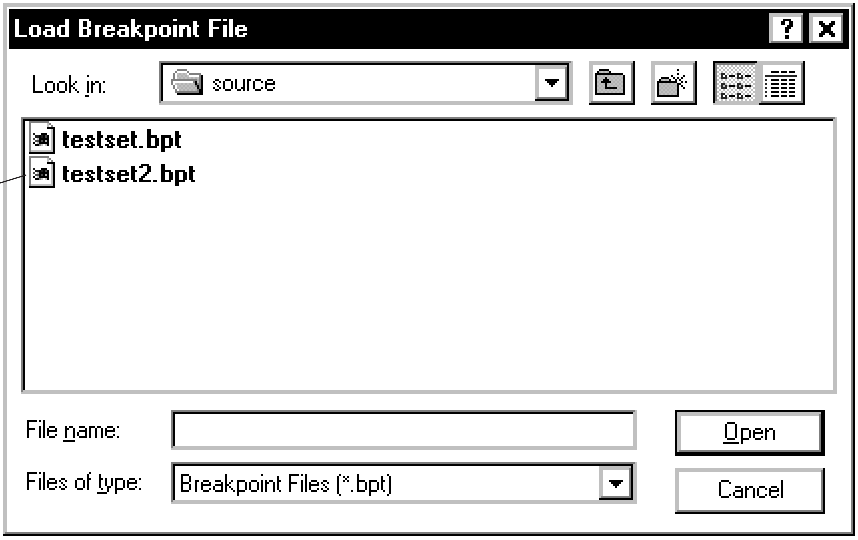
To load a list of saved breakpoints, follow these steps:

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

   - ❑ Click the Breakpoint Dialog icon on the toolbar:

   - ❑ From the Configure menu, select Breakpoints.

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



Look in: source

testset.bpt
testset2.bpt

File name:

Files of type: Breakpoint Files (*.bpt)

Select from a list of files.

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

4) Click Open.

5) In the Breakpoint Control dialog box, click Close.

Note:

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

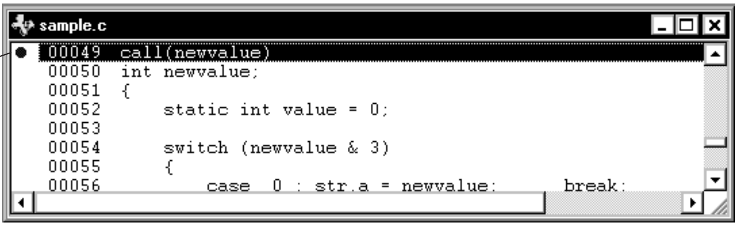
7.8 Using Hardware Breakpoints

You are able to set a hardware breakpoint once the development phase is complete and an application has been programmed into ROM or FLASH. A hardware breakpoint is used and set in a similar manner as a software breakpoint. The only difference is that a hardware breakpoint is limited to only two addresses in the Disassembly/C Source window. This limitation is imposed by the ICEBreaker, which provides the debugger with only two watchpoints when an application is programmed into ROM or FLASH.

Hardware breakpoints are also useful in combination with conditional execution (described in section 7.4 on page 7-11).

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

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

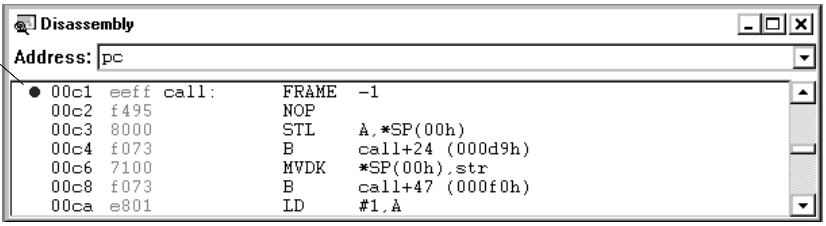


sample.c

```
● 00049  call(newvalue)
  00050  int newvalue;
  00051  {
  00052      static int value = 0;
  00053
  00054      switch (newvalue & 3)
  00055      {
  00056          case  0 : str.a = newvalue;      break;
```

A breakpoint is set at this C statement.

A breakpoint is also set at the associated assembly language statement.



Disassembly

Address: pc

```
● 00c1  eeff call:       FRAME  -1
  00c2  f495             NOP
  00c3  8000             STL    A,*SP(00h)
  00c4  f073             B      call+24 (000d9h)
  00c6  7100             MVDK   *SP(00h),str
  00c8  f073             B      call+47 (000f0h)
  00ca  e801             LD     #1,A
```

Notes:

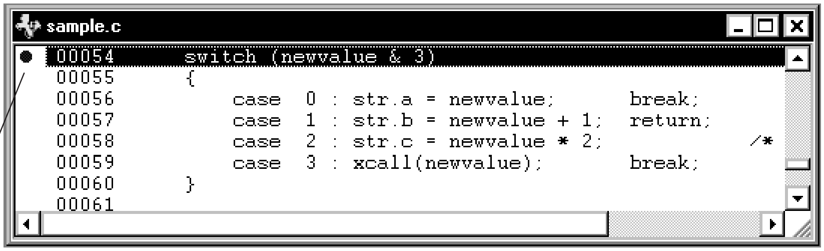
1) After execution is halted by a breakpoint, you can continue program execution by reissuing any of the run or single-step commands.

2) You can set up to 2 breakpoints (limitation imposed by the ICEBreaker).

3) You cannot set multiple breakpoints at the same statement.

Setting a hardware breakpoint

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



```
● 00054      switch (newvalue & 3)
  00055      {
  00056          case  0 : str.a = newvalue;      break;
  00057          case  1 : str.b = newvalue + 1;  return;
  00058          case  2 : str.c = newvalue * 2;           /*
  00059          case  3 : xcall(newvalue);       break;
  00060      }
  00061
```

A breakpoint is set on this statement.

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

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

1) Open the context menu for the window. (For more information, see page 1-6.)

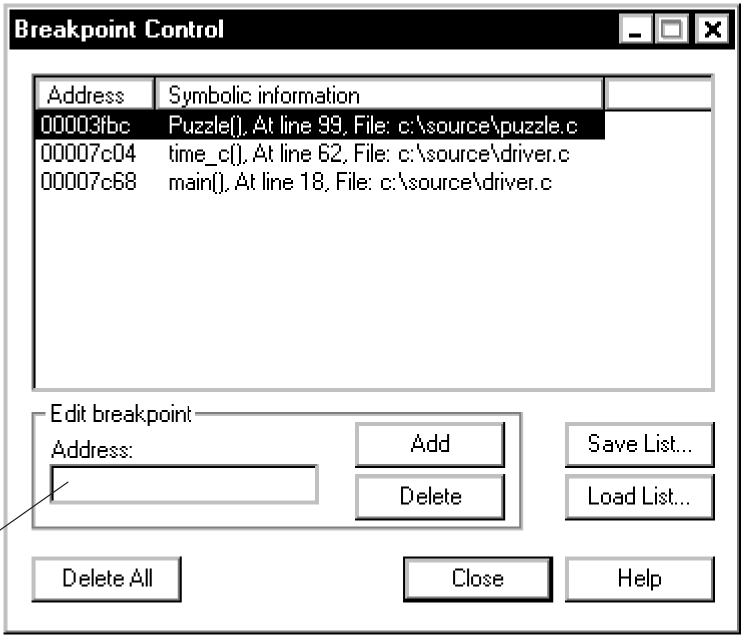
2) Select Toggle Breakpoint from the context menu.

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

❑ Click the Breakpoint Dialog icon on the toolbar:

❑ From the Configure menu, select Breakpoints.

This displays the Breakpoint Control dialog box:



Address	Symbolic information
00003fbc	Puzzle(), At line 99, File: c:\source\puzzle.c
00007c04	time_c(), At line 62, File: c:\source\driver.c
00007c68	main(), At line 18, File: c:\source\driver.c

List of set breakpoints

To set a breakpoint, enter an absolute address, any C expression, the name of a C function, or the name of an assembly language label and click Add.

To set a breakpoint, follow these steps:

3) In the Address field of the Breakpoint Control dialog box, enter an absolute address, any C expression, the name of a C function, or the name of an assembly language label. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.

4) Click Add. The new breakpoint appears in the breakpoint list.

5) Click Close to close the Breakpoint Control dialog box.

Clearing a hardware breakpoint

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

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

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

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

You can also clear a breakpoint by using the Breakpoint Control dialog box (see the illustration on page 7-16):

1) Open the Breakpoint Control dialog box by using one of these methods:
   - Click the Breakpoint Dialog icon on the toolbar:
   - From the Configure menu, select Breakpoints.
2) Select the address of the breakpoint that you want to clear.
3) Click Delete. The breakpoint is removed from the breakpoint list.
4) Click Close to close the Breakpoint Control dialog box.

Clearing all hardware breakpoints

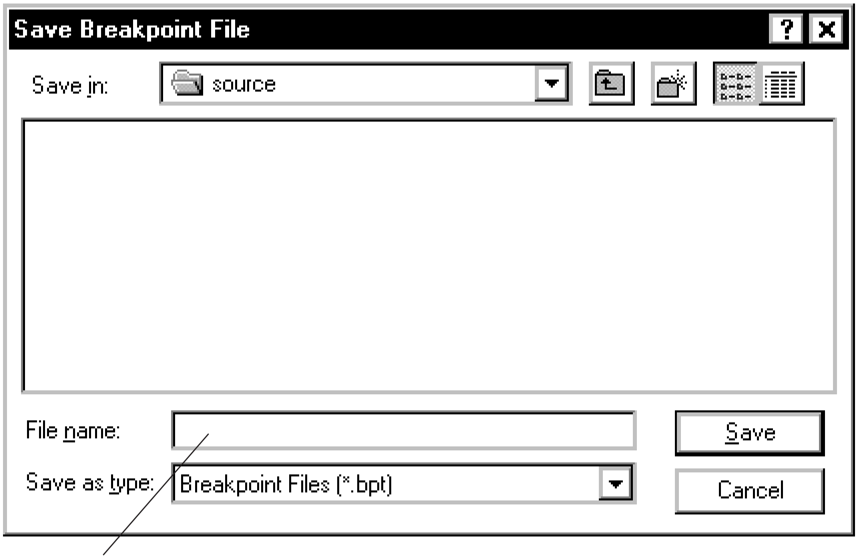
To clear all hardware breakpoints, follow these steps:

1) Open the Breakpoint Control dialog box by using one of these methods:
   - Click the Breakpoint Dialog icon on the toolbar:
   - From the Configure menu, select Breakpoints.
2) Click Delete All.
3) Click Close to close the Breakpoint Control dialog box.

Saving breakpoint settings

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

1) Open the Breakpoint Control dialog box by using one of these methods:
   - ❑ Click the Breakpoint Dialog icon on the toolbar:
   - ❑ From the Configure menu, select Breakpoints.
2) Click Save List. This displays the Save Breakpoint File dialog box:



Save in: source

File name:

Save as type: Breakpoint Files (*.bpt)

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

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

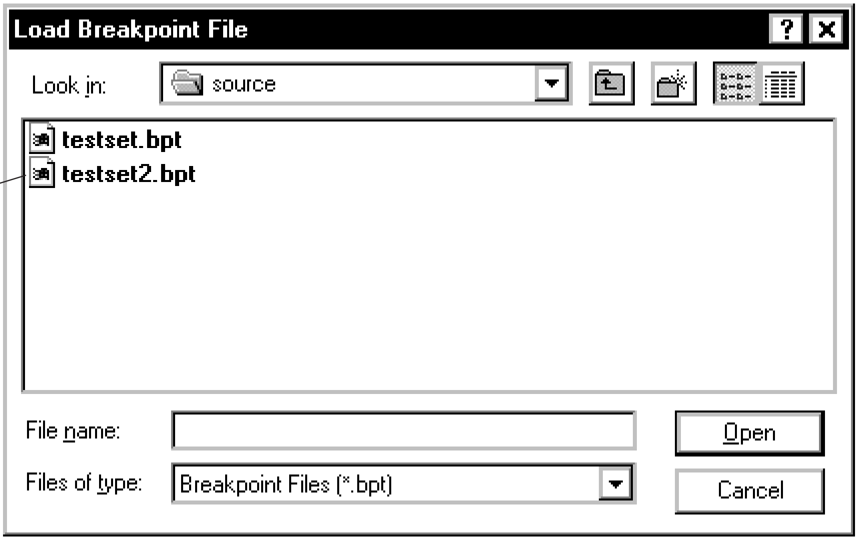
Notes:

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

Loading saved breakpoint settings

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

1) Open the Breakpoint Control dialog box by using one of these methods:
   - ❑ Click the Breakpoint Dialog icon on the toolbar:
   - ❑ From the Configure menu, select Breakpoints.
2) Click Load List. This displays the Load Breakpoint File dialog box:



Load Breakpoint File

Look in: source

testset.bpt
testset2.bpt

File name:

Files of type: Breakpoint Files (*.bpt)

Select from a list of files.

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

Note:

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

Error messages related to hardware breakpoints

The following error messages may appear in the Command window when using hardware breakpoints:

❑ **Hardware Resource Limit Exceeded** *address*. The debugger displays this message in the Command window when you set more than two hardware breakpoints in a memory region configured as ROM (that is, by clicking next to more than two statements in the Disassembly/C Source window). This message can also appear if you set a hardware breakpoint in the ROM memory region after you have configured both ICEBreaker watchpoint registers for hardware breakpoints (that is, through the analysis interface). If you need to set a hardware breakpoint at another address, you must clear one of the two existing hardware breakpoints.

❑ **Resource in use. Clear breakpoint to free**. The debugger displays this message in the Command window when you set a hardware breakpoint through the analysis interface and the corresponding watchpoint register is already being used for a hardware breakpoint. (A watchpoint register is used when you set a hardware breakpoint by clicking next to a statement in the Disassembly/C Source window). You must clear the breakpoint to free up the resource for use through the analysis interface.

❑ **Resource in use. Select Bypass to free**. This message is displayed in the Command window when you try to set a hardware breakpoint by clicking next to a statement at an address that is already configured as a hardware breakpoint through the Tools→Analysis→Break→Watchpoint X Setup dialog box. To free up the resource in use, select **Bypass** from the Action drop list in the Watchpoint X dialog box.

❑ **CANNOT STEP**. This message is displayed in the Command window when you, for instance, try to single-step over a switch-case statement. This message also appears when you initiate a command in C mode, such as STEP/NEXT/CSTEP/CNEXT. The message indicates that the debugger is not able to step over a C statement due to inadequate breakpoints.

Note:

The debugger inherently uses breakpoints to support stepping in C code. When an application is in ROM, there is a limitation to stepping in C because the debugger is limited to only two hardware breakpoints. Occasionally, an application may require many more breakpoints to support, for instance, *switch-case statements*. In this case, the total number of breakpoints required are equal to at least the number of *case statements*. In such a situation, the debugger simply displays the message **CANNOT STEP** in the Command window.

When an application is in RAM, the debugger has an adequate number of breakpoints to support stepping in C.

Chapter 8

Managing Data

The debugger allows you to examine and modify many types of data related to the ’C54x and to your program. You can display and modify these values:

❑ The contents of individual memory locations or a range of memory

❑ The contents of ’C54x registers

❑ Variables, including scalar types (ints, chars, etc.) and aggregate types (arrays, structures, etc.)

Topic **Page**

8.1 Where Data Is Displayed

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

Type of Data	Window Name	Purpose
Memory locations	Memory window	Displays the contents of a range of memory
Register values	CPU window	Displays the contents of ’C54x registers
Pointer data, variables, aggregate types, and specific memory locations or registers	Watch window	Displays selected data
Variable values	Variable window	Displays values for variables in the current or selected function

These dedicated 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.

Determining the type of a variable

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

whatis *symbol*

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

Command	Result Displayed in the Command Window
`whatis aai`	`int aai[10][5];`

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

Evaluating an expression

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

? *expression*

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

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

Here are some examples that use the ? command.

Command	Result Displayed in the Command Window

`? aai`	`aai[0][0] 1`
	`aai[0][1] 23`
	`aai[0][2] 45`
	.
	.
	.

`? j`	`4194425`
`? j=0x5a`	`90`

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

eval *expression*
or
e *expression*

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

For information about the PDM version of the EVAL command, see section 12.9, *Evaluating Expressions*, on page 12-21.

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 use this method to edit:

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

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

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

Editing data using expressions that have side effects

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

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

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

? IFR ⏎

```
? IFR++ ⏎          Side effect: increments the contents of IFR by 1
eval ––IFR ⏎       Side effect: decrements the contents of IFR by 1
? IFR = 8 ⏎                                    Side effect: sets IFR to 8
eval IFR/=2 ⏎             Side effect: divides contents of IFR by 2
```

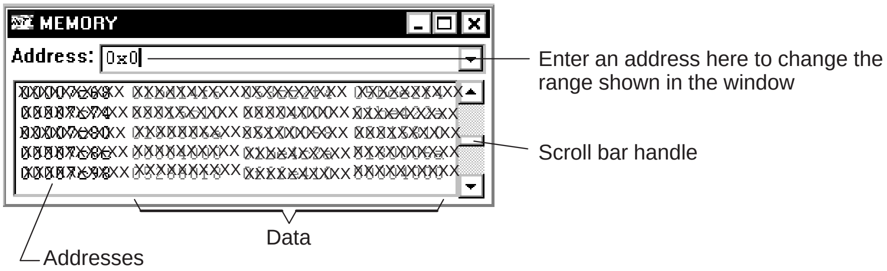
Not all expressions have side effects. For example, if you enter **? IFR+4**, the debugger displays the result of adding 4 to the contents of IFR but does not modify IFR’s contents. Expressions that have side effects must contain an assignment operator or an operator that implies an assignment. Operators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

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

8.4 Managing Data in Memory

The most common way to observe memory contents is to view the display in a Memory window. In mixed and assembly modes, the debugger displays the default Memory window automatically (labeled Memory). You can open any number of additional Memory windows to display different memory ranges. Figure 8–1 shows the default Memory window.

Figure 8–1. The Default Memory Window


MEMORY
Address:
0x0
Enter an address here to change the range shown in the window
Scroll bar handle
Data
Addresses

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

The debugger allows you to change the memory range displayed in the Memory window and to open additional Memory windows. The debugger also allows you to change the values at individual locations; for more information, see section 8.3, *Basic Methods for Changing Data Values*, on page 8-5.

Changing the memory range displayed in a Memory window

To change the memory range displayed in a Memory window, enter a new starting address in the Address field of the Memory window, as shown in Figure 8–1. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.

You can also change the display of any data-display window—including the Memory window—by scrolling through the window's contents. In the Memory window, the scroll bar handle is displayed in the middle of the scroll bar (see Figure 8–1). The middle of memory contents is defined as the most recent starting address that you entered in the Address field of the Memory window or with the MEM command (described on page 13-37). You can scroll up or down to see 1K bytes of memory on either side of the current starting address.

Opening an additional Memory window

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

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

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

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

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

    ```
    mem 0x0 ↵
    ```

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

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

    ```
    mem &SYM ↵
    ```

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

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

    ```
    mem SP – AR0 + label ↵
    ```

- The *display format* parameter is optional. When used, the data is displayed in the selected format, as shown in Table 8–2 on page 8-25.

- Use the *window name* parameter to name the additional Memory window. The debugger appends the *window name* to the Memory window label. If you do not supply a name, the debugger does not open a new window; it simply updates the default Memory window to reflect the changes.

Displaying program memory

By default, the Memory windows display data memory, but you can also display program memory or I/O space. To display program memory, follow any address parameter with **@prog**. For example, you can follow the MEM command's expression parameter with @prog. This suffix tells the debugger that the expression parameter identifies a program memory address instead of a data memory address.

If you display program memory in the Memory window, the debugger changes the window's label to Memory [Prog] so there is no confusion about what type of memory is displayed at any given time.

Any of the examples presented in this section could be modified to display program memory:

```
mem &SYM@prog
mem (SP – ARO + label)@prog
wa *0x26@prog
```

You can also use the suffix @data to display data memory; however, since data memory is the default, the @data suffix is unnecessary.

Displaying memory contents while you are debugging C

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

Note:

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

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

  ? *0x26 ⏎

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

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

  1) In the Expression field, enter **`*0x26.`**

  2) In the Format combo box, enter x - Hexadecimal.

  3) Click OK.

  The debugger displays the memory value in the Watch window.

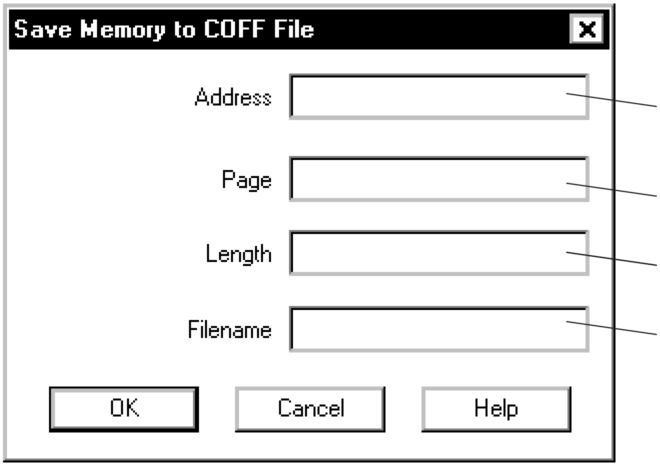
- You can also use the DISP command to display memory contents in a Watch window. The Watch window shows memory in an array format with the specified address as member [0]. In this situation, you can also use casting to display memory contents in a different numeric format:

  ```
  disp *(float *)0x26 ↵
  ```

Saving memory values to a file

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

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



Enter the first address of the block that you want to save.

Enter the memory page of the block.

Enter a length, in words, of the block.

Enter a file name for the saved block.

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

3) In the Page field, enter a 1-digit number that identifies the type of memory (program or data) that the address occupies:

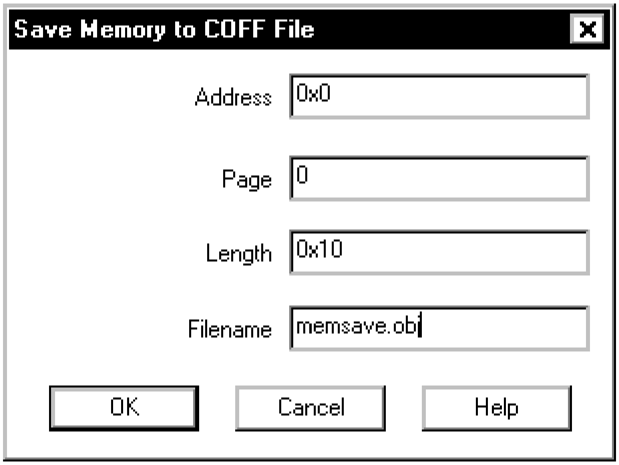
To identify this page . . .	Use this value as the *page* parameter . . .
Program memory	0
Data memory	1

4) In the Length field, enter a length, in words, of the block. This parameter can be any C expression.

5) In the Filename field, enter a name for the saved block of memory. If you do not supply an extension, the debugger adds a .obj extension.

6) Click OK.

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



Save Memory to COFF File

Address 0x0

Page 0

Length 0x10

Filename memsave.obj

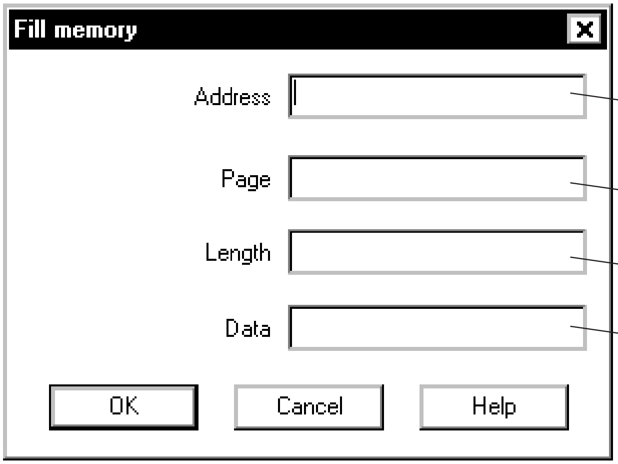
The value of the Length field is measured in words and data memory locations are measured in bytes. For this example, you must enter a length of words to equal 0x40 bytes (0x0000–0x0040).

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

Filling a block of memory

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

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



Enter the first address of the block that you want to fill.

Enter the memory page of the block.

Enter a length, in words, of the block.

Enter the data value that you want to use.

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

3) In the Page field, enter a 1-digit number that identifies the type of memory (program or data) that the address occupies:

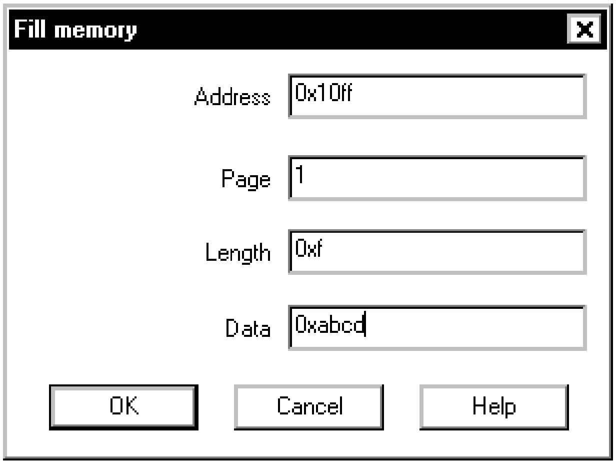
To identify this page . . .	**Use this value as the *page* parameter . . .**
Program memory	0
Data memory	1

4) In the Length field, enter a length, in words, of the block.

5) In the Data field, enter a value that you want placed in each word in the block.

6) Click OK.

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



Fill memory

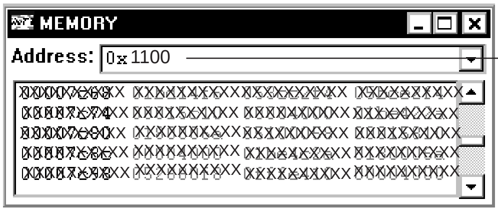
Address 0x10ff

Page 1

Length 0xf

Data 0xabcd

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



Address: 0x1100

Change the Memory display by entering a new address.

You can also use the debugger to fill a block of memory byte by byte with the Configure→Memory Fill→Fill Byte menu option.

1) From the Configure menu, select Memory Fill→Fill Byte. This displays the Fill Memory Byte dialog box.

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

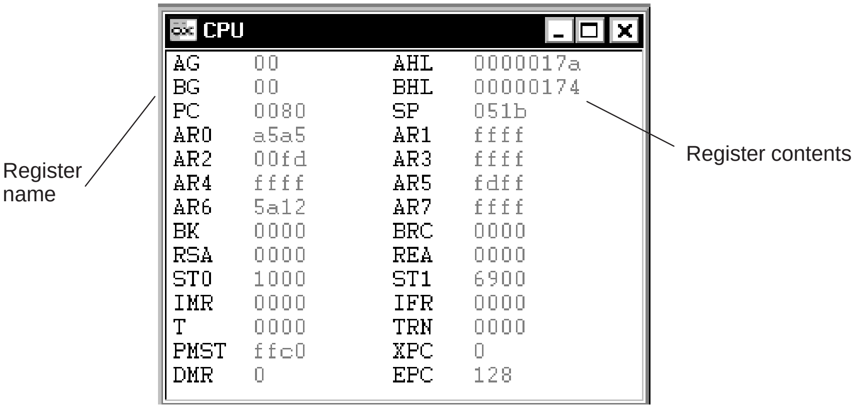
3) In the Length field, enter a length, in bytes, of the block.

4) In the Data field, enter a value that you want placed in each byte in the block.

5) Click OK.

8.5 Managing Register Data

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



```
AG     00          AHL    0000017a
BG     00          BHL    00000174
PC     0080        SP     051b
AR0    a5a5        AR1    ffff
AR2    00fd        AR3    ffff
AR4    ffff        AR5    fdff
AR6    5a12        AR7    ffff
BK     0000        BRC    0000
RSA    0000        REA    0000
ST0    1000        ST1    6900
IMR    0000        IFR    0000
T      0000        TRN    0000
PMST   ffc0        XPC    0
DMR    0           EPC    128
```

The debugger provides commands that allow you to display and modify the contents of specific registers. You can use the data-management commands or the debugger's overwrite editing capability to modify the contents of any register displayed in the CPU or Watch window. For more information, see section 8.3, *Basic Methods for Changing Data Values*, on page 8-5.

Displaying register contents

The main way to observe register contents is to view the display in the CPU window. You can rearrange the registers in the CPU window to display the ones that you are most interested in at the top of the CPU window. To do so, drag and drop the registers to the desired location in the CPU window.

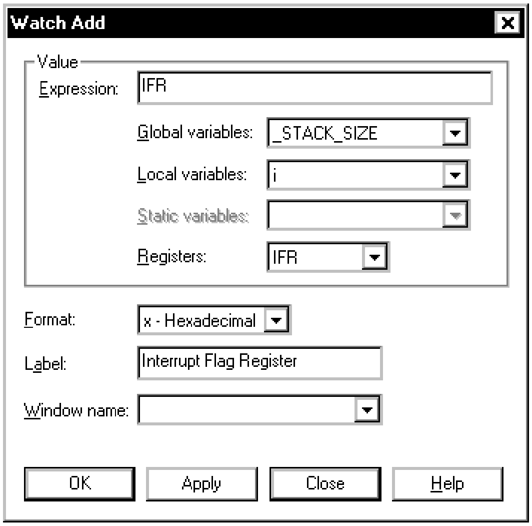
In addition to the CPU window, you can observe the contents of selected registers by using the ? (evaluate expression) command or the Configure→Watch Add menu option:

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

  ? IFR ⏎

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

- If you want to observe a register over a longer period of time, you can use the Configure→Watch Add menu option to display the register in a Watch window. For example, if you want to observe the interrupt flag register (IFR), you would fill in the Expression and Registers box fields as shown:



Watch Add

Value

Expression: IFR

Global variables: _STACK_SIZE

Local variables: i

Static variables:

Registers: IFR

Format: x - Hexadecimal

Label: Interrupt Flag Register

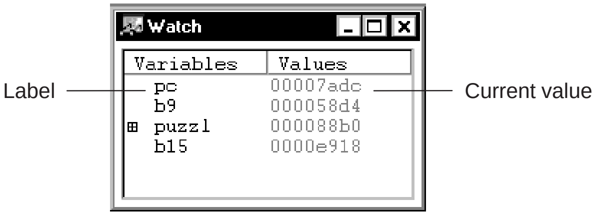
Window name:

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

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

8.6 Managing Data in a Watch Window

The debugger allows you to open a Watch window that shows you how program execution affects specific expressions, variables, registers, or memory locations. You can selectively watch a set of variables and/or registers, as well as expressions. You can also use the Watch window to display members of complex, aggregate data types, such as arrays and structures.



Variables	Values
pc	00007adc
b9	000058d4
⊞ puzzl	000088b0
b15	0000e918

Label

Current value

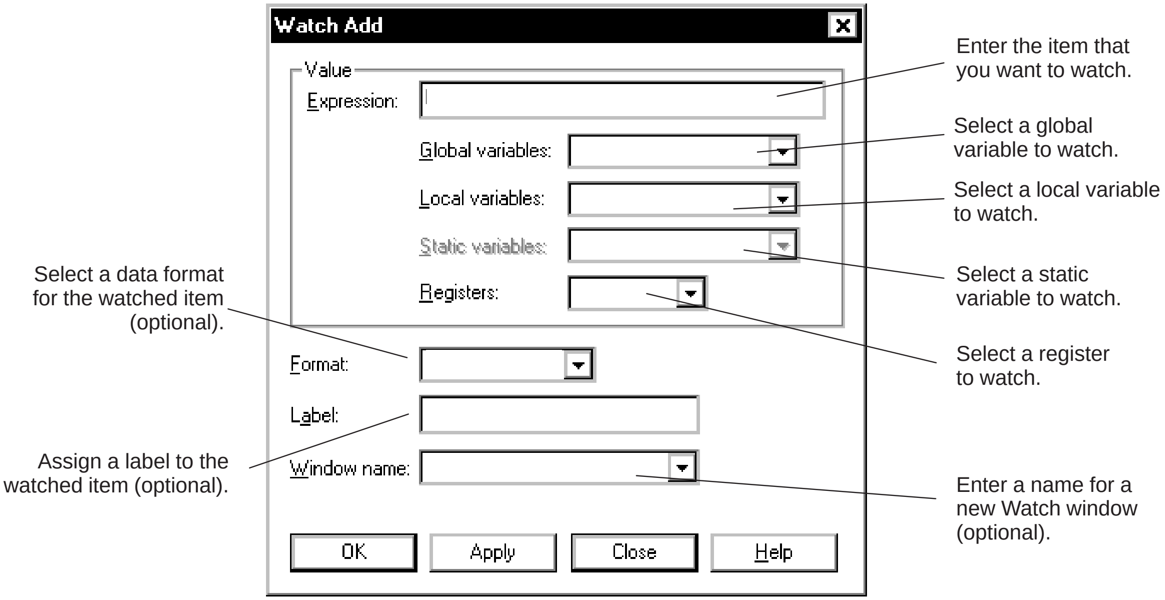
The debugger displays a Watch window *only when you specifically request a Watch window*.

Remember, you can use the data-management commands or the debugger’s overwrite editing capability to modify the contents of any value displayed in the Watch window. For more information, see section 8.3, *Basic Methods for Changing Data Values*, on page 8-5.

Displaying data in a Watch window

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

1) From the Configure menu, select Watch Add. This displays the Watch Add dialog box:



Enter the item that you want to watch.

Select a global variable to watch.

Select a local variable to watch.

Select a static variable to watch.

Select a register to watch.

Enter a name for a new Watch window (optional).

Select a data format for the watched item (optional).

Assign a label to the watched item (optional).

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

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

   ```
   *0x26
   ```

3) If you want to change the data format for the watched item, select the format you want to use from the Format drop list. The format field is optional.

4) If you want to assign a label for the watched item, use the Label field. If you leave the Label field blank, the debugger displays the expression, variable, or register as the label.

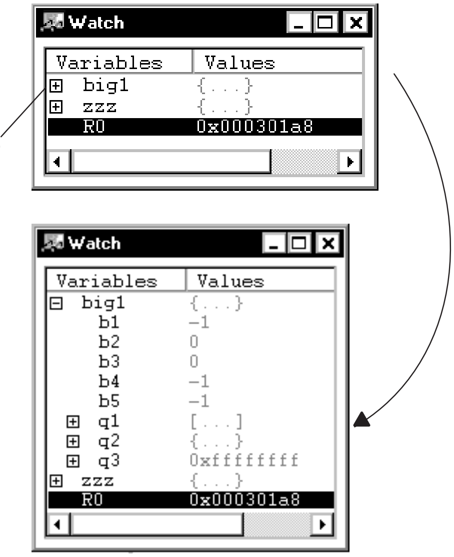
5) If you want to open a new Watch window, enter a name for the new Watch window in the Window name field. This field is optional. When you enter a window name, the debugger appends the window name to the Watch window label. If you do not supply a name, the debugger adds the item to the default Watch window.

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

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

Displaying additional data

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



Click this icon to display the contents of the array.

Variables	Values
⊞ big1	{...}
⊞ zzz	{...}
R0	0x000301a8

Variables	Values
⊟ big1	{...}
b1	−1
b2	0
b3	0
b4	−1
b5	−1
⊞ q1	[...]
⊞ q2	{...}
⊞ q3	0xffffffff
⊞ zzz	{...}
R0	0x000301a8

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

Deleting watched values

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

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

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

wr [{***** | *window name*}]

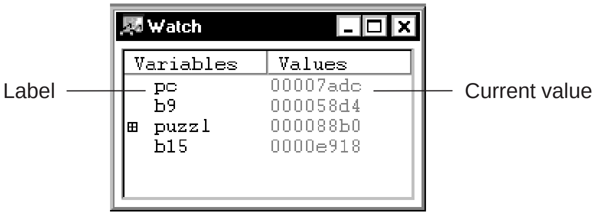
The optional *window name* parameter deletes a particular Watch window; ***** deletes all Watch windows.

Note:

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

8.7 Managing Data in a Variable Window

The Variable window provides direct access to all the variables currently found in your program. By using the Variable window, you can selectively change the values of existing variables without having to alter your C code file. Changing variable values allows you to see how a specific variable is incremented, as well as help determine how your code affects different variables without changing values in the actual file.



Variables	Values
pc	00007adc
b9	000058d4
⊞ puzzl	000088b0
b15	0000e918

Accessing a Variable window

You can access a Variable window using one of the following ways:

- Automatically when you are displaying C code
- With the View→Variable Window menu option
- With the CALLS command if you have previously closed the Calls window

Displaying data in a Variable window

You can select the types of variables to be displayed in a Variable window by clicking on one of the tabs at the bottom of the window.

- The Local tab displays variables and their associated values for the current function.
- The Auto tab displays variables and their associated values on the currently executing line of C code, as well as variables in the previous C statement.

The Variable window consists of a Context drop-down menu, which contains the current call stack functions. You can select a function for which you want the variables displayed or modified.

Modifying data in a Variable window

You can use the debugger's overwrite editing capability to modify the contents of any value displayed in the Variable window. The overwrite editing capability allows you to change a value simply by typing over its displayed value.

To modify data in a Variable window, follow these steps:

1) Select the data item that you want to modify. Choose one of these methods:

   - Double-click the data item that you want to modify.
   - Select the data item that you want to modify and press F9 .

2) Type the new information. If you make a mistake or change your mind, press ESC ; this resets the field to its original value.

3) When you finish typing the new information, press ↵ or click on another data value. This replaces the original value with the new value.

How a Variable window differs from a Watch window

The Variable window is different from the Watch window in that you are not able to add values. You can only modify values that already exist in the Variable window. As with the Watch window, you can use the debugger's overwrite editing capability (discussed in the preceding section) to modify the contents of any value displayed in the Variable window.

8.8 Managing Pipeline Information (Simulator Only)

The simulator supports additional features that allow you to monitor the pipeline. The simulator supports pseudoregisters that you can query with ? or DISP or add to the Watch window. The simulator also supports pipeline conflict detection, through the use of debugger options –l and –w.

The instruction pipeline consists of six phases: instruction prefetch, instruction fetch, decode, operand access1, operand access2, 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.

Monitoring the pipeline

The simulator provides twelve 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 d_ins and d_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.

Detecting pipeline conflicts

The ’C54x simulator supports pipeline conflict detection. When the debugger option –l is used, a warning message is written to the command window and code execution is halted each time a pipeline conflict occurs. For more information on the –l option, see page 2-12.

There are times, however, when you will not want to halt code execution each time a pipeline conflict is detected. For those instances, you can use the –w debugger option in conjunction with the –l option. The –w option writes the pipeline conflict messages to a file located on your current directory. This allows code execution to be completed, without warnings in the command window and without halts. Once code execution is complete, you can review all the warnings that took place by looking at that file. For more information on the –w option, see page 2-15.

The pipeline conflicts that can be detected by the simulator are:

- DAGEN register latency
  - ARx latency
  - BK latency
- Stack pointer latency
  - As an offset in direct addressing (CPL = 1)
  - Stack operation (CPL = 0)
- TREG latency
- PMST latency
  - OVLY latency
  - MPMC latency
  - DROM latency
  - IPTR latency
- Status registers (ST0/ST1) latency
  - ARP latency
  - CMPT latency
  - CPL latency
  - DP latency
  - SXM latency
  - ASM latency
  - BRAF latency
- BRC register latency
- MMR access of accumulators latency

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 an 0x prefix).
- ❑ Enumerated types are displayed symbolically.

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

Changing the default format for specific data types

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

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

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

Table 8–2. Display Formats for Debugger Data

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

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

Table 8–3. Data Types for Displaying Debugger Data

	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			√	√			√	√		Hexadecimal (x)

Here are some examples:

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

```
setf short, u ↵
```

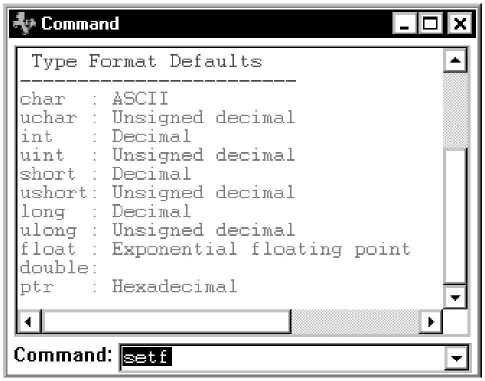
❑ To return all data of type short to its default display format, enter:

```
setf short, * ↵
```

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

```
setf ↵
```

The display should look something like this:



```
 Type Format Defaults
-------------------------
char   : ASCII
uchar  : Unsigned decimal
int    : Decimal
uint   : Unsigned decimal
short  : Decimal
ushort : Unsigned decimal
long   : Decimal
ulong  : Unsigned decimal
float  : Exponential floating point
double :
ptr    : Hexadecimal
```

Command: setf

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

setf * ↵

Changing the default format with data-management commands

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

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

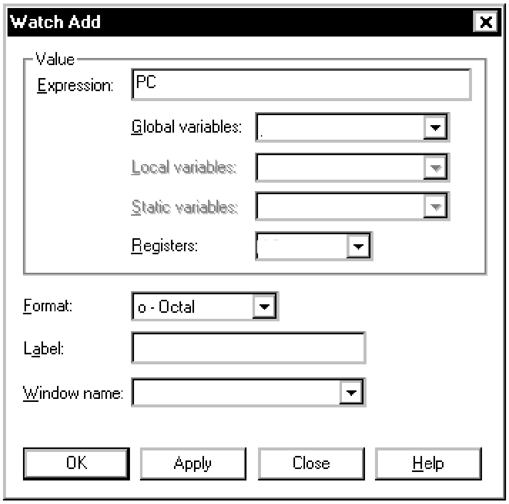
When you do not use a display format parameter, data is shown in its natural format (unless you have changed the format for the data type with SETF).

Here are some examples:

- To display memory contents in octal, enter:

  `mem 0x0,o` ⏎

- To watch the PC in octal, from the Format drop list, select o - Octal:



Watch Add

Expression: PC

Format: o - Octal

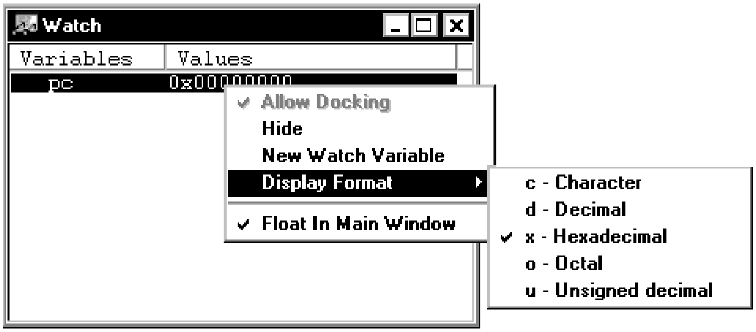
- To change the format of the PC in the Watch window, follow these steps:

  1) In the Watch window, select PC.

  2) Right click the mouse to bring up the Watch window context menu.

  3) From the context menu, select Display Format. A submenu of data formats appears.

  4) From the submenu, select the format in which you want the PC to display.



Watch

Variables	Values
pc	0x00000000

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

Chapter 9

Profiling Code Execution

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

Topic Page

9.1 Overview of the Profiling Environment

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

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

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

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

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

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

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

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

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

9.2 Overview of the Profiling Process

Profiling consists of five simple steps:

Step 1

Enter the profiling environment. See *Entering the Profiling Environment*, page 9-4.

Step 2

Identify the areas of code where you want to collect statistics. See *Defining Areas for Profiling*, page 9-5.

Step 3

Identify the profiling session stopping points. See *Defining a Stopping Point,* page 9-15.

Step 4

Begin profiling. See *Running a Profiling Session,* page 9-17.

Step 5

View the profile data. See *Viewing Profile Data,* page 9-20.

Note:

When you compile a program that will be profiled, you must use the –g and the –as compiler shell options. The –g option includes symbolic debugging information; the –as option ensures that you will be able to include ranges as profile areas. For more information on these options, see the TMS320C54x C Compiler User's Guide.

A profiling strategy

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

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

9.3 Entering the Profiling Environment

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

Some restrictions apply to the profiling environment:

- ❑ The debugger is always in mixed mode.
- ❑ Command, Disassembly, File, and Profile are the only windows available; additional windows, such as a Watch window, cannot be opened.
- ❑ The profiling environment supports only a subset of the debugger commands. Table 9–1 lists the debugger commands that can and cannot be used in the profiling environment.

Table 9–1. Debugger Commands That Can/Cannot Be Used in the Profiling Environment

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

Chapter 13, *Summary of Commands*, summarizes all of the debugger commands and tells you whether a command is valid in the profiling environment.

9.4 Defining Areas for Profiling

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

❑ *Individual lines* in C or disassembly
❑ *Ranges* in C or disassembly
❑ *Functions* in C only

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

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

The following subsections explain how to mark, disable, reenable, and unmark profile areas by using the mouse or the dialog box. For restrictions on profiling areas, see page 9-12.

Marking an area with a mouse

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

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

Notes:

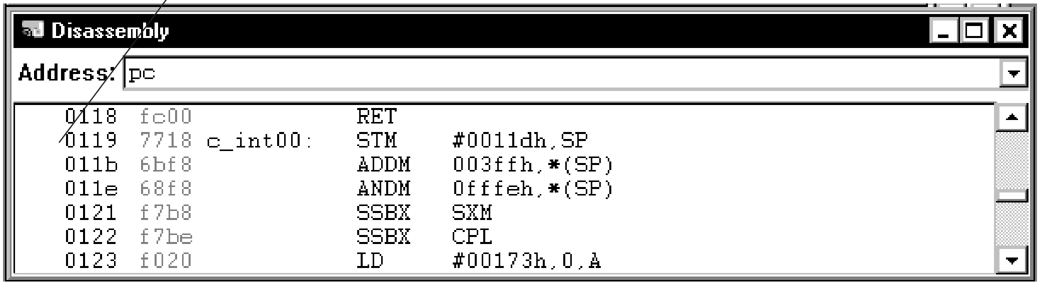
1) Marking an area in C *does not* mark the associated code in disassembly.

2) Areas can be nested; for example, you can mark a line within a marked range. The debugger reports statistics for both the line and the function.

3) Ranges cannot overlap, and they cannot span function boundaries.

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

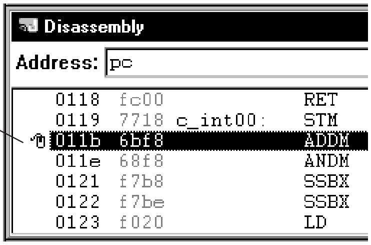
1) In the File or Disassembly window, click once to the left of the line that you want to mark or to the left of the first line of the range that you want to mark:



```
0118 fc00            RET
0119 7718 c_int00:   STM    #0011dh,SP
011b 6bf8            ADDM   003ffh,*(SP)
011e 68f8            ANDM   0ffeh,*(SP)
0121 f7b8            SSBX   SXM
0122 f7be            SSBX   CPL
0123 f020            LD     #00173h,0,A
```

Click next to the line that you want to mark.

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

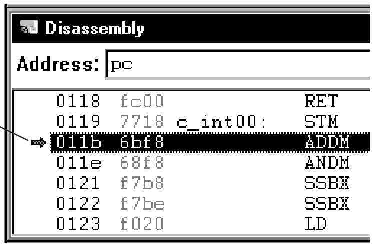


```
0118 fc00            RET
0119 7718 c_int00:   STM
011b 6bf8            ADDM
011e 68f8            ANDM
0121 f7b8            SSBX
0122 f7be            SSBX
0123 f020            LD
```

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

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

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

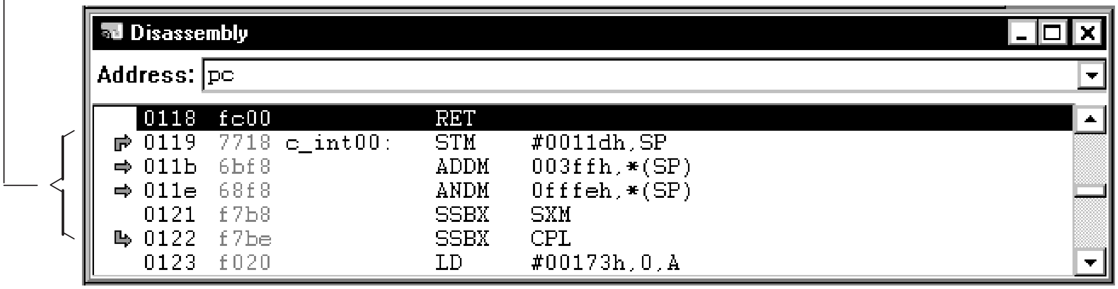


```
0118 fc00            RET
0119 7718 c_int00:   STM
011b 6bf8            ADDM
011e 68f8            ANDM
0121 f7b8            SSBX
0122 f7be            SSBX
0123 f020            LD
```

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

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

This range is marked and enabled.



```
pc

   0118  fc00                RET
↱  0119  7718  c_int00:      STM    #0011dh,SP
⇨  011b  6bf8                ADDM   003ffh,*(SP)
⇨  011e  68f8                ANDM   0ffffeh,*(SP)
   0121  f7b8                SSBX   SXM
↳  0122  f7be                SSBX   CPL
   0123  f020                LD     #00173h,0,A
```

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

1) In the File window, click next to the statement that declares the function that you want to mark.

2) When you see the mouse icon, click again to mark and enable the C function. A green arrow appears, indicating that the function is marked.

Note:

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

If you are not in the profiling environment, single-clicking next to a line or function sets a software breakpoint.

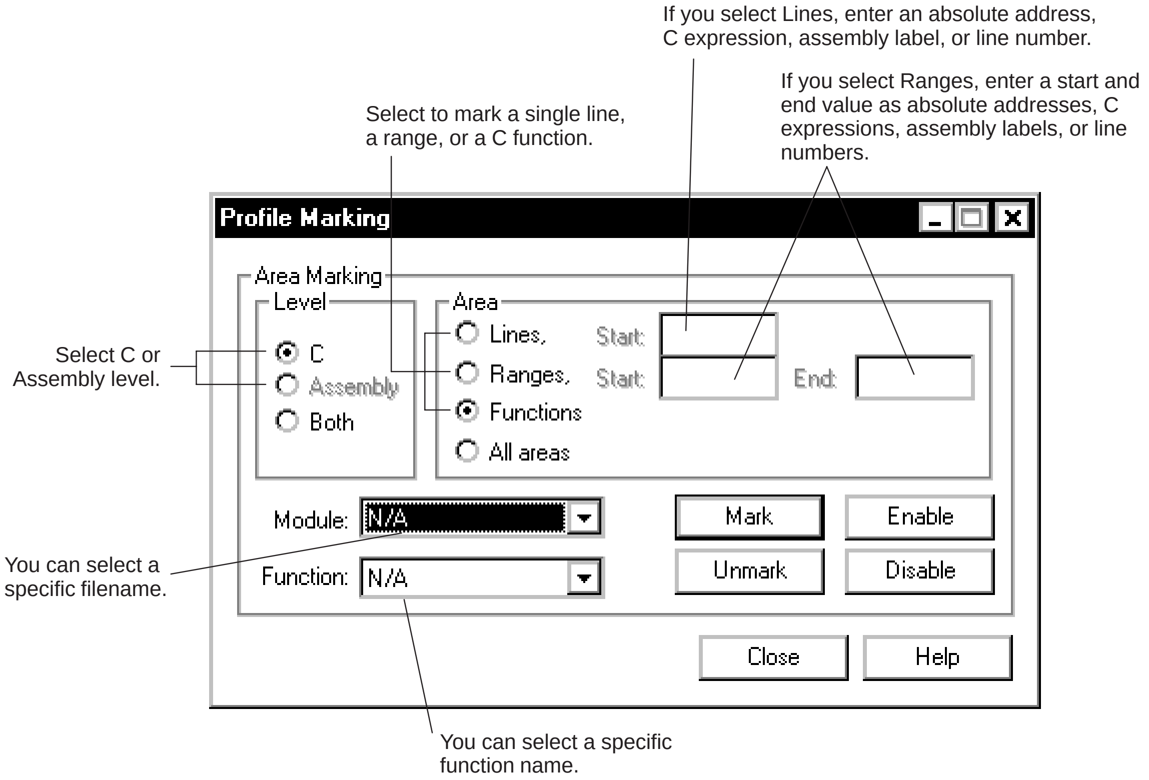
Marking an area with a dialog box

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

1) Open the Profile Marking dialog box by using one of these methods:

   ❑ From the Tools→Profile menu, select Select Areas.
   ❑ From the context menu for the Profile window, select Select Areas.

   This displays the Profile Marking dialog box:



If you select Lines, enter an absolute address, C expression, assembly label, or line number.

If you select Ranges, enter a start and end value as absolute addresses, C expressions, assembly labels, or line numbers.

Select to mark a single line, a range, or a C function.

Select C or Assembly level.

You can select a specific filename.

You can select a specific function name.

2) In the Level box, select C or Assembly.

3) In the Area box, select Lines, Ranges, or Functions. See Table 9–2 for a list of valid combinations.

4) Depending on what you select in step 3, do one or more of the following:

   ❑ Next to Lines, enter an absolute address, C expression, assembly label, or line number. If you are entering an absolute address, be sure to prefix it with 0x.

   ❑ Next to Ranges, enter a Start and an End value as absolute addresses, C expressions, assembly labels, or line numbers.

   ❑ From the Module combo box, select a specific filename.

   ❑ From the Function combo box, select a specific function name.

   See Table 9–2 for a list of valid combinations.

5) Click Mark.

6) Click Close to close the dialog box.

Table 9–2. Using the Profile Marking Dialog Box to Mark Areas

(a) Marking lines

To mark this area...	**If C level is selected...**	**If Assembly level is selected...**
By line number, address	❑ Select a module name. ❑ In the Area box, select Lines. ❑ Next to Lines, specify a line number.	❑ In the Area box, select Lines. ❑ Next to Lines, specify an absolute address, a C expression, or an assembly label
All lines in a function	❑ Select a function name. ❑ In the Area box, select Lines.	❑ Select a function name. ❑ In the Area box, select Lines.

(b) Marking ranges

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

(c) Marking functions

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

Disabling an area

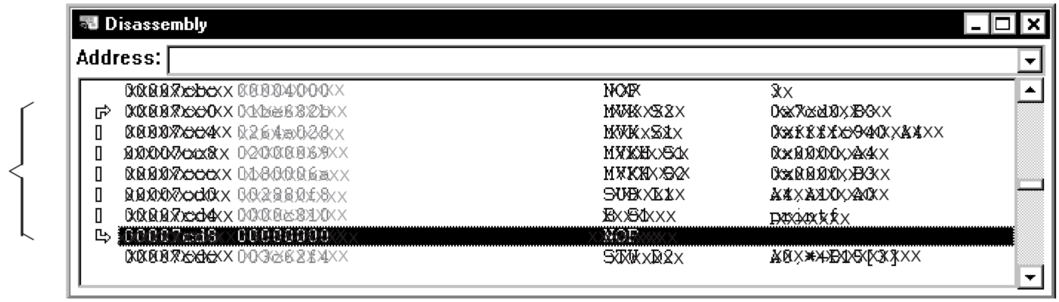
At times, it is useful to identify areas that you do not want to affect profile statistics. To do this, *disable* the appropriate area. Disabling effectively subtracts the timing information of the disabled area from all profile areas that include or call the disabled area. Areas must be marked before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

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

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

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

This range is still marked, but it is disabled.



```
   00007cbc 00004000    NOP          3
⮣  00007cc0 01be682b    MVK .S2      0x7cd0,B3
▯  00007cc4 0264a028    MVK .S1      0xffffc940,A4
▯  00007cc8 02000069    MVKH .S1     0x0000,A4
▯  00007ccc 0180006a    MVKH .S2     0x0000,B3
▯  00007cd0 002880f8    SUB .L1      A4,A10,A0
▯  00007cd4 0000c810    B .S1        printf
⮡  00007cd8 00000000    NOP
   00007cdc 003c62f4    STW .D2      A0,*+B15[3]
```

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

1) Open the Profile Marking dialog box by using one of these methods:

   - ❑ From the Tools→Profile menu, select Select Areas.
   - ❑ From the context menu for the Profile window, select Select Areas.

   This displays the Profile Marking dialog box.

2) In the Level box, select C, Assembly, or Both.

3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 9–3 on page 9-13 for a list of valid combinations.

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

   See Table 9–3 for a list of valid combinations.
5) Click Disable.
6) Click Close to close the dialog box.

Reenabling a disabled area

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

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

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

   This displays the Profile Marking dialog box.
2) In the Level box, select C, Assembly, or Both.
3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 9–3 for a list of valid combinations.
4) Depending on what you select in step 3, do one or more of the following:
   - ❑ Next to Lines, enter an absolute address, C expression, assembly label, or line number.
   - ❑ Next to Ranges, enter a Start and an End value as an absolute address, C expression, assembly label, or line number.
   - ❑ From the Module combo box, select a specific filename.
   - ❑ From the Function combo box, select a specific function name.

   See Table 9–3 for a list of valid combinations.
5) Click Enable.
6) Click Close to close the dialog box.

Unmarking an area

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

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

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

1) Open the Profile Marking dialog box by using one of these methods:
   - ❑ From the Tools→Profile menu, select Select Areas.
   - ❑ From the context menu for the Profile window, select Select Areas.
2) In the Level box, select C, Assembly, or Both.
3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 9–3 for a list of valid combinations.
4) Depending on what you select in step 3, do one or more of the following:
   - ❑ Next to Lines, enter an absolute address, C expression, assembly label, or line number.
   - ❑ Next to Ranges, enter a Start and an End value as absolute addresses, C expressions, assembly labels, or line numbers.
   - ❑ From the Module combo box, select a specific filename.
   - ❑ From the Function combo box, select a specific function name.

   See Table 9–3 for a list of valid combinations.
5) Click Unmark.
6) Click Close to close the dialog box.

Restrictions on profiling areas

The following restrictions apply to profiling areas:

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

Table 9–3. Disabling, Enabling, Unmarking, or Viewing Areas

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

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

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

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

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

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

Table 9–3. Disabling, Enabling, Unmarking, or Viewing Areas (Continued)

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

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

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

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

9.5 Defining a Stopping Point

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

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

Note:

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

Setting and clearing a software breakpoint in the profiling environment is similar to setting and clearing a software breakpoint in the basic debugging environment. The only difference between the two is you must double-click next to a statement to set or clear a software breakpoint in the profiling environment, and you single-click in the basic debugging environment. For more information about setting and clearing software breakpoints in the basic debugging environment, see section 7.7 on page 7-14.

Setting a software breakpoint

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

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

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

   - Click the Breakpoint Dialog icon on the toolbar:

   - From the Configure menu, select Breakpoints.

2) In the Address field of the Breakpoint Control dialog box, enter an absolute address, any C expression, the name of a C function, or an assembly language label.

3) Click Add. The new breakpoint appears in the breakpoint list.

4) Click Close to close the Breakpoint Control dialog box.

Clearing a software breakpoint

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

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

1) Open the Breakpoint Control dialog box by using one of these methods:
   - Click the Breakpoint Dialog icon on the toolbar:
   - From the Configure menu, select Breakpoints.
2) Select the address of the breakpoint that you want to clear.
3) Click Delete. The breakpoint is removed from the breakpoint list.
4) Click Close to close the Breakpoint Control dialog box.

9.6 Running a Profiling Session

Once you have defined profile areas and a stopping point, you can run a profiling session. You can run two types of profiling sessions:

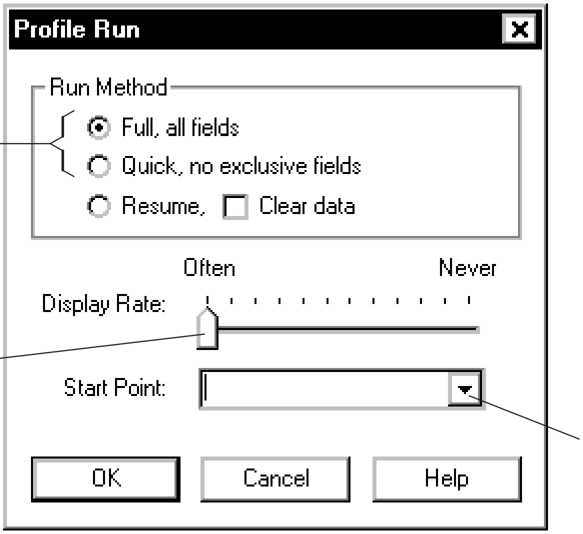
- A *full profile* collects a full set of statistics for the defined profile areas.
- A *quick profile* collects a subset of the available statistics (it does not collect exclusive or exclusive max data, which are described in section 9.7 on page 9-20). This reduces overhead because the debugger does not have to track entering/exiting subroutines within an area.

Running a full or a quick profiling session

To run a profiling session, follow these steps:

1) Open the Profile Run dialog box by using one of these methods:
   - Click the Run icon on the toolbar:
   - From the Debug menu, select Run.
   - Press F5.

   This displays the Profile Run dialog box:



Run Method:
● Full, all fields
○ Quick, no exclusive fields
○ Resume, □ Clear data

Display Rate: Often

Start Point:

Select the type of profiling session that you want to perform.

Slide the frequency bar to specify how often the display is updated.

Click this arrow to choose from a list of starting points.

2) In the Run Method box, select the type of profiling session that you want to perform: Full or Quick.

3) Slide the Display Rate frequency bar to specify how often the display is updated.

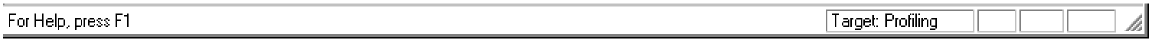
   You can choose a Display Rate from Often to Never. A Display Rate of Never causes the profiler to display profiling information only when the profiling session is complete.

4) In the Start Point field, enter the starting point for the profiling session. The starting point can be a label, a function name, or a memory address. If you specify a memory address, be sure to prefix the address with **0x**.

   You can choose from a list of starting points by clicking on the arrow at the end of the Start Point field.

5) Click OK.

After you click OK, your program **restarts** and **runs to the defined starting point**. You can tell that the debugger is profiling because the status bar changes to *Target: Profiling*, as shown here.



For Help, press F1

Target: Profiling

Profiling begins when the starting point is reached and continues until a stopping point is reached or until you halt the profiling session by doing one of the following:

- Click the Halt icon on the toolbar:

- From the Debug menu, select Halt!.

- Press ESC .

Resuming a profiling session that has halted

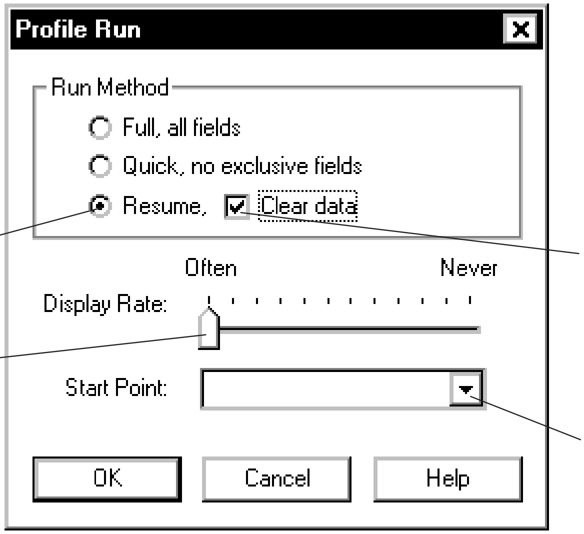
To resume a profiling session that has halted, follow these steps:

1) Open the Profile Run dialog box by using one of these methods:

   - Click the Run icon on the toolbar:

   - From the Debug menu, select Run.
   - Press F5 .

   This displays the Profile Run dialog box:



Profile Run

Run Method

○ Full, all fields

○ Quick, no exclusive fields

● Resume, ☑ Clear data

Display Rate: Often — Never

Start Point:

To resume a profiling session that has halted, select Resume.

Slide the scale bar to specify how often the display is updated.

To clear out previously collected profile data, select Clear data.

Click this arrow to choose from a list of starting points.

2) In the Run Method box, select Resume.

3) If you want to clear out the previously collected data, select Clear data in the Run Method box.

4) Slide the Display Rate scale to specify how often the display is updated.

   You can choose a Display Rate from Often to Never. A Display Rate of Never causes the profiler to display profiling information only when the profiling session is complete.

5) In the Start Point field, enter the starting point for the profiling session. The starting point can be a label, a function name, or a memory address. If you specify a memory address, be sure to prefix the address with **0x**.

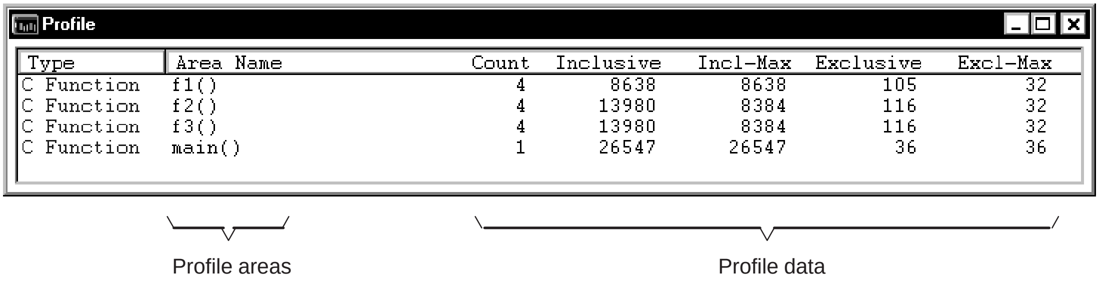
   You can choose from a list of starting points by clicking on the arrow at the end of the Start Point field.

6) Click OK.

9.7 Viewing Profile Data

The statistics collected during a profiling session are displayed in the Profile window. Figure 9–1 shows an example of this window.

Figure 9–1. An Example of the Profile Window



Type	Area Name	Count	Inclusive	Incl-Max	Exclusive	Excl-Max
C Function	f1()	4	8638	8638	105	32
C Function	f2()	4	13980	8384	116	32
C Function	f3()	4	13980	8384	116	32
C Function	main()	1	26547	26547	36	36

The example in Figure 9–1 shows the Profile window with some default conditions:

- Column headings show the labels for the default set of profile data, including *Count, Inclusive, Incl-Max, Exclusive*, and *Excl-Max*.
- The data is sorted on the address of the first line in each area.
- All marked areas are listed, including disabled areas.

You can modify the Profile window to display selected profile areas or different data; you can also sort the data differently. The following subsections explain how to do these things.

Viewing different profile data

By default, the Profile window shows a set of statistics labeled as Count, Inclusive, Incl-Max, Exclusive, and Excl-Max. The Address field, which is not part of the default statistics, can also be displayed. Table 9–4 describes the statistic that each field represents.

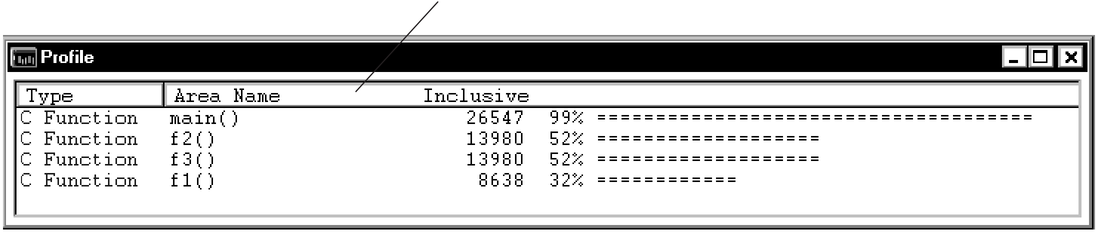
Table 9–4. Types of Data Shown in the Profile Window

Label	Profile Data
Count	The number of times a profile area is entered during a session
Inclusive	The total execution time (cycle count) of a profile area, including the execution time of any subroutines called from within the profile area
Incl-Max (inclusive maximum)	The maximum inclusive time for one iteration of a profile area
Exclusive	The total execution time (cycle count) of a profile area, excluding the execution time of any subroutines called from within the profile area In general, the exclusive data provides the best statistics for comparing the execution time of one profile area to another area.
Excl-Max (exclusive maximum)	The maximum exclusive time for one iteration of a profile area
Address	The memory address of the line. If the area is a function or range, the Address field shows the memory address of the first line in the area.

In addition to viewing this data in the default manner, you can view each of these statistics individually. The benefit of viewing them individually is that in addition to a cycle count, you are also supplied with a percentage indication and a histogram.

To view the fields individually, click the Area Name column heading in the Profile window.

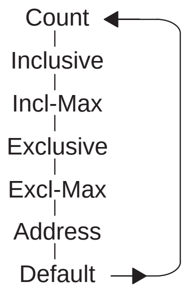
Click the Area Name column heading in the Profile window to change the type of data displayed



```
Type        Area Name          Inclusive
C Function  main()                 26547  99% =====================================
C Function  f2()                   13980  52% ===================
C Function  f3()                   13980  52% ===================
C Function  f1()                    8638  32% ============
```

When you click the Area Name column heading in the Profile window, fields are displayed individually in the order shown in Figure 9–2.

Figure 9–2. Cycling Through the Profile Window Fields



Note: Exclusive and Excl-Max are shown only when you run a full profile.

One advantage of using the mouse is that you can change the display while you are profiling.

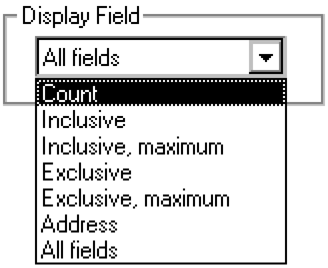
You can also use the Profile View dialog box to select the field you want to display. To do so, follow these steps:

1) Open the Profile View dialog box by using one of these methods:

   ❑ From the Tools→Profile menu, select Change View.
   ❑ From the context menu for the Profile window, select Change View.

   This displays the Profile View dialog box.

2) In the Display Field box, select the data field that you want to display:



3) Click OK.

Sorting profile data

By default, the data displayed in the Profile window is sorted according to the memory addresses of the displayed areas. The area with the least significant address is listed first, followed by the area with the next least significant address, etc. When you view fields individually, the data is automatically sorted from highest cycle count to lowest (instead of by address).

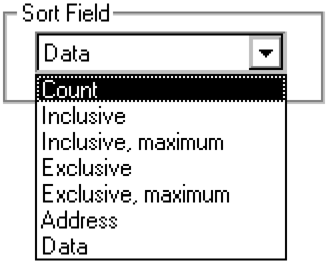
To sort the data on any of the data fields, follow these steps:

1) Open the Profile View dialog box by using one of these methods:

   - ❑ From the Tools→Profile menu, select Change View.
   - ❑ From the context menu for the Profile window, select Change View.

   This displays the Profile View dialog box.

2) In the Sort Field box, select the data field that you want to sort on:



3) Click OK.

For example, to sort all the data on the basis of values of the Inclusive field, select Inclusive in the Sort Field box. The area with the highest Inclusive field displays first, and the area with the lowest Inclusive field displays last. This applies even when you are viewing individual fields.

Viewing different profile areas

By default, all marked areas are listed in the Profile window. You can modify the window to display selected areas. To do this, follow these steps:

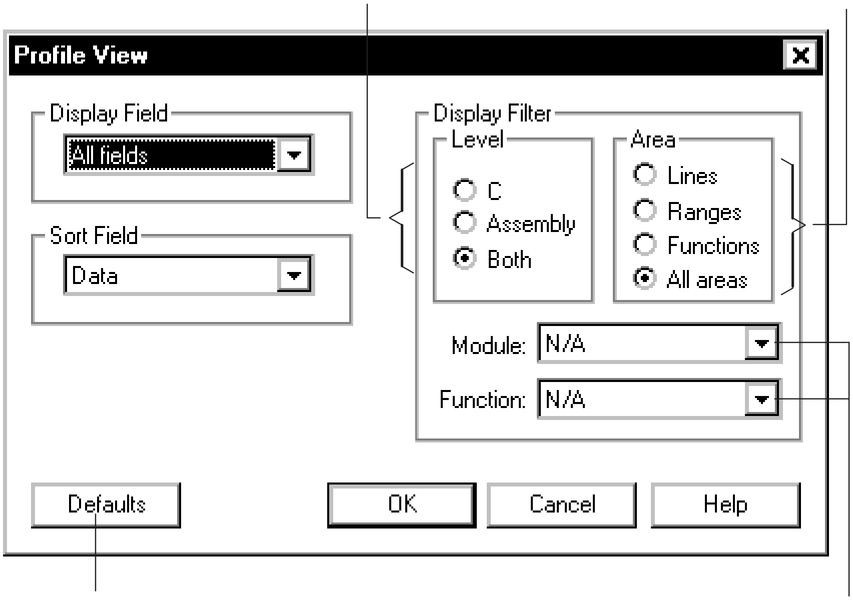
1) Open the Profile View dialog box by using one of these methods:

   - From the Tools→Profile menu, select Change View.
   - From the context menu for the Profile window, select Change View.

   This displays the Profile View dialog box.

Select to show C data, assembly data, or both.

Select to show line areas, range areas, function areas, or all areas.



Profile View

Display Field: All fields

Sort Field: Data

Display Filter

Level: ○ C ○ Assembly ● Both

Area: ○ Lines ○ Ranges ○ Functions ● All areas

Module: N/A

Function: N/A

To reset the Profile window to its default settings, click Defaults.

You can select a specific filename or function name to filter on.

2) In the Level box, select C, Assembly, or Both.

3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 9–3 on page 9-13 on for a list of valid combinations.

4) If you want to view areas within a specific file or function, do one of the following:

   - From the Module combo box, select a specific filename.
   - From the Function combo box, select a specific function name.

   See Table 9–3 on page 9-13 for a list of valid combinations.

5) Click OK.

If you want to reset the Profile window to its default characteristics, use the Profile View dialog box (Profile→Change View). Click the Defaults button, then click OK.

Interpreting session data

General information about a profiling session is displayed in the Command window during and after the session. This information identifies the starting and stopping points. It also lists statistics for three important areas:

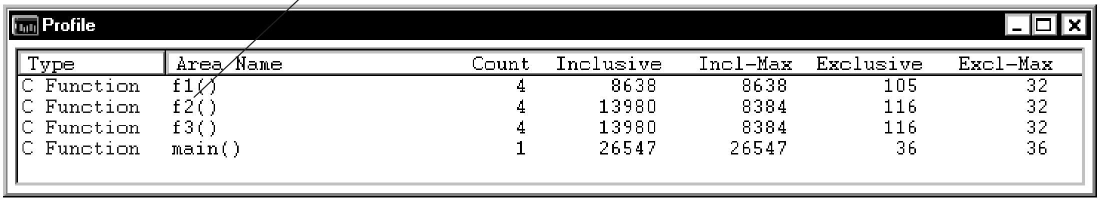
- *Run cycles* shows the number of execution cycles consumed by the program from the starting point to the stopping point.
- *Profile cycles* equals the run cycles minus the cycles consumed by disabled areas.
- *Hits* shows the number of internal breakpoints encountered during the profiling session.

Viewing code associated with a profile area

You can view the code associated with a displayed profile area. The debugger updates the display so that the associated C or disassembly statements are shown in the File or Disassembly window.

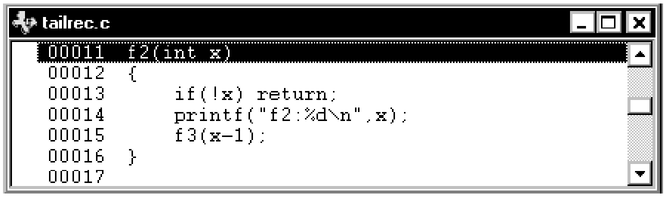
To select the profile area in the Profile window and display the associated code, double-click the area that you want to display:

Double-click an area to display the associated code.



Type	Area Name	Count	Inclusive	Incl-Max	Exclusive	Excl-Max
C Function	f1()	4	8638	8638	105	32
C Function	f2()	4	13980	8384	116	32
C Function	f3()	4	13980	8384	116	32
C Function	main()	1	26547	26547	36	36

If the area is a function name, the debugger opens a File window and displays that function:



```
f2(int x)
{
    if(!x) return;
    printf("f2:%d\n",x);
    f3(x-1);
}

```

If the area is in disassembly code, the debugger displays that code in the Disassembly window.

To view the code associated with another area, double-click another area.

If you are attempting to show disassembly, you might need to make several attempts, because you can access program memory only when the target is not running.

9.8 Saving Profile Data to a File

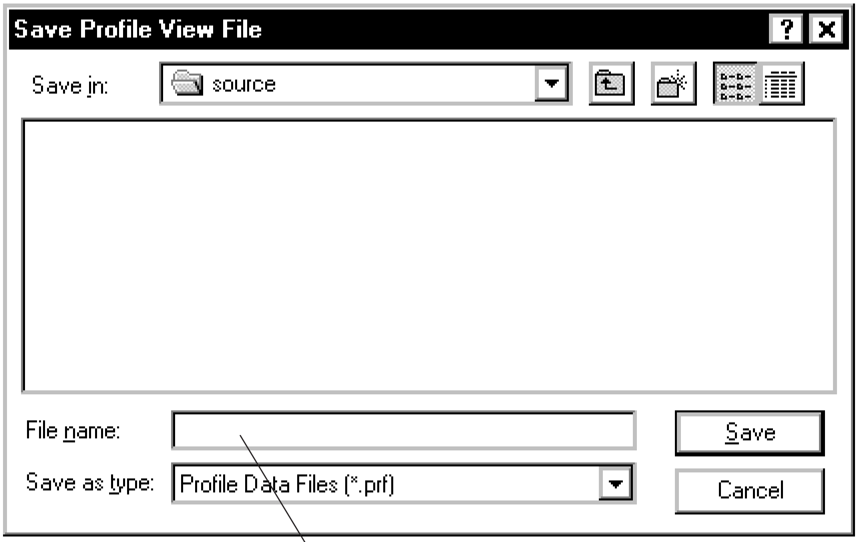
You may want to run several profiling sessions during a debugging session. Whenever you start a new profiling session, the results of the previous session are lost. However, you can save the results of the current profiling session to a system file.

The saved file contents are in ASCII and are formatted in exactly the same manner as they are displayed (or would be displayed) in the Profile window. The general profiling-session information that is displayed in the Command window is also written to the file.

Saving the contents of the Profile window

To save the contents of the Profile window to a system file, follow these steps:

1) From the Tools→Profile menu, select Save View. This displays the Save Profile View File dialog box:



Enter a name for the file. Use a .prf extension.

2) In the File name field, enter a name for the file. You can use a .prf extension to identify the file as a profile data file.

3) Click Save.

This saves only the current view; if, for example, you are viewing only the Count field, then only that information is saved. If the file already exists, debugger overwrites the file with the new data.

Saving all data for currently displayed areas

To save all data for the currently displayed areas, follow these steps:

1) From the Tools→Profile menu, select Save All. This displays the Save Profile File dialog box.

2) In the File name field, enter a name for the file. You can use a .prf extension to identify the file as a profile data file.

3) Click Save.

This saves all views of the data—including the individual count, inclusive, etc.—with the percentage indications and histograms. If the file already exists, debugger overwrites the file with the new data.

Chapter 10

Monitoring Hardware Functions with the Emulator Analysis Module

The 'C54x has an on-chip analysis module that allows the emulator to monitor hardware functions. Using the analysis module, you can count occurrences of certain hardware functions or set hardware breakpoints on these occurrences.

You access the analysis features through dialog boxes described in this chapter. These dialog boxes provide a transparent means of loading the special set of pseudoregisters that the debugger uses to access the on-chip analysis module.

Topic **Page**

10.1 Major Functions of the Analysis Module

The ’C54x analysis module provides a detailed look into events occurring in hardware, expanding your debugging capabilities beyond software breakpoints. The analysis module examines ’C54x bus cycle information in real time and reacts to this information through actions such as hardware breakpoints and event counting. The analysis module allows you to:

- **Count events.** The analysis module has an *internal counter* that can count nine types of *events*. You can count the number of times a defined bus event or other internal event occurs during execution of your program.

  Events that can be counted include:

  - 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 module to halt the processor during execution of your program. The events that cause the processor to stop are called *break events*. You can define a break event as one or more of the following conditions:

  - 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. In addition, any of the debugger’s basic features available with software breakpoints can also be used with hardware breakpoints. As a result, you can take advantage of all the step and most of the run commands.

❑ **Set global breakpoints with EMU0/1 pins.** In a system of multiple ’C54x 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.

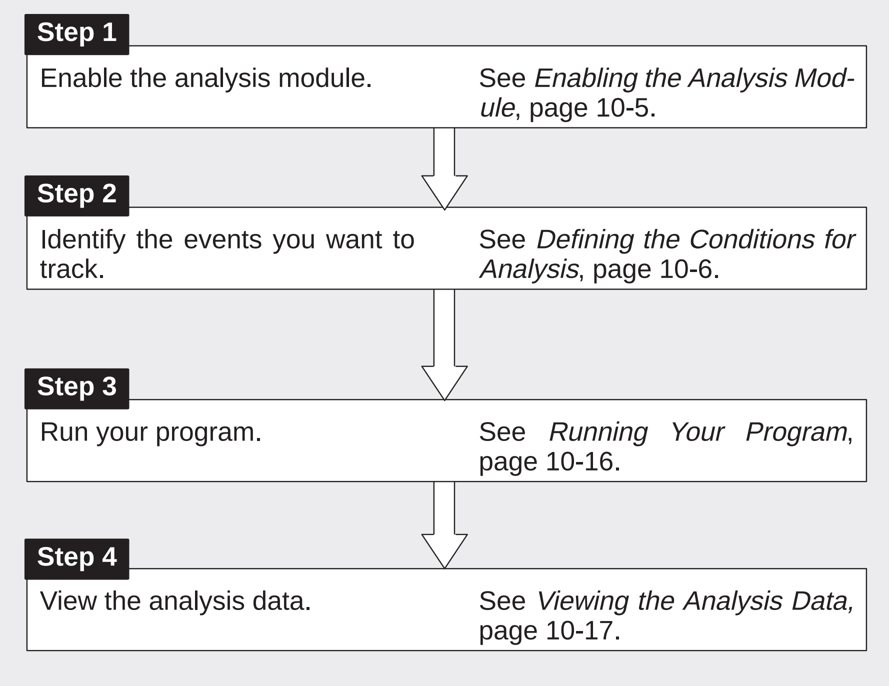
In addition to setting global breakpoints, you can set up the EMU0/1 pins to take advantage of the emulator’s external counter. The *external counter* keeps track of the internal counter; each time the internal counter passes 0, a signal is sent through the EMU0/1 pins, incrementing the external counter.

❑ **Track 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.

10.2 Overview of the Analysis Process

Completing an analysis session consists of four simple steps:



Step 1

Enable the analysis module.

See *Enabling the Analysis Module*, page 10-5.

Step 2

Identify the events you want to track.

See *Defining the Conditions for Analysis*, page 10-6.

Step 3

Run your program.

See *Running Your Program*, page 10-16.

Step 4

View the analysis data.

See *Viewing the Analysis Data,* page 10-17.

10.3 Enabling the Analysis Module

When the debugger comes up, analysis is disabled by default. To begin tracking system events, you must explicitly enable analysis by selecting Enable Events on the Tools→Analysis menu. When you select Enable Events, a check mark appears next to the menu item, indicating anaysis is enabled. To disable analysis, select Enable Events again, and the check mark disappears, indicating analysis is disabled.

When analysis is disabled, all the events you previously enabled remain unchanged. You can simply reenable analysis and use the events you already defined.

During a single debugging session, you may want to change the parameters of the analysis module several times. For example, you may want to define new parameters such as 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 must enable the analysis module only once during a debugging session. It is not necessary to enable the analysis module each time you run your program.

10.4 Defining the Conditions for Analysis

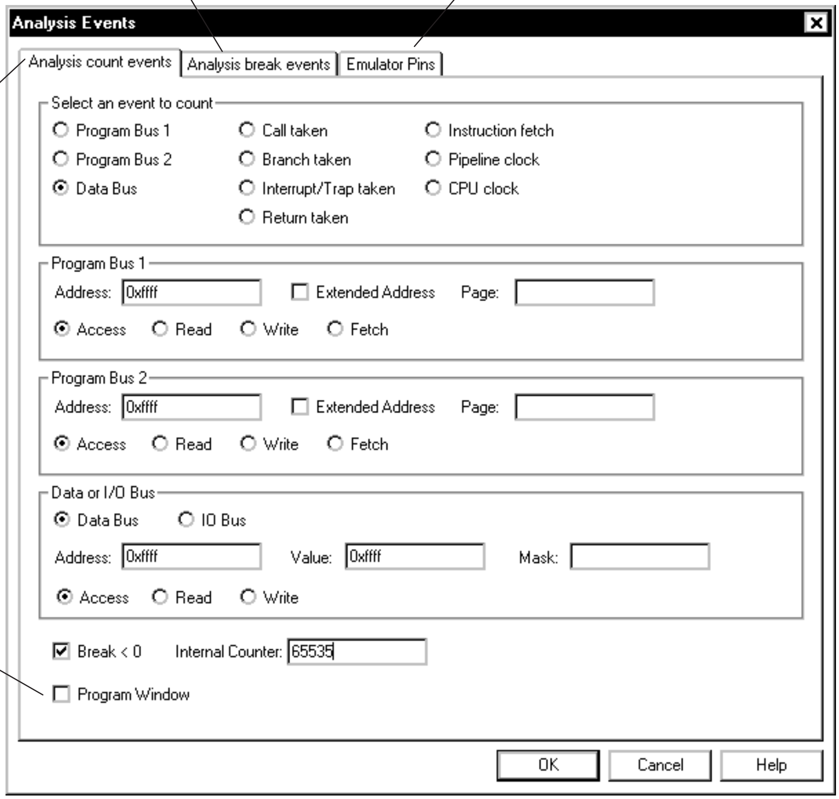
The analysis module detects hardware events and monitors the internal signals of the processor according to the parameters you define that count events or halt the processor.

You must define the conditions the analysis module must meet to track a particular event. To do this, select the events you want to track by enabling the appropriate conditions in the Analysis Events dialog boxes. To bring up the Analysis Events dialog boxes, select the Setup Events... option found on the Tools→Analysis menu.

Figure 10–1. Analysis Events Dialog Boxes

Analysis break events tab

Emulator pins tab



Analysis Events

Analysis count events | Analysis break events | Emulator Pins

Select an event to count

○ Program Bus 1 ○ Call taken ○ Instruction fetch
○ Program Bus 2 ○ Branch taken ○ Pipeline clock
● Data Bus ○ Interrupt/Trap taken ○ CPU clock
○ Return taken

Program Bus 1

Address: 0xffff □ Extended Address Page:

● Access ○ Read ○ Write ○ Fetch

Program Bus 2

Address: 0xffff □ Extended Address Page:

● Access ○ Read ○ Write ○ Fetch

Data or I/O Bus

● Data Bus ○ IO Bus

Address: 0xffff Value: 0xffff Mask:

● Access ○ Read ○ Write

☑ Break < 0 Internal Counter: 65535

□ Program Window

Analysis count events tab

Enables and disables the Program window

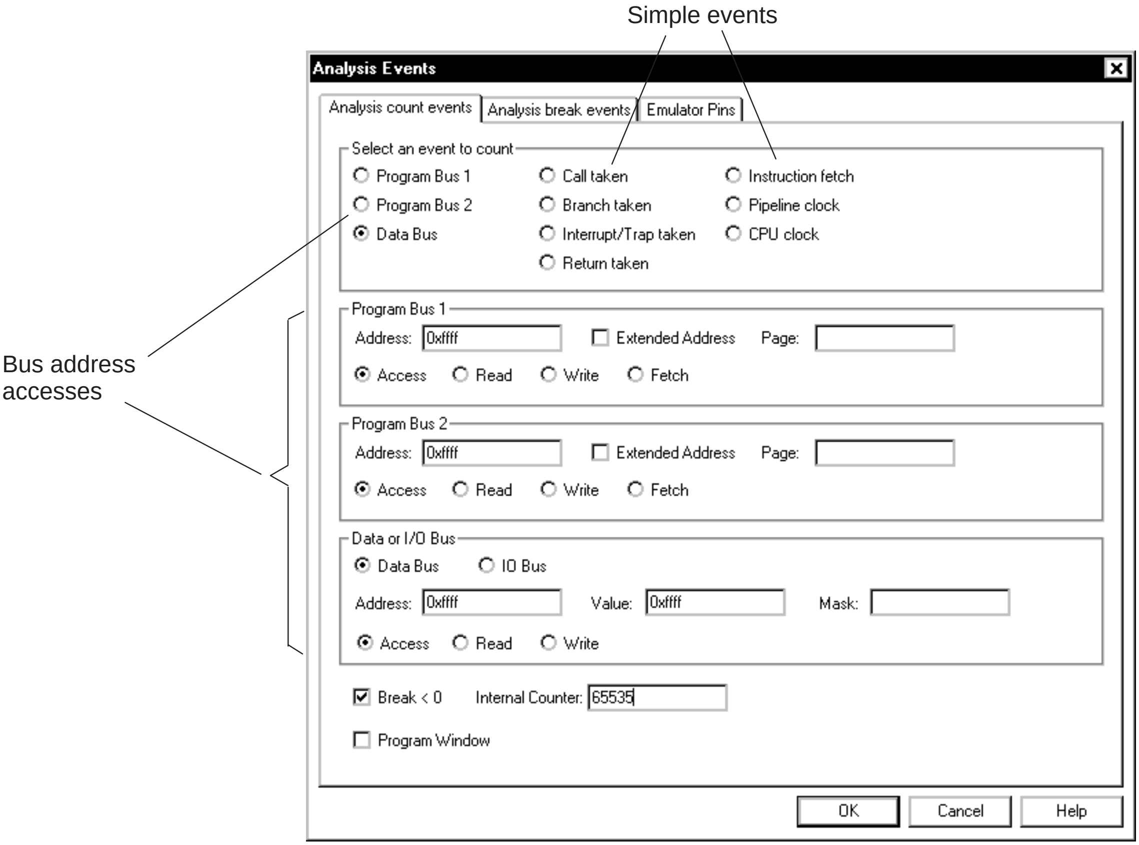
Counting events

The analysis module's internal counter counts bus events and detects other internal events. This counter keeps track of how many times an event occurs. The Analysis count events dialog box allows you to count two basic types of events:

- ❑ Simple events
- ❑ Bus address accesses

Figure 10–2 shows the Analysis count events dialog box and the types of events that you can select. To count any of the events, simply select that event. You can count only one type of event at a time.

Figure 10–2. The Two Basic Types of Events That Can Be Counted



Analysis Events

Analysis count events | Analysis break events | Emulator Pins

Select an event to count

○ Program Bus 1 ○ Call taken ○ Instruction fetch
○ Program Bus 2 ○ Branch taken ○ Pipeline clock
● Data Bus ○ Interrupt/Trap taken ○ CPU clock
○ Return taken

Program Bus 1
Address: 0xffff □ Extended Address Page:
● Access ○ Read ○ Write ○ Fetch

Program Bus 2
Address: 0xffff □ Extended Address Page:
● Access ○ Read ○ Write ○ Fetch

Data or I/O Bus
● Data Bus ○ IO Bus
Address: 0xffff Value: 0xffff Mask:
● Access ○ Read ○ Write

☑ Break < 0 Internal Counter: 65535

□ Program Window

OK | Cancel | Help

To watch the progress of the event counter, view the status of the event in the Analysis Statistics window.

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 occurs, simply enable the event in the Analysis count events dialog box as shown in Figure 10–2.

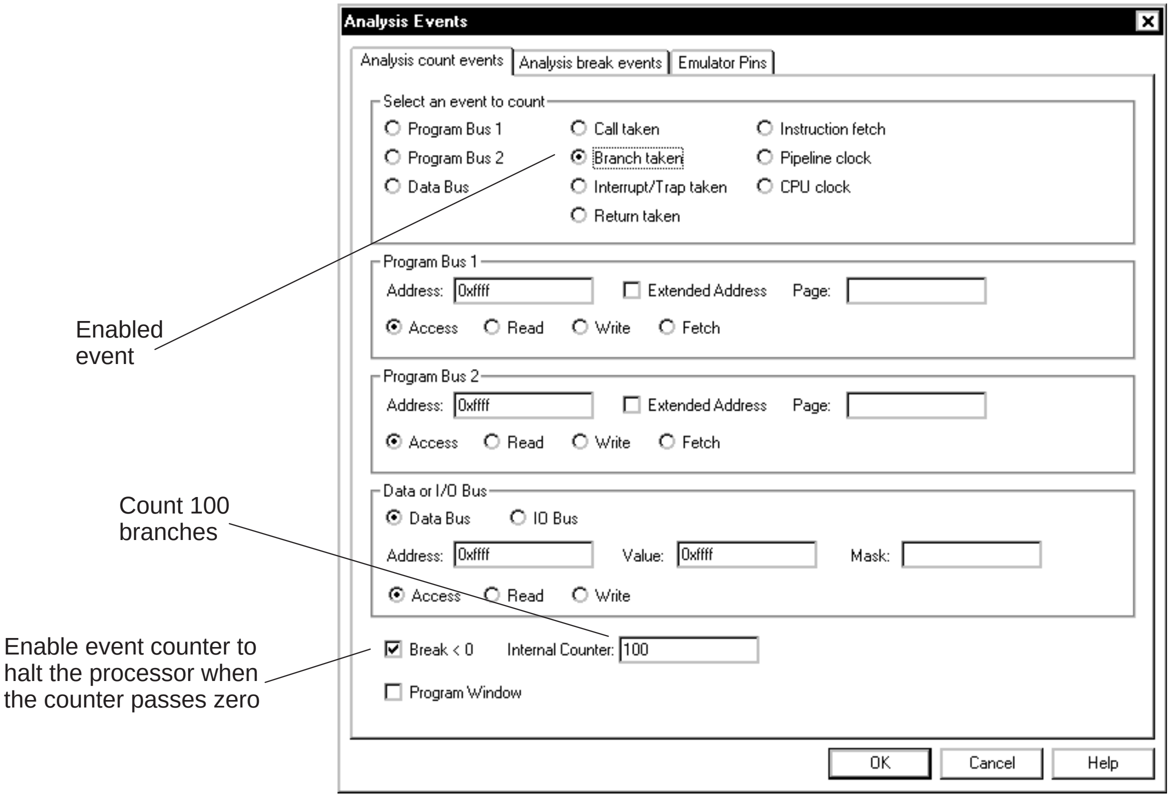
The analysis module has an internal counter that counts bus events as well as detects other internal events. This counter keeps track of how many times an event occurs. To stop after a certain number of events are detected, follow these steps:

❑ Specify the event you want to count
❑ Enable the internal counter *(Break < 0*)
❑ 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 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 halts the processor. Figure Figure 10–3 illustrates how to count 100 branches and to halt the processor on the 100th branch.

Figure 10–3. Enabling the Event Counter



Analysis Events

Analysis count events | Analysis break events | Emulator Pins

Select an event to count

○ Program Bus 1 ○ Call taken ○ Instruction fetch
○ Program Bus 2 ● Branch taken ○ Pipeline clock
○ Data Bus ○ Interrupt/Trap taken ○ CPU clock
○ Return taken

Program Bus 1

Address: 0xffff □ Extended Address Page:
● Access ○ Read ○ Write ○ Fetch

Program Bus 2

Address: 0xffff □ Extended Address Page:
● Access ○ Read ○ Write ○ Fetch

Data or I/O Bus

● Data Bus ○ IO Bus
Address: 0xffff Value: 0xffff Mask:
● Access ○ Read ○ Write

☑ Break < 0 Internal Counter: 100

□ Program Window

Enabled event

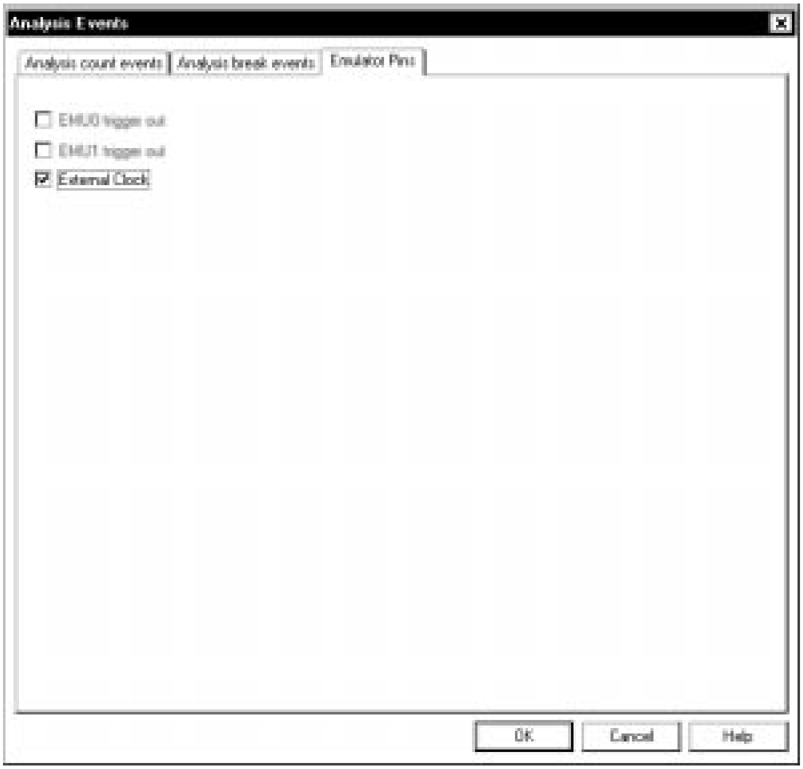
Count 100 branches

Enable event counter to halt the processor when the counter passes zero

To watch the progress of the event counter, open the Analysis Statistics window using the View menu. For more information on the Analysis Statistics window, see Section 10.6 on page 10-17.

Enabling the external counter

The emulator's *external counter* (clock) 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 select the External Clock checkbox in the Emulator Pins dialog box.



□ EMU0 trigger out

□ EMU1 trigger out

☑ External Clock

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.

Setting hardware breakpoints

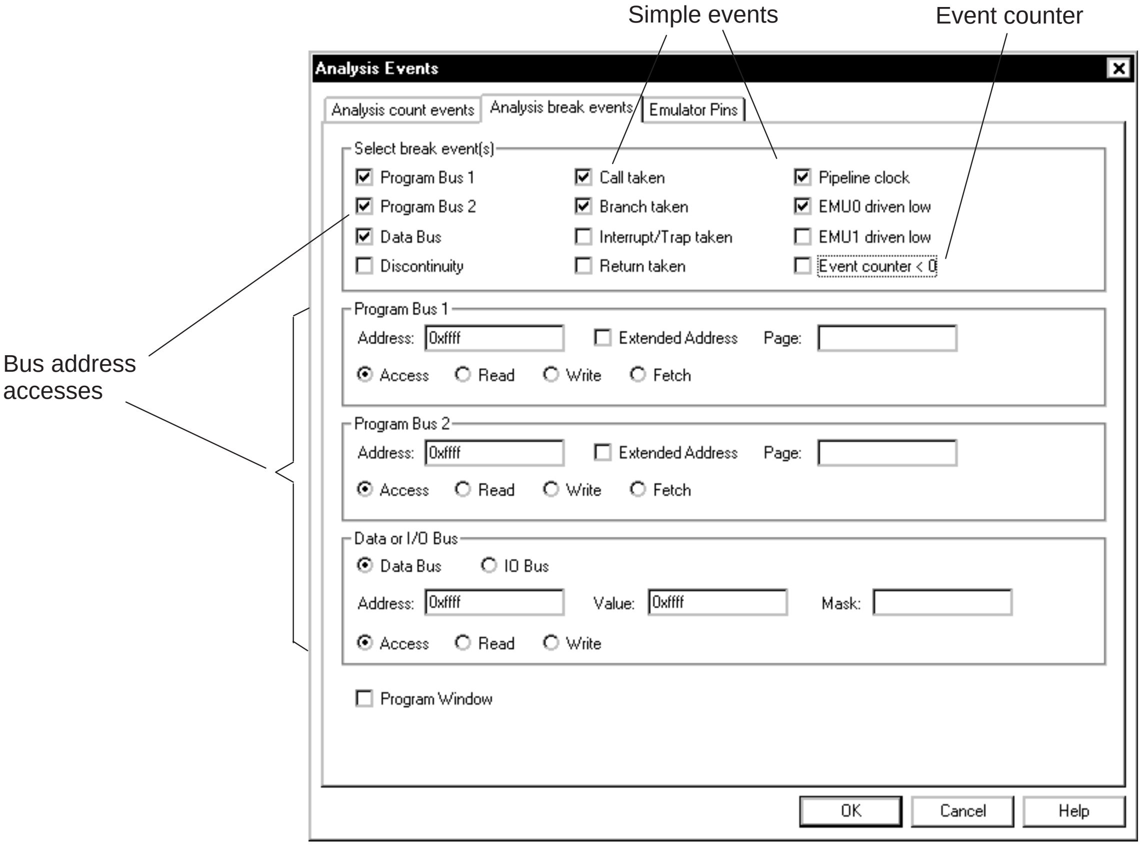
You can set a hardware breakpoint, which halts the processor, on three types of events:

- ❑ The event counter passing 0
- ❑ Simple events
- ❑ Bus address accesses

You can select as many events as you want. To specify one or more events on which to halt the processor, follow these steps:

1) From the Tools→Analysis menu, select Setup Events....

2) Select the Analysis break events tab from the Analysis Events dialog box. The Analysis break events dialog box appears:



Select break event(s)

☑ Program Bus 1 ☑ Call taken ☑ Pipeline clock
☑ Program Bus 2 ☑ Branch taken ☑ EMU0 driven low
☑ Data Bus □ Interrupt/Trap taken □ EMU1 driven low
□ Discontinuity □ Return taken □ Event counter < 0

Program Bus 1
Address: 0xffff □ Extended Address Page:
● Access ○ Read ○ Write ○ Fetch

Program Bus 2
Address: 0xffff □ Extended Address Page:
● Access ○ Read ○ Write ○ Fetch

Data or I/O Bus
● Data Bus ○ IO Bus
Address: 0xffff Value: 0xffff Mask:
● Access ○ Read ○ Write

□ Program Window

3) Select the event or events on which you want to set a hardware breakpoint by clicking the check box(es) next to that event or events.

The events selected cause the debugger to halt the processor whenever the occurrence of a call taken, a branch taken, a pipeline clock, the EMU0 pin being driven low, or a program or data bus being accessed is detected.

If you want to enable a hardware breakpoint at a particular program or data address, you can enter the address as a symbol or value in any format.

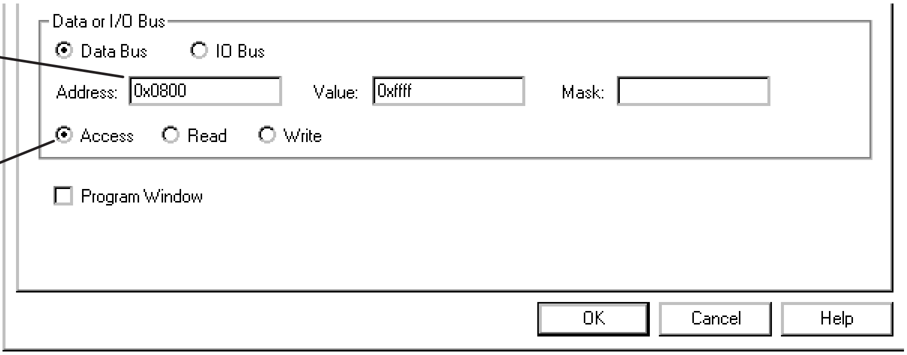
Setting up the event comparators

The analysis module has separate event comparators for the program and data buses. You can set up analysis 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. 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 select Data Bus, load the data address field with 0x0800 and select Access in the Analysis break events dialog box, the processor halts every time a read or write occurs on the data bus at address 0x0800:



Data or I/O Bus

● Data Bus ○ IO Bus

Address: 0x0800 Value: 0xffff Mask:

● Access ○ Read ○ Write

□ Program Window

Break when a read or write occurs at address 0x0800 on the data bus.

Look for read *or* writes.

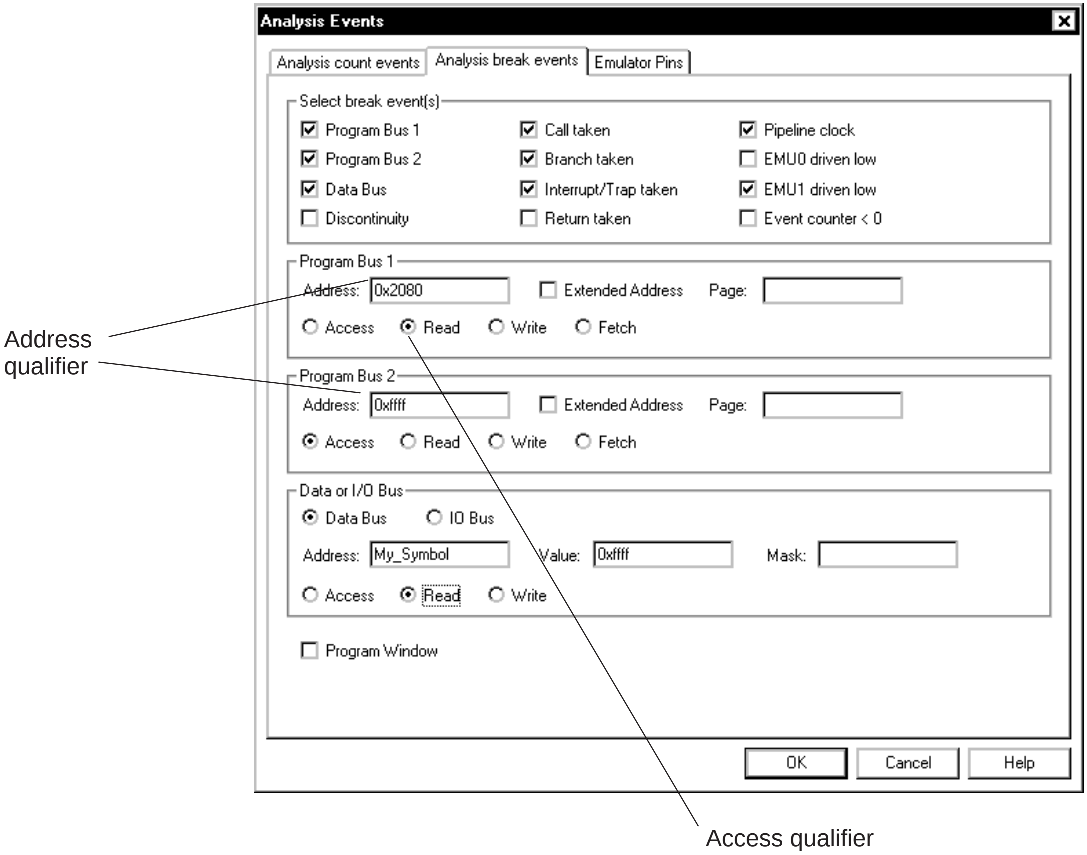
Enabling one of the program or data bus comparators in the Analysis count events dialog box allows you to count the number of accesses that are detected or to stop processing after a certain number of accesses have occurred.

In Figure 10–4, 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 10–4. Enabling Break Events



Analysis Events

Analysis count events | Analysis break events | Emulator Pins

Select break event(s)

☑ Program Bus 1 ☑ Call taken ☑ Pipeline clock
☑ Program Bus 2 ☑ Branch taken □ EMU0 driven low
☑ Data Bus ☑ Interrupt/Trap taken ☑ EMU1 driven low
□ Discontinuity □ Return taken □ Event counter < 0

Program Bus 1
Address: 0x2080 □ Extended Address Page:
○ Access ● Read ○ Write ○ Fetch

Program Bus 2
Address: 0xffff □ Extended Address Page:
● Access ○ Read ○ Write ○ Fetch

Data or I/O Bus
● Data Bus ○ IO Bus
Address: My_Symbol Value: 0xffff Mask:
○ Access ● Read ○ Write

□ Program Window

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 10–4, the debugger halts the processor any time a read from the address at My_Symbol occurs.

When the Program Window checkbox is selected, the debugger compares only the values that occur in the logical window defined by program breakpoints 1 and 2. The program breakpoint 1 address is the window start address and the program breakpoint 2 address is the window end address. The resulting logical window represents the mask value that the debugger uses to check for the data value.

If the mask value is 0, then any value in the given data address is 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 are 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 data mask 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.

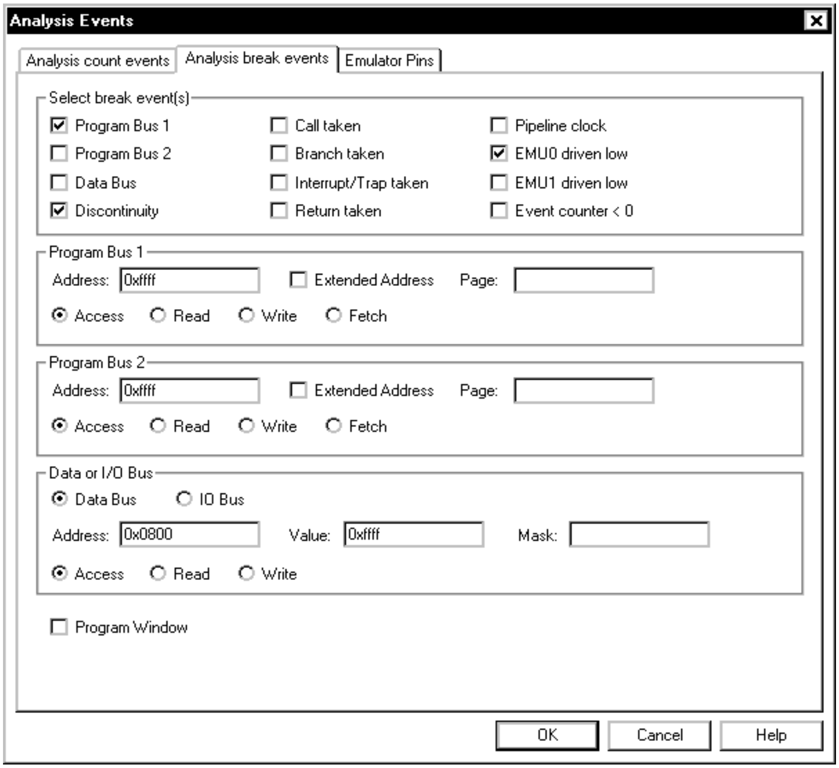
Setting up the EMU0/1 pins to set global breakpoints

To set the EMU0/1 pins to output, select the EMU0 trigger out or EMU1 trigger out checkbox in the Emulator Pins dialog box.

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 ’C54x 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 driven low option in the Analysis break events dialog box of one of the processors, as shown in Figure 10–5.

Figure 10–5. Setting Up Global Breakpoints on a System of Two ’C54x Processors



Select break event(s)

☑ Program Bus 1	□ Call taken	□ Pipeline clock
□ Program Bus 2	□ Branch taken	☑ EMU0 driven low
□ Data Bus	□ Interrupt/Trap taken	□ EMU1 driven low
☑ Discontinuity	□ Return taken	□ Event counter < 0

Program Bus 1
Address: 0xffff □ Extended Address Page:
● Access ○ Read ○ Write ○ Fetch

Program Bus 2
Address: 0xffff □ Extended Address Page:
● Access ○ Read ○ Write ○ Fetch

Data or I/O Bus
● Data Bus ○ IO Bus
Address: 0x0800 Value: 0xffff Mask:
● Access ○ Read ○ Write

□ Program Window

When processor 1 halts, its EMU0 signal halts processor 2. Setting up each processor in this way creates a global breakpoint so that any processor that reaches a breakpoint halts all other processors in the system.

10.5 Running Your Program

Once you have defined your parameters, the analysis module can begin collecting data as soon as you run your program. It will stop collecting data when the defined conditions are met. The analysis module monitors the progress of the defined events while your program is running.

> **Note:**
>
> The conditions for the analysis session must be defined *before* your analysis session begins; you cannot change conditions *during* execution of your program.

How to run the entire program

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

- Click the Run icon on the toolbar:
- From the Debug menu, select Run.
- Press F5.
- From the command line, enter the RUN command. The format for this command is:

  run [*expression*]

You can use any of the debugger run commands (STEP, CSTEP, NEXT, etc.) described in Chapter 7 except the RUNB (run benchmarks) or RUNF (run free) command.

How the Run Benchmark (RUNB) command affects analysis

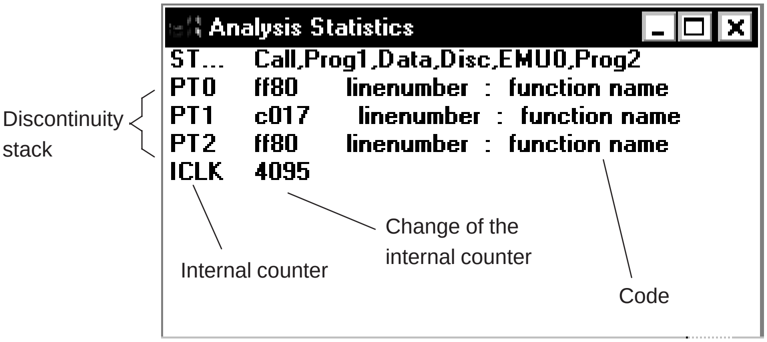
Running your program by selecting the Run Benchmarks option from the Debug menu or entering the RUNB command from the command line disables the current analysis settings and configures the counter to count CPU clock cycles. When the processor is halted after a RUNB, the analysis registers are restored to their original states.

The analysis module provides capabilities in addition to those provided by the RUNB command. With the RUNB command you can count the number of CPU clock cycles only during the execution of a specific section of code. However, the analysis module not only allows you to count CPU clock cycles, it also allows you to count other events.

10.6 Viewing the Analysis Data

You can monitor the status of the analysis module by checking the Analysis Statistics window. This window displays an ongoing progress report of the analysis module's activity. Through this window, you can monitor the status of the break events, the value of both the internal and external event counters, and the values of the PC discontinuity stack. Figure 10–6 illustrates the Analysis Statistics window.

Figure 10–6. Analysis Module View Window, Displaying an Ongoing Status Report



```
Analysis Statistics
ST...   Call,Prog1,Data,Disc,EMU0,Prog2
PT0     ff80    linenumber  :  function name
PT1     c017     linenumber  :  function name
PT2     ff80    linenumber  :  function name
ICLK    4095
```

Interpreting the status field

Multiple events can cause the processor to halt at the same time; these events are reflected in the ST field of the Analysis Statistics window. The ST field displays a list of the events that caused the processor to halt. If the analysis module itself did not halt the processor, but something else (such as a software breakpoint) did, then the status line displays "No event detected".

Interpreting the discontinuity stack

The PC discontinuity stack allows you to see how the program reached its current position. The Analysis Statistics 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.

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 causes 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 Statistics 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 11

Using the Simulator Analysis Module

The ’C54x has an on-chip analysis module that allows the simulator to simulate hardware functions. Using the analysis module, you can set data breakpoints on the occurrences of certain simulated hardware functions.

You access the features through dialog boxes described in this chapter. These dialog boxes provide a transparent means of loading the special set of pseudo-registers that the debugger uses to access the on-chip analysis module.

Topic **Page**

11.1 Major Functions of the Analysis Module

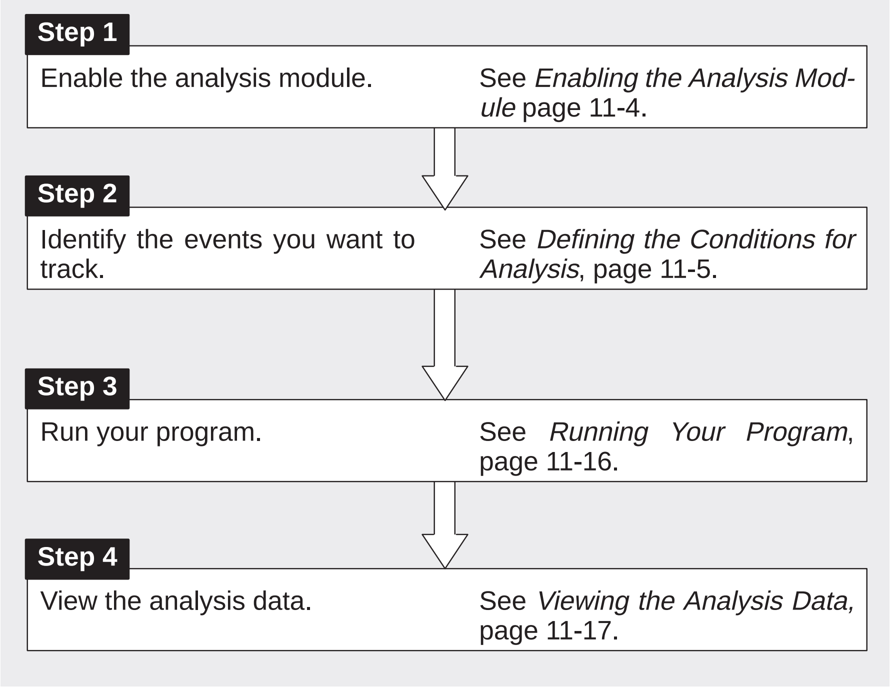
The ’C54x analysis module gives a detailed look into events occurring in hardware, expanding your debugging capabilities beyond software breakpoints. The analysis module examines ’C54x bus cycle information in real time and reacts to this information through actions such as data breakpoints.

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 Overview of the Analysis Process

Completing an analysis session consists of four simple steps:



Step 1

Enable the analysis module.

See *Enabling the Analysis Module* page 11-4.

Step 2

Identify the events you want to track.

See *Defining the Conditions for Analysis*, page 11-5.

Step 3

Run your program.

See *Running Your Program*, page 11-16.

Step 4

View the analysis data.

See *Viewing the Analysis Data,* page 11-17.

11.3 Enabling the Analysis Module

When the debugger comes up, analysis is disabled. To begin tracking system events, you must explicitly enable the interface by selecting Enable Events on the Tools→Analysis menu. When you select Enable Events, a checkmark appears next to the menu item, indicating anaysis is enabled. Also, the Analysis Window is either opened, if it is not already, or is brought to the top. To disable analysis, select Enable Events again, and the checkmark disappears, indicating analysis is disabled.

When analysis is disabled, all the events you previously enabled remain unchanged. You can simply reenable analysis and use the events you have already defined.

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

Note:

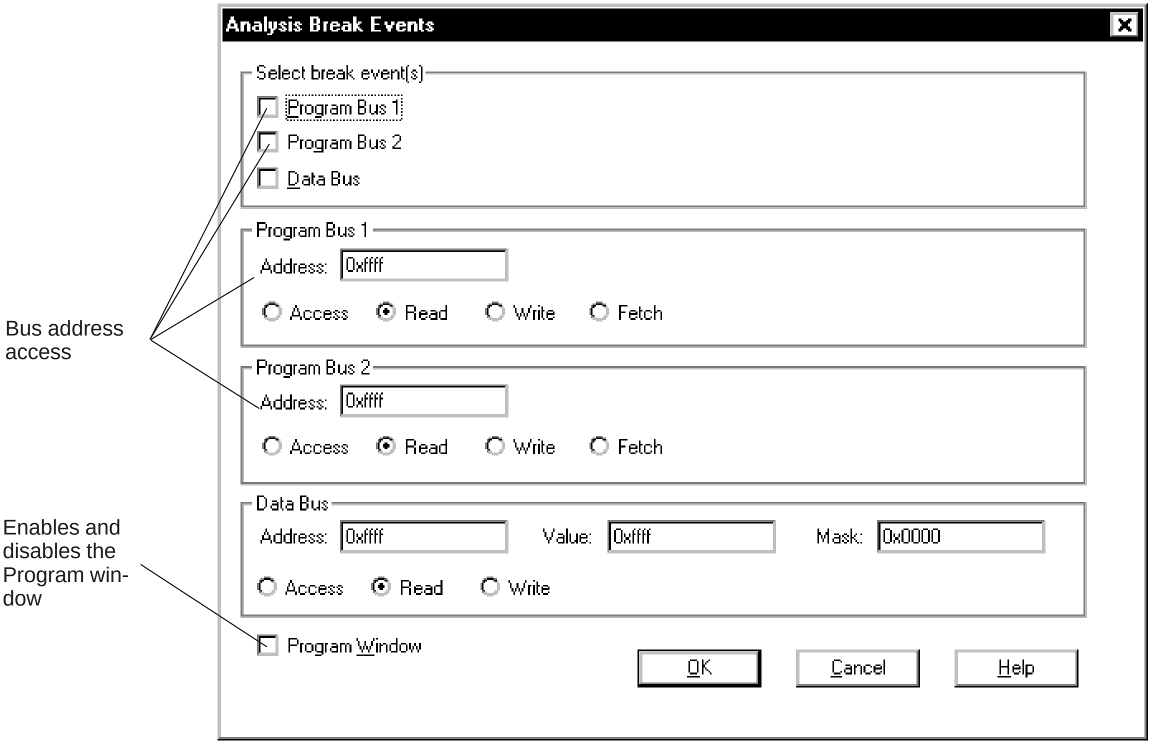
You must enable the analysis module only once during a debugging session. It is not necessary to enable the analysis module each time you run your program.

11.4 Defining the Conditions for Analysis

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

You must define the conditions that the analysis module must meet to track a particular event. To do this, select the events you want to track by enabling the appropriate conditions in the Analysis Break Events dialog box. To bring up the Analysis Break Events dialog box, select the Setup Events... option found on the Tools→Analysis menu.

Figure 11–1. Analysis Events Dialog Boxes



Analysis Break Events

Select break event(s)

□ Program Bus 1
□ Program Bus 2
□ Data Bus

Program Bus 1

Address: 0xffff

○ Access ● Read ○ Write ○ Fetch

Program Bus 2

Address: 0xffff

○ Access ● Read ○ Write ○ Fetch

Data Bus

Address: 0xffff Value: 0xffff Mask: 0x0000

○ Access ● Read ○ Write

□ Program Window

Bus address access

Enables and disables the Program window

Setting hardware breakpoints

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) 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

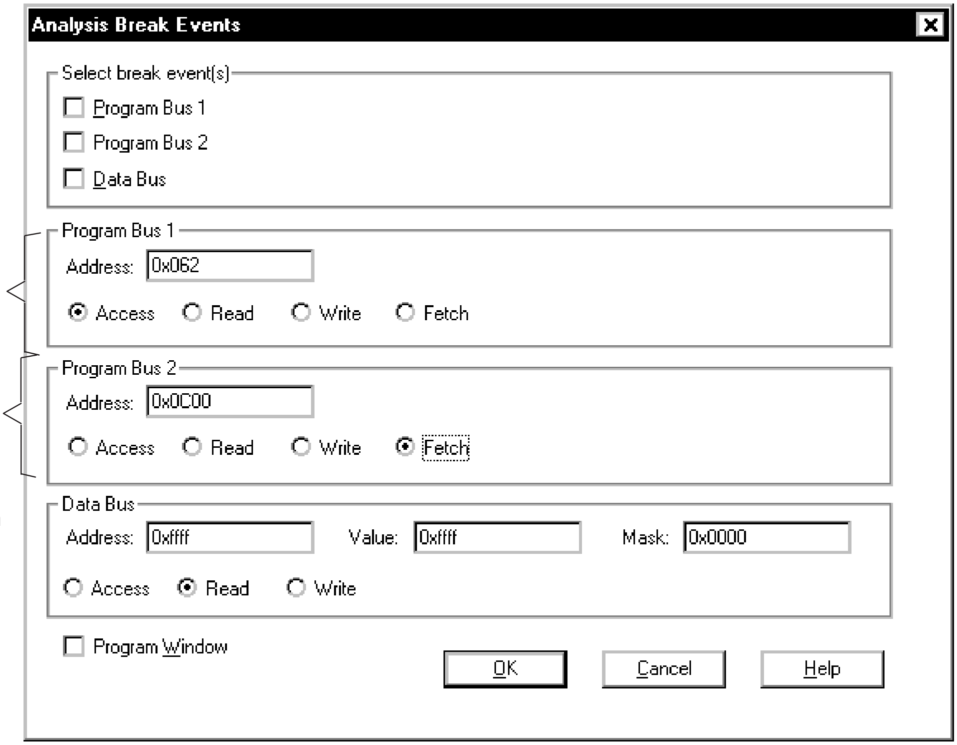
(b) Data memory

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

You can 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.

To specify one or more events on which to halt the processor, follow these steps:

1) From the Tools→Analysis menu, select Setup Events....
   The Analysis break events dialog box appears:



Analysis Break Events

Select break event(s)
□ Program Bus 1
□ Program Bus 2
□ Data Bus

Program Bus 1
Address: 0x062
● Access ○ Read ○ Write ○ Fetch

Program Bus 2
Address: 0x0C00
○ Access ○ Read ○ Write ● Fetch

Data Bus
Address: 0xffff Value: 0xffff Mask: 0x0000
○ Access ● Read ○ Write

□ Program Window

Break when a read or write occurs at address 0x062 on program bus 1.

Break when an instruction fetch occurs at address 0x0C00 on program bus 2.

2) Select the event or events on which you want to set a hardware breakpoint by clicking the check box(es) next to that event or events.

The events selected cause the debugger to halt the processor whenever a program or data bus being accessed is deleted.

If you want to enable a hardware breakpoint at a particular program or data bus address, you can enter the address as a symbol or value in any format.

Setting up the event comparators

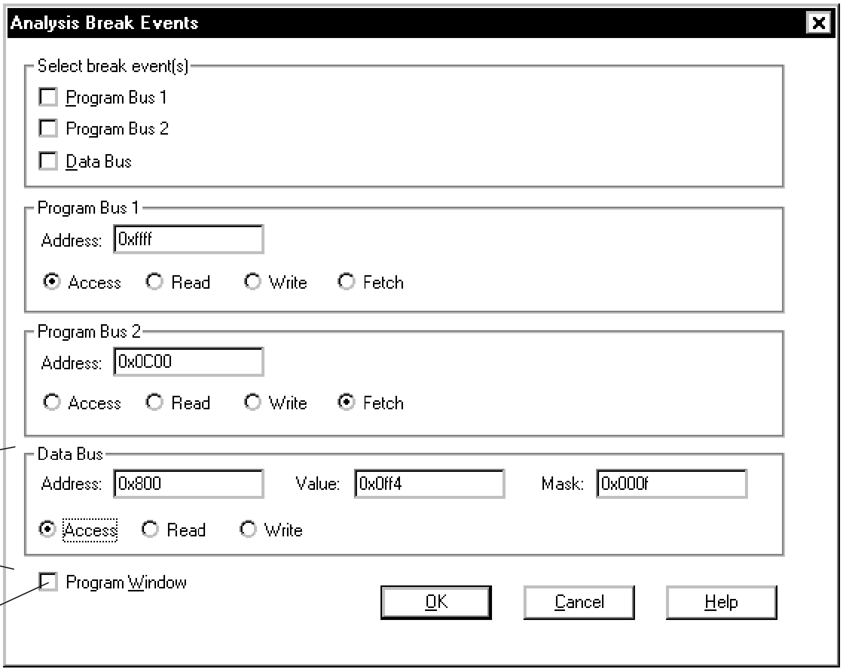
The analysis module has separate event comparators for the program and data buses. You can set up the analysis module to halt the processor on accesses to a specified address.

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.

The data bus supports noninstruction read and write accesses. You can specify data value or a mask value.

In Figure 11–2, 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 11–2. Set Up of Event Comparator



Analysis Break Events

Select break event(s)
□ Program Bus 1
□ Program Bus 2
□ Data Bus

Program Bus 1
Address: 0xffff
● Access ○ Read ○ Write ○ Fetch

Program Bus 2
Address: 0x0C00
○ Access ○ Read ○ Write ● Fetch

Data Bus
Address: 0x800 Value: 0x0ff4 Mask: 0x000f
● Access ○ Read ○ Write

☑ Program Window

Break when a read or write occurs at address 0x0800 on the data bus with a last hex digit of 4.

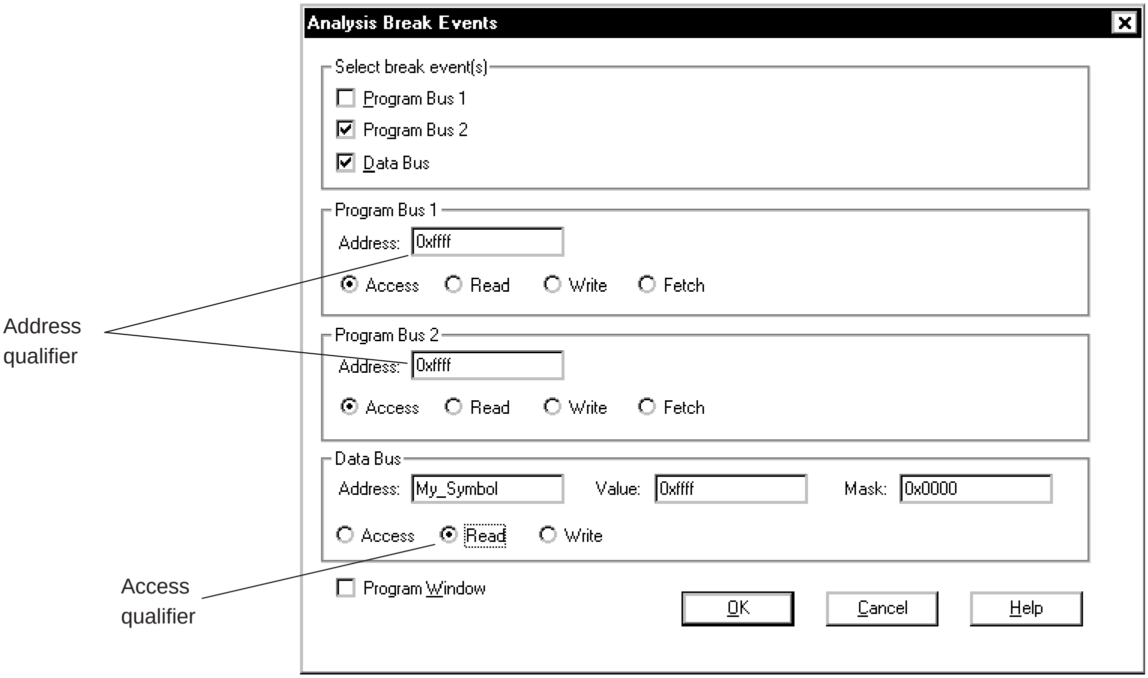
Look for reads *or* writes

In Figure 11–3, 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 11–3. Enabling Break Events



Analysis Break Events

Select break event(s)
- □ Program Bus 1
- ☑ Program Bus 2
- ☑ Data Bus

Program Bus 1
Address: 0xffff
● Access ○ Read ○ Write ○ Fetch

Program Bus 2
Address: 0xffff
● Access ○ Read ○ Write ○ Fetch

Data Bus
Address: My_Symbol Value: 0xffff Mask: 0x0000
○ Access ● Read ○ Write

□ Program Window

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 11–3, the debugger halts the processor any time a read from the address at My_Symbol occurs.

When the Program Window checkbox is selected, the debugger compares only the values that occur in the logical window defined by program breakpoints 1 and 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 is 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 are 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.

Instruction pipelining

The ’C54x debugger uses a 6-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 for input in 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 6-phase pipeline diagram shows you where in the pipeline a program address breakpoint actually halts the simulator.

```
              | P | F | D | A | R | E |

fetch :               |—> halt

read  :                                 |—> halt

write :                                 |—> halt
```

  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 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   @62h, A    ; Add location 0x62 to accumulator A
2  ADD   @65h, A    ; Add location 0x65 to accumulator A
3  DST   A,*ar5     ; Store A (high) to 0x63 and A (low) to 0x62
4  ADD   @62h, A    ; Add location 0x62 to A; 0x62 has 0x3956
5  ANDM  0000h, *ar4; AND 0 with location 0x62; put result back
7  ORM   0900h, *ar4; OR 0x0900 with loc 0x62; put result back
9  ADD   @62h, A    ; Add location 0x62 to A; 0x62 now 0x0900
10 NOP
```

If you execute the above instruction sequence, you see that data breakpoints are detected by the simulator. The data breakpoints detected are caused by the instruction at line 4, then the instruction at line 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   @62h,A     ; Add location 0x62h to accumulator A
2  ADD   @65h,A     ; Add location 0x65 to accumulator A
3  DST   A,*ar5     ; Store A(high) to 0x63 and A(low) to 0x62
4  ADD   @62h,A     ; Add location 0x62 to A; 0x62 has 0x3956
5  ANDM  0000h,*ar4 ; AND 0 with location 0x62 put result back
7  ORM   0900h,*ar4 ; OR 0x0900 with loc 0x62; put result back
9  ADD   @62h,A     ; Add location 0x62 to A; 0x62 now 0x0900
10 NOP
11 NOP
12 ADD    @62h, A   ; Add location 0x62 to A; 0x62 now 0x0900
13 NOP
14 NOP
```

If you execute the above instruction sequence, you see that a data breakpoint is detected by the simulator. The data breakpoint is caused only by the instruction at line 9. Data breakpoints are not detected at lines 4 and 12 because they are outside the program window.

Using Example 11–2, if you set the program window as follows, data breakpoints are detected at lines 4 and 9:

❑ program_window start address = 0x4
❑ program_window end address = 0x10

A data breakpoint is detected at line 4 because the instruction at program_window start is inclusive within the program window. Refer to *Instruction pipelining*, page 11-11, for information on detecting breakpoints at the six pipeline phases.

Using Example 11–2, if you set the program window as follows, data breakpoints is detected at lines 4 and 9:

❑ program_window start address = 0x4
❑ program_window end address = 0x12

A data breakpoint is not detected at line 12 because the instruction at program_window end is outside of the program window. Refer to *Instruction pipelining*, page 11-11, for information on detecting breakpoints at the six pipeline phases.

11.5 Running Your Program

Once you have defined your parameters, the analysis module can begin collecting data as soon as you run your program. It will stop collecting data when the defined conditions are met. The analysis module monitors the progress of the defined events while your program is running.

> **Note:**
>
> The conditions for the analysis session must be defined *before* your analysis session begins; you cannot change conditions *during* execution of your program.

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

- Click the Run icon on the toolbar:
- From the Debug menu, select Run.
- Press F5.
- From the command line, enter the RUN command. The format for this command is:

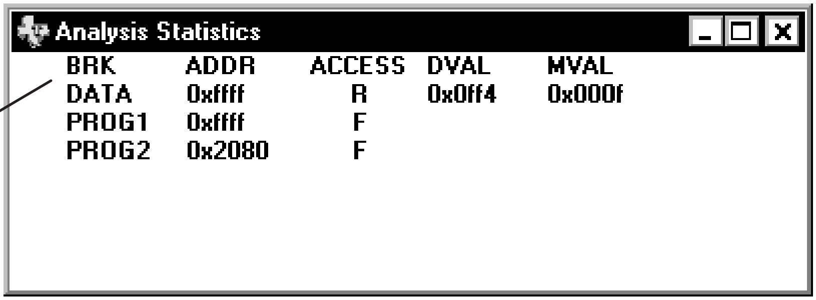
  run [*expression*]

You can use any of the debugger run commands (STEP, CSTEP, NEXT, etc.) described in Chapter 6 except the RUNB (run benchmark) or RUNF (run free) commands.

11.6 Viewing the Analysis Data

You can monitor the status of the analysis module by checking the Analysis Statistic window. This window displays an ongoing progress report of the analysis module's activity. Through this window, you can monitor the status of the break events. Figure 11–4 illustrates the Analysis Statistics window.

Figure 11–4. Global Analysis Breakpoint Checking Currently Disabled

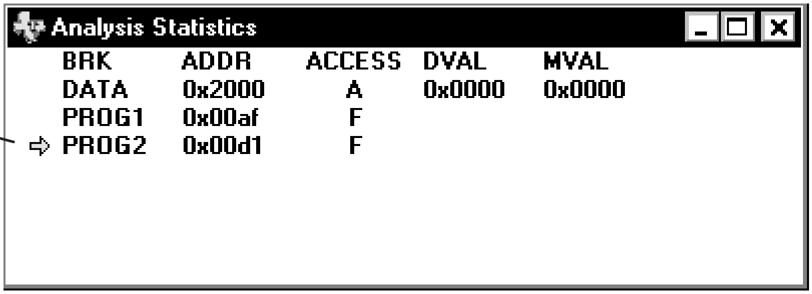


```
BRK      ADDR     ACCESS  DVAL     MVAL
DATA     0xffff     R     0x0ff4   0x000f
PROG1    0xffff     F
PROG2    0x2080     F
```

Available hardware

The BRK column displays the hardware breaks available: the data bus, program bus 1, and program bus 2. The ADDR column lists the address for entry of the BRK column. The ACCESS column 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 11–5. Analysis Interface View Window, Displaying an Ongoing Status Report



```
   BRK      ADDR     ACCESS  DVAL     MVAL
   DATA     0x2000     A     0x0000   0x0000
   PROG1    0x00af     F
⇨  PROG2    0x00d1     F
```

Arrow points to the active line.

You can double-click on the PROG1 or PROG2 row during execution. Then the file and disassembly window displays are updated to display the source and disassembly for that address.

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.

Chapter 12

Using the Parallel Debug Manager

The TMS320C54x emulation system is a true multiprocessor debugging system. It allows you to debug your entire application by using the parallel debug manager (PDM). The PDM is a command shell that controls and coordinates multiple debuggers. This chapter describes the functions that you can perform with the PDM.

See Chapter 2, *Getting Started With the Debugger,* for information about invoking the PDM and debuggers.

Topic **Page**

12.1 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 validates 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 appears 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 'C54x processors CPU_B, you would use the following command to invoke the debugger:

```
spawn emu54x –n CPU_B ↵
```

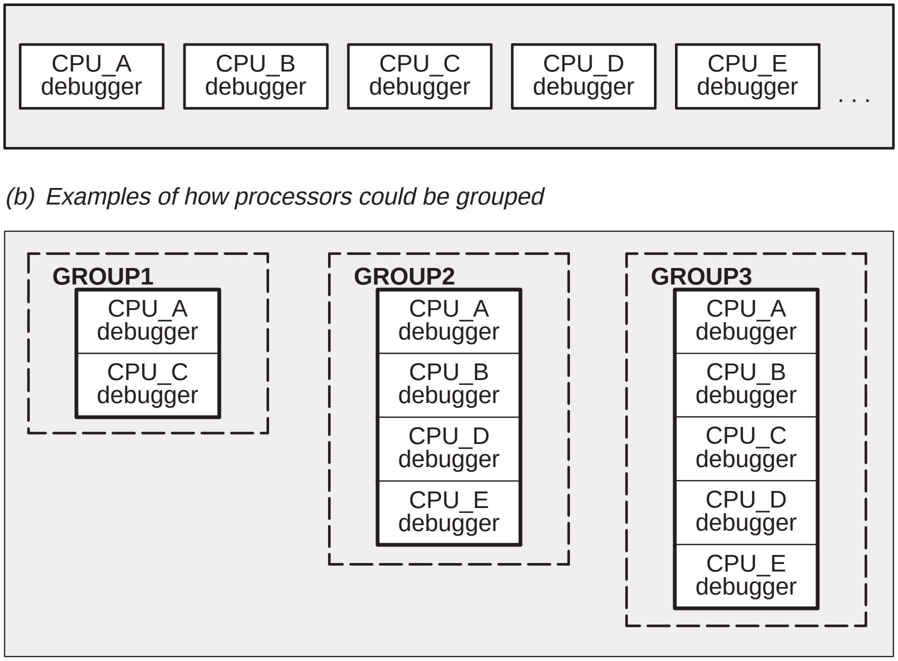
From this point on, whenever you needed to identify this debugger, you could identify it by its processor name, CPU_B.

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

Organizing processors into groups

You can organize processors into groups by using the SET command to group processors under one name. Each processor can belong to any group, all groups, or a group of its own. Figure 12–1 *(a)* shows an example of processors in a system, and Figure 12–1 *(b)* illustrates three examples of named groups. GROUP1 contains two processors, GROUP2 contains four processors, and GROUP3 contains five processors.

Figure 12–1. 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 12–1 *(b)*, you could enter the following on the PDM command line:

  ```
  set GROUP1 = CPU_A CPU_C ↵
  ```

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

- **Setting the default group**

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

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

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

  The PDM automatically sends commands to the default group when you do not specify a group name.

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

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

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

  ```
  set GROUPA = $GROUPA CPU_E ↵
  ```

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

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

```
set dgroup = $GROUPA ↵
```

❑ **Listing all groups of processors**

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

```
set ↵
```

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

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

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

```
set dgroup ↵
dgroup   ”CPU_C CPU_D CPU_E”
```

❑ **Deleting a group**

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

unset *group name*

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

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

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

```
unset * ↵
```

The asterisk *does not* work as a wild card.

Note:

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

12.2 Sending Debugger Commands to One or More Debuggers

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

The syntax for the SEND command is:

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

- The **–g** option specifies the group or processor that the debugger command should be sent to. If you do not use this option, the command is sent to the default group (dgroup).
- The **–r** (return) option determines when control returns to the PDM command line:
  - **Without –r**, control is not returned to the command line until each debugger in the group finishes running code. Any results that are printed in the COMMAND window of the individual debuggers are also echoed in the PDM command window. These results are 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 returns control to the PDM command line. However, no debugger executing the command is interrupted.
  - **With –r**, control is returned to the command line immediately, even if a debugger is still executing a command. When you use –r, you *do not* see the results of the commands that the debuggers are executing.

    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 to which QUIT was sent stop; 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 ↵
```

12.3 Running and Halting Code

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

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

The formats for these commands are:

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

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

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

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

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

Sending ESCAPE to all processors

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

The syntax for this command is:

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

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

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

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

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

```
stat –g GROUP_A ↵
```

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

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

12.4 Entering PDM Commands

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

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

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

Executing PDM commands from a batch file

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

take *batch filename*

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

The TAKE command is similar to the debugger version of this command (described on page 13-61). 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 12-10) 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 13-20.

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

  dlog *filename*

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

- To end the recording session, enter:

  dlog close ↵

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

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

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

- **Appending to an existing file.** Use the **a** parameter to open an existing file and append the information in the display area.
- **Writing over an existing file.** Use the **w** parameter to open an existing file and write over the current contents of the file. This is the default action if you specify an existing filename without using either the **a** or **w** options; you will lose the contents of an existing file if you do not use the append (a) option.

Controlling PDM command execution

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

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

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

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

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

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

  The PDM version of the LOOP command is different from the debugger version of this command (described on page 13-31). 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 12.8, beginning on page 12-18):

```
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 12-10) 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 12.8):

```
echo $var_proc1 ↵
34
```

The PDM version of the ECHO command works in exactly the same way as the debugger version described on page 13-21 works, except that you can use the PDM version outside of a batch file. (The debugger does not work with system variables).

Pausing command execution

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

The syntax for the PAUSE command is:

pause

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

```
<< pause – type return >>
```

To continue processing, press ⏎.

Using the command history

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

The PDM command history allows you to reenter 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 reenter. 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 reentering.

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

!*string*

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

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

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

  history ↵

The command history for the PDM works differently from that of the debugger; the TAB and F2 keys have no command-history meaning for the PDM.

12.5 Defining Your Own Command Strings with PDM

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

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

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

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

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

For more information about these features, see section 3.1, *Defining Your Own Command Strings*.

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

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

  ```
  alias ml, ”send –g %1 ml” ↵
  ```

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

```
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 do not want the PDM to evaluate (use the processors associated with the group name as the prompt)—you just want the group name. As a result, you do not need to use the dollar sign when you want to use only the group name.

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

  ```
  switch GROUP3 ↵
  ```

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

12.6 Entering Operating-System Commands

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

system *operating-system command*

The SYSTEM command is similar to the debugger’s SYSTEM command (described on page 3-5), but there are some differences.

- **Similarities.** You can enter operating-system commands without having to leave the primary environment (in this case, the PDM) and without having to open another operating-system window.

- **Differences.** Unlike the debugger version of the SYSTEM command:

  - The PDM version of the SYSTEM command cannot be entered without an operating-system command parameter. Therefore, you cannot use the command to open a shell.

  - There is no flag parameter; command output is always displayed in the PDM window.

12.7 Understanding the PDM's Expression Analysis

The PDM analyzes expressions differently than individual debuggers do (expression analysis for the debugger is described in Chapter 14, *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 12-19 and 12-10, respectively).

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

Table 12–1 summarizes the PDM operators. The PDM interprets the operators in the order in which they are listed in Table 12–1 (left to right, top to bottom). In other words, ( ) is first, * is second, / is third, and so forth.

Table 12–1. 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

12.8 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
```

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 12-8) 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 12-4), use the UNSET command with an asterisk instead of a variable name:

```
unset * ⏎
```

Note:

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

12.9 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 do not use this option, the command is sent to the default group (dgroup).
- When you send the EVAL command to more than one processor, the PDM takes the *variable name* that you supply and appends a suffix for each processor. The suffix consists of the underscore character (_) followed by the name that you assigned the processor. That way, you can differentiate between the resulting variables.
- The *expression* can be any expression that uses the symbols described in section 12.7.
- When you use the optional *format* parameter, the value that the variable is set to will be in one of the following formats:

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

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

```
set dgroup = CPU_A CPU_B ↵
eval val = j + k ↵
set ↵
dgroup    ”CPU_A CPU_B”
val_CPU_A ”22”
val_CPU_B ”34”
```

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

Chapter 13

Summary of Commands

This chapter describes the basic debugger and PDM commands and profiling commands.

Topic **Page**

13.1 Functional Summary of Debugger Commands

This section summarizes the debugger commands according to these categories:

❑ **Managing multiple debuggers.** These commands allow you to group debuggers, run code on multiple processors, and send commands to a group of debuggers.

❑ **Changing modes.** These commands (listed on page 13-4) allow you to switch freely between the debugging modes (auto, mixed, and assembly).

❑ **Managing windows.** These commands (listed on page 13-4) allow you to make a window active and move or resize the active window.

❑ **Displaying and changing data.** These commands (listed on page 13-5) allow you to display and evaluate a variety of data items.

❑ **Performing system tasks.** These commands (listed on page 13-6) allow you to perform several system functions and provide you with some control over the target system.

❑ **Managing breakpoints.** These commands (listed on page 13-7) provide you with a command line method for controlling software breakpoints.

❑ **Displaying files and loading programs.** These commands (listed on page 13-4) allow you to change the displays in the File and Disassembly windows and to load object files into memory.

❑ **Customizing the screen.** These commands (listed on page 13-4) allow you to customize the debugger display, then save and later reuse the customized displays.

❑ **Memory mapping.** These commands (listed on page 13-7) allow you to define the areas of target memory that the debugger can access.

❑ **Running programs.** These commands (listed on page 13-8) provide you with a variety of methods for running your programs in the debugger environment.

❑ **Profiling commands.** These commands (listed on page 13-9) allow you to collect execution statistics for your code.

Managing multiple debuggers

To do this...	Use this command...	See page...
Use the command history	!	13-11
Assign a variable to the result of an expression	@	13-12
Define a custom command string	alias	13-13
Record the information shown in PDM display area	dlog	13-20
Display a string to the PDM display area	echo	13-21
Evaluate an expression in a debugger or group of debuggers and set a variable to the result	eval	13-22
List available PDM commands	help	13-27
View the description of a PDM command	help	13-27
List the last 20 commands	history	13-28
Conditionally execute PDM commands	if/elif/else/endif	13-28
Loop through PDM commands	loop/break/ continue/ endloop	13-30
Pause the PDM	pause	13-42
Halt code execution	pesc	13-42
Global halt	phalt	13-43
Run code globally	prun	13-46
Run free globally	prunf	13-47
Single-step globally	pstep	13-47
Exit any debugger and/or the PDM	quit	13-48
Send a command to an individual processor or a group of processors	send	13-53
Change the PDM prompt	set	13-54
Create your own system variables	set	13-54
Define or modify a group of processors	set	13-54
List all system variables or groups of processors	set	13-54
Set the default group	set	13-54
Invoke an individual debugger	spawn	13-57
Find the execution status of a processor or a group of processors	stat	13-59
Enter an operating-system command	system	13-60
Execute a batch file	take	13-61
Delete an alias definition	unalias	13-62
Delete a group or system variable	unset	13-62

Changing modes

To put the debugger in...	Use this command...	See page...
Assembly mode	asm	13-14
Auto mode for debugging C code	c	13-15
Mixed mode	mix	13-38

Managing windows

To do this...	Use this command...	See page...
Reposition a window	move	13-39
Resize a window	size	13-56
Make a window active	win	13-66
Make a window as large as possible	zoom	13-67

Customizing the screen

To do this...	Use this command...	See page...
Change the command-line prompt	prompt	13-46
Load and use a previously saved custom screen configuration	sconfig	13-52
Save a custom screen configuration	ssave	13-58

Displaying files and loading programs

To do this...	Use this command...	See page...
Display a text file in a File window	file	13-25
Load an object file and its symbol table	load	13-30
Modify disassembly with the patch assembler	patch	13-41
Load only the object-code portion of an object file	reload	13-48
Load only the symbol-table portion of an object file	sload	13-57

Displaying and changing data

To do this...	Use this command...	See page...
Evaluate and display the result of a C expression	?	13-10
Display C and/or assembly language code at a specific point	addr	13-12
Display the Calls window	calls	13-16
Display assembly language code at a specific address	dasm	13-18
Display the values in an array or structure, or display the value that a pointer is pointing to	disp	13-18
Evaluate a C expression without displaying the results	eval	13-22
Display a specific line in the File window	line	13-30
Display a specific C function	func	13-26
Change the range of memory displayed in the Memory window or display an additional Memory window	mem	13-37
Change the format for displaying data values	setf	13-55
Display the current debugger version	version	13-64
Continuously display the value of a variable, register, or memory location within the Watch window	wa	13-64
Delete a data item from the Watch window	wd	13-65
Show the type of a data item	whatis	13-66
Delete all data items from the Watch window	wr	13-67

Performing system tasks

To do this...	Use this command...	See page...
Define your own command string	alias	13-13
Change the current working directory from within the debugger environment	cd, chdir	13-16
Clear all displayed information from the display area of the Command window	cls	13-16
List the contents of the current directory or any other directory	dir	13-18
Record the information shown in the display area of the Command window	dlog	13-20
Display a string to the Command window while executing a batch file	echo	13-21
Display a help topic for a debugger command	help	13-27
Conditionally execute debugger commands in a batch file	if/else/endif	13-29
Loop debugger commands in a batch file	loop/endloop	13-31
Pause the execution of a batch file	pause	13-42
Exit the debugger	quit	13-48
Reset communication with the emulator	reconnect	13-48
Reset the target system	reset	13-49
Associate a beeping sound with the display of error messages	sound	13-57
Enter any operating-system command or exit to a system shell	system	13-60
Execute commands from a batch file	take	13-61
Delete an alias definition	unalias	13-62
Name additional directories that can be searched when you load source files	use	13-63

Managing breakpoints

To do this...	Use this command...	See page...
Add a software breakpoint	ba	13-14
Delete a software breakpoint	bd	13-14
Display a list of all the software breakpoints that are set	bl	13-15
Reset (delete) all software breakpoints	br	13-15

Memory mapping

To do this...	Use this command...	See page...
Initialize a block of memory word by word	fill	13-25
Initialize a block of memory byte by byte	fillb	13-26
Add an address range to the memory map	ma	13-32
Enable or disable memory mapping	map	13-33
Connect a simulated I/O port to an input or output file (simulator only)	mc	13-33
Delete an address range from the memory map	md	13-36
Disconnect a simulated I/O port (simulator only)	mi	13-38
Display a list of the current memory map settings	ml	13-39
Reset the memory map (delete all range definitions)	mr	13-39
Save a block of memory to a system file	ms	13-40
Connect an input file to the pin (simulator only)	pinc	13-43
Disconnect the input file from the pin (simulator only)	pind	13-44
List the pins that are connected to the input files (simulator only)	pinl	13-44

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	13-17
Single-step through assembly language or C code, one C statement at a time	cstep	13-17
Run a program up to a certain point	go	13-27
Halt the target system	halt	13-27
Single-step through assembly language or C code; step over function calls	next	13-41
Reset the target system	reset	13-49
Reset the program to its entry point	restart	13-49
Execute code in a function and return to the function's caller	return	13-49
Run a program	run	13-50
Run a program with benchmarking—count the number of CPU clock cycles consumed by the executing portion of code	runb	13-51
Disconnect the emulator from the target system and run free	runf	13-51
Single-step through assembly language or C code	step	13-59
Execute commands from a batch file	take	13-61

Profiling commands

All of the profiling commands can be entered from the Tools→Profile menu and associated dialog boxes. In many cases, using the Tools→Profile menu and dialog boxes is the easiest way to enter some of these commands. For this reason and also because there are over 100 profiling commands, most of these commands are not described individually in this chapter (as the basic debugger commands are).

Listed below are some of the profiling commands that you might choose to enter from the command line; these commands are also described in the alphabetical command summary. The remaining profiling commands are summarized in section 13.3, *Summary of Profiling Commands*, on page 13-68.

To do this...	Use this command...	See page...
Run a full profiling session	pf	13-42
Run a quick profiling session	pq	13-44
Resume a profiling session	pr	13-45
Switch to profiling environment	profile	13-45
Add a stopping point	sa	13-51
Delete a stopping point	sd	13-52
List all the stopping points	sl	13-56
Delete all the stopping points	sr	13-58
Save all the profile data to a file	vaa	13-63
Save currently displayed profile data to a file	vac	13-63
Reset the display in the Profile window to show all areas and the default set of data	vr	13-64

13.2 Alphabetical Summary of Debugger and PDM Commands

There are three types of debugger commands:

- Basic debugger commands
- Parallel Debug Manager (PDM) commands that allow you to control multiple debuggers
- Profiler commands that allow you to control the debugger profiling environment

Most of the commands can be used in the basic debugger environment and/or the profiling environment. Other commands can be used only by the parallel debug manager (PDM). Some commands can be used in more than one environment; other commands can be used in only one of the environments. Each command description identifies the applicable environments for the command.

Commands are not case sensitive; to emphasize this, command names are shown in both uppercase and lowercase throughout this book.

? *Evaluate Expression*

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

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The ? (evaluate expression) command evaluates an expression and shows the result in the display area of the Command window. The *expression* can be any C expression, including an expression with side effects; however, you cannot use a string constant or function call in the *expression.* If the *expression* identifies an address, you can follow it with @prog to identify program memory or @data to identify data memory. Without the suffix, the debugger treats an address expression as a program-memory location.

If the result of *expression* is not an array or structure, then the debugger displays the results in the Command window. If *expression* is a structure or array, ? displays the entire contents of the structure or array; you can halt long listings by pressing ESC .

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

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

! *Use the PDM Command History*

Syntax **!**{*prompt number* **|** *string*}
!!

Menu selection none

Toolbar selection none

Environments □ basic debugger ☑ PDM □ profiling

Description The PDM supports a command history that is similar to the UNIX command history. The PDM prompt identifies the number of the current command. This number is incremented with every command. The PDM command history allows you to reenter any of the last twenty commands.

- ❑ The *number* parameter is the number of the PDM prompt that contains the command that you want to reenter.
- ❑ The *string* parameter tells the PDM to execute the last command that began with *string.*
- ❑ The !! command tells the PDM to execute the last command that you entered.

@ — *Substitute Result of an Expression*

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

Menu selection none

Toolbar selection none

Environments □ basic debugger ☑ PDM □ profiling

Description Unlike the SET command, the @ command first evaluates the *expression*, and then sets the *variable name* to the result. The *expression* can be any expression that uses the symbols described in section 12.7, *Understanding the PDM's Expression Analysis*, on page 12-17. The *variable name* can consist of up to 128 alphanumeric characters or underscore characters.

addr — *Display Code at Specified Address*

Syntax **addr** {*addresss*[**@prog** | **@data**] | *function name*}

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM □ profiling

Description Use the ADDR command to display C code or the disassembly at a specific point. ADDR's behavior changes depending on the current debugging mode:

- In assembly mode, ADDR works like the DASM command, positioning the code starting at *address* or at *function name* as the first line of code in the Disassembly window.

- In a C display, ADDR works like the FUNC command, displaying the code starting at *address* or at *function name* as the first line of code in the File window.

- In mixed mode, ADDR affects both the Disassembly and File windows by displaying code starting at *address* or at *function name* as the first line of code in the Disassembly and File window.

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 Configure→Alias Commands

Toolbar selection none

Environments ☑ basic debugger ☑ PDM ☑ profiling

Description You can use the ALIAS command to associate one or more debugger or PDM commands with a single *alias name.*

- The debugger version of the ALIAS command allows you to associate one or more debugger commands with a single *alias name.*
- The PDM version of the ALIAS command allows you to associate one or more PDM commands with a single alias name *or* associate one or more debugger commands with a single alias name.

You can include as many commands in the *command string* as you like, as long you separate them with semicolons and enclose the entire string of commands in quotation marks. Also, you can identify command parameters by a percent sign followed by a number (%1, %2, etc.). The total number of characters for an individual command (expanded to include parameter values) is limited to 132. (This restriction applies to the debugger version of the ALIAS command only.)

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

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

asm — *Enter Assembly Mode*

Syntax **asm**

Menu selection View→Assembly

Toolbar selection none

Environments ☑ basic debugger □ PDM □ profiling

Description The ASM command changes from the current debugging mode to assembly mode. If you are already in assembly mode, the ASM command has no effect.

ba — *Add Software Breakpoint*

Syntax **ba** *address*

Menu selection Configure→Breakpoints

Toolbar selection ●

Environments ☑ basic debugger □ PDM ☑ profiling

Description The BA command sets a software breakpoint at a specific *address*. The *address* can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label.

You can set breakpoints in program memory (RAM) only; the *address* parameter is treated as a program-memory address.

bd — *Delete Software Breakpoint*

Syntax **bd** *address*

Menu selection Configure→Breakpoints

Toolbar selection ●

Environments ☑ basic debugger □ PDM ☑ profiling

Description The BD command clears a software breakpoint at a specific *address*. The *address* can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label. The *address* is treated as a program-memory address.

bl — *List Software Breakpoints*

Syntax **bl**

Menu selection Configure→Breakpoints

Toolbar selection

Environments ☑ basic debugger □ PDM ☑ profiling

Description The BL command lists all the software breakpoints that are currently set in your program. It displays a table of breakpoints in the display area of the Command window. BL lists all the breakpoints that are set in the order in which you set them.

br — *Reset Software Breakpoint*

Syntax **br**

Menu selection Configure→Breakpoints

Toolbar selection

Environments ☑ basic debugger □ PDM ☑ profiling

Description The BR command clears all software breakpoints that are set.

c — *Enter Auto Mode*

Syntax **c**

Menu selection View→C (Auto)

Toolbar selection none

Environments ☑ basic debugger □ PDM □ profiling

Description The C command changes from the current debugging mode to auto mode. If you are already in auto mode, the C command has no effect.

calls

Opens Calls Window

Syntax **calls**

Menu selection View→Call Stack Window

Toolbar selection none

Environments ☑ basic debugger ☐ PDM ☐ profiling

Description The CALLS command displays the Calls window. The debugger displays this window automatically when you are in auto/C or mixed mode. However, you can close the Calls window; the CALLS command opens the window again.

cd, chdir

Change Directory

Syntax **cd** [*directory name*]
chdir [*directory name*]

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☐ PDM ☑ profiling

Description The CD or CHDIR command changes the current working directory from within the debugger. You can use relative pathnames as part of the *directory name*. If you do not use a *directory name*, the CD command displays the name of the current directory. You can also use the CD command to change the current drive. For example,

```
cd c:
cd d:\csource
cd c:\asmsrc
```

cls

Clear Screen

Syntax **cls**

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☐ PDM ☑ profiling

Description The CLS command clears all displayed information from the display area of the Command window.

cnext — *Single-Step C, Next Statement*

Syntax

cnext [*expression*]

Menu selection

Debug→Next C

Toolbar selection

Environments

☑ basic debugger □ PDM □ profiling

Description

The CNEXT command is similar to the CSTEP command. It runs a program one C statement at a time, updating the display after executing each statement. If you are using CNEXT to step through assembly language code, the debugger does not update the display until it has executed all assembly language statements associated with a single C statement. Unlike CSTEP, CNEXT steps over function calls rather than stepping into them—you do not see the single-step execution of the function call.

The *expression* parameter specifies the number of statements that you want to single-step. You can use a conditional *expression* for conditional single-step execution. (Section 7.4, *Running Code Conditionally*, page 7-11, discusses this in detail.)

cstep — *Single-Step C*

Syntax

cstep [*expression*]

Menu selection

Debug→Step C

Toolbar selection

Environments

☑ basic debugger □ PDM □ profiling

Description

The CSTEP command single-steps through a program one C statement at a time, updating the display after executing each statement. If you are using CSTEP to step through assembly language code, the debugger does not update the display until it has executed all assembly language statements associated with a single C statement.

The *expression* parameter specifies the number of statements that you want to single-step. You can use a conditional *expression* for conditional single-step execution. (Section 7.4, *Running Code Conditionally*, page 7-11, discusses this in detail.)

dasm — *Display Disassembly at Specific Address*

Syntax **dasm** {*address*[**@prog** | **@data**] **|** *function name*}

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The DASM command displays code beginning at a specific point within the Disassembly window. You can follow the *address* with @prog to identify program memory or @data to identify data memory. Without the suffix, the debugger treats an address as a program-memory location.

dir — *List Directory Contents*

Syntax **dir** [*directory name*]

Menu selection none

Toolbar 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 do not use the parameter, the debugger lists the contents of the current directory.

disp — *Add Structure, Array, or Pointer to Watch Window*

Syntax **disp** *expression* [**,** *display format*] [**, @data** | **@prog** | **@io**]

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM □ profiling

Description The DISP command opens a Watch window to display the contents of one of the following:

- An array
- A structure
- Pointer expressions to a scalar type (of the form **pointer*)

If the *expression* is not one of these types, then DISP acts like a ? command. 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.

When the Watch window is open, you can display the data pointed to by a pointer or display the members of the array or structure by clicking the box icon next to watched item:

⊞

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

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

You can use the *display format* parameter only when you are displaying a scalar type, an array of scalar type, or an individual member of an aggregate type.

You can also use the DISP command with a typecast expression to display memory contents in any format. Here are some examples:

```
disp *0
disp *(float *)123
disp *(char *)0x111
```

This shows memory in the Watch window as an array of locations; the location that you specify with the *expression* parameter is member [0], and all other locations are offset from that location.

divide

Specify the Clock Divide Ration for a Device

Syntax **divide** *r*

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ profiling

Description The DIVIDE command is used to specify the clock divide ration for the device.

The *r* parameter is a real number or integer specifying the ratio of serial prot clock versus the CPU clock. Use the divide ration when the serial port is configured to use the external clock. when you use the DIVIDE command, it must be the first command in the file.

dlog — *Record Display Area*

Syntax

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

Menu selection

File→Open→Log File
or
File→Close→Log File

Toolbar selection

none

Environments

☑ basic debugger ☑ PDM ☑ profiling

Description

The DLOG command allows you to record the information displayed in the Command window or in the PDM display area into a log file and to record all commands that you enter from the command line, from the toolbar, from the menus, or with function keys.

To begin a recording session in the display area of the PDM, use:

dlog *filename*

To end the recording session, enter:

dlog close ↵

You can write over existing log files or append additional information to existing files. The optional parameters of the DLOG command control how existing log files are used:

- **Appending to an existing file.** Use the **a** parameter to open an existing file and append the information in the display area to the information already in the file.

- **Writing over an existing file.** Use the **w** parameter to open an existing file and write over the current contents of the file. This is the default action if you specify an existing filename without using either the **a** or **w** options; you will lose the contents of an existing file if you do not use the append (a) option.

echo

Echo String to Display Area **Batch File Only**

Syntax **echo** *string*

Menu selection none

Toolbar 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 cannot use quote 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 **Batch File Only**

Description ELIF provides an alternative test by which you can execute PDM commands in the IF/ELIF/ELSE/ENDIF command sequence. See page 13-28 for more information about these commands.

else

Execute Alternative Commands **Batch File Only**

Description ELSE provides an alternative list of debugger or PDM commands in the IF/ELSE/ENDIF or IF/ELIF/ELSE/ENDIF command sequences, respectively. See pages 13-28 and 13-29 for more information about these commands.

endif

Terminate Conditional Sequence **Batch File Only**

Description ENDIF identifies the end of a conditional-execution command sequence begun with an IF command. See pages 13-28 and 13-29 for more information about these commands.

endloop

Terminate Looping Sequence **Batch File Only**

Description ENDLOOP identifies the end of the LOOP/ENDLOOP command sequence. See pages 13-30 and 13-31 for more information about the LOOP/ENDLOOP commands.

eval — *Evaluate Expression*

Syntax

eval *expression* [**@data** | **@prog** | **@io**]
e *expression* [**@data** | **@prog** | **@io**]

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description

The EVAL command evaluates an expression like the ? command does *but does not show the result* in the display area of the Command window. EVAL is useful for assigning values to registers or memory locations in a batch file (where it is 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* **PDM Environment**

Syntax

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

Menu selection none

Toolbar 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 do not use this option, the command is sent to the default group (dgroup).
- When you send the EVAL command to more than one processor, the PDM takes the *variable name* that you supply and appends a suffix for each processor. The suffix consists of the underscore character (_) followed by the name that you assigned the processor.
- The *expression* can be any expression that uses the symbols described in section 12.7, *Understanding the PDM’s Expression Analysis*, on page 12-17.

❑ 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

ext_addr — *Enable or Disable Extended Addressing*

Syntax **ext_addr** {**on** | **off**}

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ profiling

Description The EXT_ADDR command enables or disables extended addressing. If your code depends on target system extended memory, you must enable extended addressing before you load your code.

You cannot use the EXT_ADDR command before you define your extended memory configuration.

ext_addr_def — *Define Extended Addressing* **Emulator Only**

Syntax **ext_adr_def** *map start* [{**@prog**|**@data**}], *reg addr* [{**@prog**|**@data**|**@io**}], *mask*

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ profiling

Description EXT_ADDR_DEF describes your extended memory configuration to the debugger and tells the debugger where to look for the mapper register (XPC).

You must define your extended memory configuration before you can use the EXT_ADDR command to enable extended addressing.

- ❑ The *map start* parameter defines the beginning of the mapped memory range. By default, the *map start* parameter is treated as a program-memory address.

  The map start address is determined by the OVLY bit:

  - ■ If OVLY is 1, map start is 0x8000@prog.
  - ■ If OVLY is 0, map start is 0x0000@prog.

  You can follow the *map start* parameter with @prog to identify program memory, or with @data to identify data memory.

- ❑ The *reg addr* parameter defines the location of the mapping register (XPC). Use the address 0x1E@data.

  By default, the *reg addr* parameter is treated as a data-memory address. However, you can follow the *reg addr* parameter with @data to identify data memory, @prog to identify program memory, or @io to identify I/O space.

- ❑ The *mask* parameter must be 0x7F since the program address bus (PAB) is 23 bits wide.

ext_addr_def — *Define Extended Addressing* — **Simulator Only**

Syntax **ext_adr_def** *mapped mem start* [**@page**], *mr addr* [**@page**], *mapper mask*

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ profiling

Description EXT_ADDR_DEF describes your extended memory configuration to the debugger and tells the debugger where to look for the mapper register (XPC).

You must define your extended memory configuration before you can use the EXT_ADDR command to enable extended addressing.

- ❑ The *mapped mem start* parameter defines the beginning of the mapped memory range. By default, the *mapped mem start* parameter is treated as a program-memory address.

  The map start address is determined by the OVLY bit:

  - If OVLY is 1, map start is 0x8000@prog.
  - If OVLY is 0, map start is 0x0000@prog.

  You can follow the *mapped mem start* parameter with @prog to identify program memory, or with @data to identify data memory.

- The *mr addr* parameter defines the location of the mapping register (XPC). Use the address 0x1E@data.

  By default, the *mr addr* parameter is treated as a data-memory address. However, you can follow the *mr addr* parameter with @data to identify data memory, @prog to identify program memory, or @io to identify I/O space.

- The *mapper mask* parameter must be 0x7F since the program address bus (PAB) is 23 bits wide.

file

Display Text File

Syntax **file** *filename*

Menu selection File→Open

Toolbar selection 📂

Environments ☑ basic debugger □ PDM ☑ profiling

Description The FILE command displays the contents of any text file in the File window. This command is intended primarily for displaying C source code. You can view as multiple text files at the same time using multiple File windows.

fill

Fill Memory Word by Word

Syntax **fill** *address*, *page*, *length*, *data*

Menu selection Configure→Memory Fill

Toolbar selection none

Environments ☑ basic debugger □ PDM □ profiling

Description The FILL command fills a block of memory word by word with a specified value.

- The *address* parameter identifies the first address in the block.

- ❑ The *page* parameter is a 1-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 number of words to fill.
- ❑ The *data* parameter is the value that is placed in each word in the block.

fillb — *Fill Memory Byte by Byte*

Syntax **fillb** *address*, *length*, *data*

Menu selection Configure→Memory Fill

Toolbar selection none

Environments ☑ basic debugger □ PDM □ profiling

Description The FILLB command fills a block of memory byte by byte with a specified value.

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

func — *Display Function*

Syntax **func** {*function name* | *address*}

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The FUNC command displays a specified C function in the File window. You can identify the function by its name or by an address in the function; an *address* parameter is treated as a program-memory address. FUNC works the same way FILE works, but with FUNC you do not need to identify the name of the file that contains the function.

go — *Run to Specified Address*

Syntax **go** [*address*]

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☐ PDM ☐ profiling

Description The GO command executes code up to a specific point in your program. The *address* parameter is treated as a program-memory address. If you do not supply an *address*, then GO acts like a RUN command without an *expression* parameter.

halt — *Halt Target System*

Syntax **halt**

Menu selection Debug→Halt!

Toolbar selection

Environments ☑ basic debugger ☐ PDM ☐ profiling

Description The HALT command halts your program, if you are using a simulator, or halts the target system after you have entered a RUNF command, if you are using an emulator. When you invoke the debugger, it automatically executes a HALT command. If you enter a RUNF, quit the debugger, and later reinvoke the debugger, you will effectively reconnect the emulator to the target system and run the debugger in its normal mode of operation.

help — *Display Help Topic for Debugger Command*

Syntax **help** [*debugger command*]

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☐ PDM ☑ profiling

Description The HELP command opens a help topic that describes the *debugger command*. If you omit the *debugger command*, the debugger displays a list of help topics.

help — *List PDM Commands* — PDM Environment

Syntax **help** [*command*]

Menu selection none

Toolbar 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 20 PDM Commands*

Syntax **history**

Menu selection none

Toolbar selection none

Environments □ basic debugger ☑ PDM □ profiling

Description The HISTORY command displays the last 20 PDM commands that you have entered.

if/elif/else/endif — *Conditionally Execute PDM Commands* — PDM Environment

Syntax

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

Menu selection none

Toolbar selection none

Environments □ basic debugger ☑ PDM □ profiling

Description These commands allow you to execute PDM commands conditionally in a batch file or from the command line.

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

The IF/ELIF/ELSE/ENDIF can be entered interactively or included in a batch file that is executed by the TAKE command. When you enter IF from the PDM command line, a question mark (?) prompts you for the next entry. The PDM continues to prompt you for input using the ? until you enter ENDIF. After you enter ENDIF, the PDM immediately executes the IF command.

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

if/else/endif

Conditionally Execute Debugger Commands **Batch File Only**

Syntax

if *expression*
debugger commands
[**else**
debugger commands]
endif

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description

These commands allow you to execute debugger commands conditionally in a batch file. If the *expression* if nonzero, the debugger executes the commands between the IF and the ELSE or ENDIF. The ELSE portion of the command sequence is optional.

You can substitute a keyword for the expression. Keywords evaluate to true (1) or false (0). You can use the following keywords with the IF command:

- **$$EMU$$** (tests for the emulator version of the debugger)
- **$$SIM$$** (tests for the simulator version of the debugger)

The conditional commands work with the following provisions:

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

line — *Display the Specified Line Number in the FILE Window*

Syntax **line** *line number*

Menu selection none

Environments ☑ basic debugger □ PDM □ profiling

Description Use the LINE command to view specific lines of code. The LINE command displays the specified *line number* in the middle of the FILE window. When the *line number* is already displayed in the FILE window, the LINE command does not affect the display.

load — *Load Executable Object File*

Syntax **load** *object filename*

Menu selection File→Load→Load Program

Toolbar selection none

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 do not supply an extension, the debugger looks for *filename*.out. The LOAD command clears the old symbol table and closes any Watch windows.

loop/break/continue/endloop — *Loop Through PDM Commands* **PDM Environment**

Syntax

loop *Boolean 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 BREAK, CONTINUE, or ENDLOOP.
- ❑ If the Boolean expression evaluates to false (0), the loop is not entered.

The optional BREAK command allows you to exit the loop without having to reach the ENDLOOP. This is helpful when you are testing a group of processors and want to exit if an error is detected.

The CONTINUE command, which is also optional, acts as a goto and returns command flow to the enclosing LOOP command. CONTINUE is useful when the part of the loop that follows is complicated; returning to the top of the loop avoids further nesting.

The LOOP/BREAK/CONTINUE/ENDLOOP commands can be entered interactively or included in a batch file that is executed by the TAKE command. When you enter LOOP from the PDM command line, a question mark (?) prompts you for the next entry. The PDM continues to prompt you for input using the ? until you enter ENDLOOP. After you enter ENDLOOP, the PDM immediately executes the LOOP command.

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

loop/endloop *Loop Through Debugger Commands* **Batch File Only**

Syntax

loop *expression*
debugger commands
endloop

Menu selection none

Toolbar 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 the same method as in the run conditional command expression:

- ❑ If you use an *expression* that is not Boolean, the debugger evaluates the expression as a loop count.
- ❑ If you use a Boolean *expression*, the debugger executes the command repeatedly as long as the expression is true.

The LOOP/ENDLOOP commands work under the following conditions:

- You can use LOOP/ENDLOOP commands only in a batch file.
- You must enter each debugger command on a separate line in the file.
- You cannot nest LOOP/ENDLOOP commands within the same file.

ma — *Add Block to Memory Map*

Syntax **ma** *address*, *page*, *length*, *type*

Menu selection Configure→Memory Maps

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The MA command identifies valid ranges of target memory. A new memory range must not overlap an existing entry; if you define a range that overlaps an existing range, the debugger ignores the new range.

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

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

  ```
  Conflicting map range
  ```

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

map — *Enable/Disable Memory Mapping*

Syntax **map** {**on** | **off**}

Menu selection Configure→Memory Maps

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The MAP command enables or disables memory mapping. Disabling memory mapping can cause bus fault problems in the target because the debugger may attempt to access nonexistent memory.

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

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

mc — *Connect Simulated I/O Port to a File* **Simulator Only**

Syntax **mc** *port address*, *page* **,** *length*, *filename, fileaccess*

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The MC command connects P|R 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 *port address* must be previously defined with the MA command 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 1-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 *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 EPF 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 locates, 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 are 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 are 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 the NR parameter, execution does not halt, and you are not notified when EOF is reached. The input file reqinds automatically, and the simulator resumes reading from the file.

Note:

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 the core-memory map register 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.

md — *Delete Block From Memory Map*

Syntax **md** *address***,** *page*

Menu selection Configure→Memory Maps

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The MD command deletes a range of memory from the debugger's memory map.

❑ The *address* parameter identifies the starting address of the range of 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 display area of the Command window:

```
Specified map not found
```

❑ The *page* parameter is a 1-digit number that identifies the type of memory (program, 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

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.

mem — *Modify Memory Window Display*

Syntax **mem** *expression*[**@data** | **@prog** | **@io**] [**,** [*display format*] [**,** *window name*]]

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☐ PDM ☐ profiling

Description The MEM command identifies a new starting address for the block of memory displayed in the Memory window. The optional *window name* parameter opens an additional Memory window, allowing you to view a separate block of memory. The debugger displays the contents of memory at *expression* in the first data position in the Memory window. The end of the range is defined by the size of the window. The *expression* can be an absolute address, a symbolic address, or any C expression.

You can display either program or data memory:

❑ By default, the Memory window displays data memory. Although it is not necessary, you can explicitly specify data memory by following the *expression* parameter with a suffix of **@data**.

❑ You can display the contents of program memory by following the *expression* parameter with a suffix of **@prog**. When you do this, the Memory window's label changes to Memory [Prog] so 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 is displayed in one of the following formats:

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

mi — *Disconnect I/O Port* — **Simulator Only**

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

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The MI command disconnects a simulated I/O port from its associated input or output file.

- The *port address* parameter identifies the address of the I/O port, which must be defined previously with the MC command.
- The *page* parameter is a 1-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 read/write/execute characteristics must match the parameter used when the memory address was connected.

mix — *Enter Mixed Mode*

Syntax **mix**

Menu selection View→Mixed

Toolbar selection none

Environments ☑ basic debugger □ PDM □ profiling

Description The MIX command changes from the current debugging mode to mixed mode. If you are already in mixed mode, the MIX command has no effect.

ml — *List Memory Map*

Syntax **ml**

Menu selection Configure→Memory Maps

Toolbar selection none

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 a Window*

Syntax **move** *window name* [**,** [*X position*] [**,** [*Y position*] [**,** [*width*] [**,** *length*]]]]

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The MOVE command moves the upper left corner of the window to the specified XY position, repositioning the rest of the window relative to that corner. If you choose, you can resize the window while you move it (see the SIZE command for valid *width* and *length* values). Specify the *X position, Y position, width,* and *length* parameters in pixels. If you omit these parameters, the MOVE command defaults to the window's current position and size.

You can spell out the entire *window name*, but you need to specify only enough letters to identify the window.

mr — *Reset Memory Map*

Syntax **mr**

Menu selection none

Toolbar selection none

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 File→Save→Memory

Toolbar selection none

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 1-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, in words, of the block. This parameter can be any C expression.
- The *filename* is a system file. If you do not supply an extension, the debugger adds a .obj extension.

next

Single-Step, Next Statement

Syntax **next** [*expression*]

Menu selection Debug→Next

Toolbar selection {}

Environments ☑ basic debugger □ PDM □ profiling

Description

The NEXT command is similar to the STEP command. If you are in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time. Unlike STEP, NEXT never updates the display when executing called functions; NEXT always steps to the next consecutive statement. Unlike STEP, NEXT steps over function calls rather than stepping into them—you do not see the single-step execution of the function call.

The optional *expression* parameter specifies the number of statements that you want to single-step. You can use a conditional *expression* for conditional single-step execution. (Section 7.4, *Running Code Conditionally*, page 7-11, 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*.

pause

Pause Execution **Batch File Only**

Syntax **pause**

Menu selection none

Toolbar 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 a dialog box. To continue processing, click OK or press ↵.

pesc

Send ESC Key to Debuggers

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

Menu selection none

Toolbar 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. PESC halts program execution, but all processors in a group do not halt at the same real time; individual processors halt in the order they were added to the group.

The –g option identifies the group or processor that the command should be sent to. If you do not use this option, the ESC key is sent to the default group (dgroup).

pf

Profile, Full

Syntax **pf** *starting point* [**,** *update rate*]

Menu selection Tools→Profile→Profile Mode
Debug→Run

Toolbar selection

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 stopping point encountered. The *starting point* parameter can be a label, a function name, or a memory address.

The optional *update rate* parameter determines how often the Profile window is updated. The *update rate* parameter can have one of these values:

Value	Description
0	This is the default. Statistics are not updated until the session is halted (although you can force an update by clicking the mouse in the window).
≥ 1	Statistics are updated during the session. A value of **1** means that data is updated as often as possible.

phalt — *Halt Processors in Parallel*

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

Menu selection none

Environments □ basic debugger ☑ PDM □ profiling

Description The PHALT command halts one or more processors. If you send a PRUN or PRUNF command to a group or to an individual processor, you can use PHALT to halt the group or the individual processor. Each processor in a group is halted at the same real time. If you do not use the **–g** option to specify a group or a processor name, the PHALT command is sent to the default group (dgroup).

pinc — *Connect Pin*

Syntax **pinc** *pinname, filename*

Menu selection none

Environments ☑ basic debugger □ PDM □ profiling

Description The PINC command connects an input file to an interrupt pin.

- The *pinname* parameter identifies the interrupt pin and must be one of the 16 interrupt pins ($\overline{\text{INT}}$0–$\overline{\text{INT}}$3) or the $\overline{\text{BIO}}$ pin.
- The *filename* parameter is the name of your input file.

pind

Disconnect Pin

Syntax **pind** *pinname*

Menu selection none

Environments ☑ basic debugger □ PDM □ profiling

Description The PIND command disconnects an input file from an interrupt pin. The *pinname* parameter identifies the interrupt pin and must be one of the 16 interrupt pins ($\overline{\text{INT0}}$–$\overline{\text{INT3}}$) or the $\overline{\text{BIO}}$ pin.

pinl

List Pin

Syntax **pinl**

Menu selection none

Environments ☑ basic debugger □ PDM □ profiling

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

pq

Profile, Quick

Syntax **pq** *starting point* [**,** *update rate*]

Menu selection Tools→Profile→Profile Mode
Debug→Run

Toolbar selection ≣↓

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 stopping point encountered. PQ is similar to PF, except that PQ does not collect exclusive or exclusive max data.

The *update rate* parameter is the same as for the PF command.

pr

Resume Profiling Session

Syntax

pr [*clear data* [**,** *update rate*]]

Menu selection

Tools→Profile→Profile Mode
Debug→Run

Toolbar selection

≣↓

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 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 continues to collect data (adding the data to the existing data for the profiled areas) and to use the previous internal profile stacks.
nonzero	All previously collected profile data and internal profile stacks are cleared.

The *update rate* parameter is the same as for the PF and PQ commands.

profile

Switch to Profiling Environment

Syntax

profile

Menu selection

Tools→Profile→Profile Mode

Toolbar selection

none

Environments

☑ basic debugger □ PDM ☑ profiling

Description

The PROFILE command toggles between the basic debugger and profiling environments. If you enter PROFILE from the basic debugger environment, the debugger switches to the profiling environment. If you enter PROFILE from the profiling environment, the debugger switches to the basic debugger environment.

prompt — *Change Command-Line Prompt*

Syntax **prompt** *new prompt*

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The PROMPT command changes the command-line prompt. The *new prompt* can be any string of characters (a semicolon or comma ends the string). The *new prompt* cannot be longer than 132 characters.

prun — *Run Code in Parallel*

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

Menu selection none

Toolbar selection none

Environments □ 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 do not use this option, then code runs 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 returns control to the PDM command line. However, no debugger executing the command is interrupted.
- **With –r**, control is returned to the command line immediately, even if a debugger is still executing a command. You can type new commands, but the processors cannot execute the commands until they finish with the current command; however, you can perform PHALT, PESC, and STAT commands when the processors are still executing.

prunf — *Run Free in Parallel*

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

Menu selection none

Toolbar selection none

Environments □ basic debugger ☑ PDM □ profiling

Description The PRUNF command starts the processors running free, which means they are disconnected from the emulator. RUNF synchronizes the debuggers to cause the processors to begin execution at the same real time. The **–g** option identifies the group or processor that the command should be sent to. If you do not use this option, then code runs on the default group (dgroup).

The PHALT command stops a PRUNF; note that the debugger automatically executes a PHALT when the debugger is invoked.

pstep — *Single-Step in Parallel*

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

Menu selection none

Toolbar selection none

Environments □ basic debugger ☑ PDM □ profiling

Description The PSTEP command single-steps synchronously through assembly language code with interrupts disabled. RUNF synchronizes the debuggers to cause the processors to begin execution at the same real time. The **–g** option identifies the group or processor that the command should be sent to. If you do not use this option, then code runs on the default group (dgroup). You can use the PHALT command to stop a global run.

You can use the *count* parameter to specify the number of statements that you want to single-step.

Note:

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

quit

Exit Debugger

Syntax **quit**

Menu selection File→Exit

Toolbar 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.

reconnect

Reset Communication With Emulator **Emulator Only**

Syntax **reconnect**

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☑ PDM ☑ profiling

Description The RECONNECT command reinitializes communication between the debugger and the emulator. This command can be used after an unrecoverable fatal error.

Any software breakpoints set before a reconnect may still reside in memory after the reconnect. However, the debugger does not recognize that the breakpoints are set. You should reload memory in order to clear out any residual breakpoints.

reload

Reload Object Code

Syntax **reload** [*object filename*]

Menu selection File→Load→Reload Program

Toolbar selection none

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 Debug→Reset Target

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The RESET command resets the target system (emulator only) or simulator. This is a *software* reset.

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

restart

Reset PC to Program Entry Point

Syntax **restart**
rest

Menu selection Debug→Restart

Toolbar selection

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 Debug→Return

Toolbar selection

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 doing one of the following actions:

- Click the Halt icon on the toolbar:
- From the Debug menu, select Halt!.
- Press ESC .

run — *Run Code*

Syntax

run [*expression*]

Menu selection

Debug→Run

Toolbar selection

Environments

☑ basic debugger □ PDM □ profiling

Description

The RUN command is the basic command for running an entire program. The command's behavior depends on the type of parameter you supply:

- If you do not supply an *expression*, the program executes until it encounters a breakpoint or until you do one of the following actions:
  - Click the Halt icon on the toolbar:
  - From the Debug menu, select Halt!.
  - Press ESC .
- If you supply a logical or relational *expression*, the run becomes conditional. (Section 7.4, *Running Code Conditionally*, page 7-11, discusses this in detail.)
- If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger executes *count* instructions, halts, and updates the display.

runb

Benchmark Code

Syntax **runb**

Menu selection Debug→Run Benchmark

Toolbar 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. For RUNB 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 7.5, *Benchmarking*, on page 7-12.

runf

Run Free **Emulator Only**

Syntax **runf**

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☐ PDM ☐ profiling

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

The HALT command stops a RUNF; the debugger automatically executes a HALT when the debugger is invoked.

sa

Add Stopping Point

Syntax **sa** *address*

Menu selection none

Toolbar selection none

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.

sconfig — *Load Screen Configuration*

Syntax **sconfig** [*filename*]

Menu selection File→Load→Screen Layout

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The SCONFIG command restores the display to a specified configuration. This restores the window locations and sizes that were saved with the SSAVE command into *filename*. The debugger searches for the specified file in the current directory and then in directories named with the D_DIR environment variable. If you do not supply a *filename*, the debugger looks for init.clr.

When you use SCONFIG to restore a configuration that includes multiple File, Watch, or Memory windows, the additional windows are not opened automatically. However, when you open an additional window and use a *window name* that matches a window name that you used before you saved the configuration, the window is placed in the saved location.

sd — *Delete Stopping Point*

Syntax **sd** *address*

Menu selection none

Toolbar selection none

Environments □ basic debugger □ PDM ☑ profiling

Description The SD command deletes the stopping point at *address*.

send — *Send Debugger Command to Individual Debuggers*

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

Menu selection none

Toolbar selection none

Environments □ basic debugger ☑ PDM □ profiling

Description The SEND command sends any debugger command to an individual processor or to a group of processors. If the command produces a message, it is displayed in the COMMAND window for the appropriate debugger(s) and also in the PDM window.

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

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

set

Set a Variable to a String

Syntax

set [*group name* [**=** *list of processor names*]]
set [*variable* [**=** *string value*]]

Menu selection none

Toolbar selection none

Environments ☐ basic debugger ☑ PDM ☐ profiling

Description

The SET command allows you to create groups of processors to which you can send commands. With the SET command you can:

- **Define a group of processors.** It is useful to define a group when you plan to send commands to the same set of processors. The commands are sent to the processors in the same order in which you added the processors to the group. To define a group, specify a *group name* and then list the processors you want in the group.

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

- **List all groups of processors.** You can use the SET command without any parameters to list all the processors that belong to a group, in the order in which they were added to the group.

You can also use the SET command with system-defined variables to:

- **Change the prompt for the PDM.** To change the PDM prompt, use the SET command with the system variable called prompt. For example, to change the PDM prompt to 3PROCs, enter:

  ```
  set prompt = 3PROCs ↵
  ```

- ❑ **Check the execution status of the processors.** In addition to displaying the execution status of a processor or group of processors, the STAT command (described on page 13-59) 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 13-31).

- ❑ **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 12-18.

setf

Set Default Data-Display Format

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

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM □ profiling

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

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

  char short uint ulong double
  uchar int long float ptr

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

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

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

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

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

```
setf * ↵
```

size

Size a Window

Syntax **size** *window name* [**,** [*width*] [**,** *length*]]

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The SIZE command changes the size of the window. Specify the *width* and *length* parameters in pixels. If you omit these parameters, the SIZE command defaults to the window's current size.

You can spell out the entire *window name*, but you need to specify only enough letters to identify the window.

sl

List Stopping Point

Syntax **sl**

Menu selection none

Toolbar selection none

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 File→Load→Program Symbols

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The SLOAD command loads the symbol table of the specified object file. SLOAD is useful in an emulation environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). In such an environment, loading the symbol table allows you to perform symbolic debugging and examine the values of C variables.

SLOAD clears the existing symbol table before loading the new one but does not modify memory or set the program entry point. SLOAD closes any Watch windows.

sound — *Enable Error Beeping*

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

Menu selection none

Toolbar 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 would not see the error message). By default, sound is off.

spawn — *Invoke the 'C54x Debugger*

Syntax **spawn** **emu54x** **–n** *processor name* [*invocation options*]

Menu selection none

Toolbar 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.

- **emu54x** is the executable that invokes the debugger.

  The PDM associates the *processor name* with the actual processor according to which executable you use. To invoke a debugger, the PDM must be able to find the executable file for that debugger. The PDM first searches the current directory and then searches the directories listed with the PATH statement.

- **–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 or underscore characters and must begin with an alphabetic character. The name is not case sensitive. The processor name must match one of the names defined in your board configuration file (see Appendix C, *Describing Your Target System to the Debugger*).

sr — *Reset Stopping Point*

Syntax **sr**

Menu selection none

Toolbar selection none

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 File→Save→Screen Layout New File

Toolbar selection none

Environments ☑ basic debugger ☑ PDM ☑ profiling

Description The SSAVE command saves the current screen configuration to a file. This saves the window locations and window sizes for all debugging modes, including the size and location for multiple File, Watch, and Memory windows. However, the debugger does not save docking information about docked windows. If you have one or more docked windows and you save and reload the screen configuration, the debugger does not display any windows as docked. If you want the windows docked, you must follow the docking procedure again.

The *filename* parameter names the screen configuration file. You can include path information (including relative pathnames); if you do not supply path information, the debugger places the file in the current directory. If you do not supply a *filename*, the debugger saves the current configuration into a file named init.clr and places the file in the current directory.

If you use a filename that already exists, the debugger overwrites the file with the current configuration.

stat — *Find the Execution Status of Processors*

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

Menu selection none

Toolbar selection none

Environments □ basic debugger ☑ PDM □ profiling

Description The STAT command tells you whether a processor is running or halted. If a processor is halted when you execute this command, then the PDM also lists the current PC value for that processor. If you do not use the **–g** option, the PDM displays the status of the processors in the default group (dgroup).

step — *Single-Step*

Syntax **step** [*expression*]

Menu selection Debug→Step

Toolbar selection {↓}

Environments ☑ basic debugger □ PDM □ profiling

Description The STEP command single-steps through assembly language or C code. If you are in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time.

If you are single-stepping through C code and encounter a function call, the STEP command shows you the single-step execution of the called function (assuming that the function was compiled with the compiler’s –g option). When function execution is complete, single-step execution returns to the caller. If the function was not compiled with the –g option, the debugger executes the function but does not show single-step execution of the function.

The *expression* parameter specifies the number of statements that you want to single-step. You can use a conditional *expression* for conditional single-step execution. (Section 7.4, *Running Code Conditionally*, page 7-11, discusses this in detail.)

system — *Enter Operating-System Command*

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

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The debugger version of the SYSTEM command allows you to enter operating-system commands without explicitly exiting the debugger environment. If you enter SYSTEM with no parameters, the debugger opens a system shell and displays the operating-system prompt. At this point, you can enter any operating-system command. When you finish, enter:

```
exit ↵
```

If you prefer, you can supply the operating-system command as a parameter to the SYSTEM command. If the result of the command is a message or other display, the debugger blanks the top of the debugger display to show the information. In this case, you can use the *flag* parameter to tell the debugger whether or not it should hesitate after displaying the information. The *flag* can be 0 or 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 enter:

```
exit ↵.
(This is the default.)
```

system — *Enter Operating-System Command* — PDM Environment

Syntax **system** *operating-system command*

Menu selection none

Toolbar selection none

Environments □ basic debugger ☑ PDM □ profiling

Description The PDM version of the SYSTEM command allows you to enter a single operating-system command without explicitly exiting the PDM environment. You cannot enter more than one operating-system command with the PDM version of the SYSTEM command.

take — *Execute Batch File*

Syntax

Basic debugger: **take** *batch filename* [**,** *suppress echo flag*]
PDM: **take** *batch filename*

Menu selection File→Execute Take File

Toolbar selection none

Environments ☑ basic debugger ☑ PDM ☑ profiling

Description The TAKE command tells the debugger or the PDM to read and execute commands from a batch file. The *batch filename* parameter identifies the file that contains commands. If you do not supply a pathname as part of the filename, the debugger/PDM first looks in the current directory and then searches directories named with the D_DIR environment variable.

The *batch filename* for the PDM version of this command must have a .pdm extension, or the PDM will not be able to read the file. In addition, the batch file that the PDM reads can contain only PDM commands.

By default, the debugger echoes the commands to the display area of the Command window and updates the display as it reads the commands from the batch file. To suppress the echoing and updating, enter a 0 as the *suppress echo flag* parameter. If you omit the *suppress echo flag* parameter or enter a nonzero value for that parameter, the debugger behaves in the default manner.

unalias — *Delete Alias Definition*

Syntax **unalias** {*alias name* | *****}

Menu selection Configure→Alias Commands

Toolbar 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 * ↵
  ```

  The * symbol *does not* work as a wildcard.

unset — *Delete Group*

Syntax **unset** {*group name* | *****}

Menu selection none

Toolbar 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 * ↵
```

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 Additional Directory

Syntax **use** [*directory name*]

Menu selection none

Toolbar 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 in the display area of the Command window all of the current directories.

vaa

Save All Profile Data to a File

Syntax **vaa** *filename*

Menu selection Tools→Profile→Save All

Toolbar selection none

Environments □ basic debugger □ PDM ☑ profiling

Description The VAA command saves all statistics collected during the current profiling session. The data is stored in the emu470 system file.

vac

Save Displayed Profile Data to a File

Syntax **vac** *filename*

Menu selection Tools→Profile→Save View

Toolbar selection none

Environments □ basic debugger □ PDM ☑ profiling

Description The VAC command saves all statistics currently displayed in the Profile window. (Statistics that are not displayed are not saved.) The data is stored in a system file.

version

Display the Current Debugger Version

Syntax **version**

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The VERSION command displays the debugger's copyright date and version number, as well as the device name.

vr

Reset Profile Window Display

Syntax **vr**

Menu selection none

Toolbar selection none

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 the default sort order.

wa

Add Item to Watch Window

Syntax **wa** *expression* [@**prog** | @**data** | @**io**] [**,**[*label*] [**,** [*display format*] [**,** *window name*]]]

Menu selection Configure→Watch Add

Toolbar selection none

Environments ☑ basic debugger □ PDM □ profiling

Description The WA command displays the value of *expression* in a Watch window. If a Watch window is not open, executing WA opens a 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 do not use a *label*, the debugger displays the *expression* in the label field.

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

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

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

```
wa PC,,o ⏎
```

You can open additional Watch windows by using the *window name* parameter. When you open an additional Watch window, the debugger appends the *window name* to the Watch window label. You can create as many Watch windows as you need.

If you omit the *window name* parameter, the debugger displays the expression in the default Watch window (labeled Watch).

wd — *Delete Item From Watch Window*

Syntax **wd** *expression* [**,** *window name*]

Menu selection Configure→Watch Add

Toolbar selection none

Environments ☑ basic debugger □ PDM □ profiling

Description The WD command deletes a specific item from the Watch window. The WD command’s *expression* parameter must correspond to one of the variable names listed in the Watch window. The optional *window name* parameter specifies a particular Watch window. If no window name is given, the expression is deleted from the default Watch window.

whatis — *Find Data Type*

Syntax **whatis** *symbol*

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM □ profiling

Description The WHATIS command shows the data type of *symbol* in the display area of the Command window. The *symbol* can be any variable (local, global, or static), a function name, structure tag, typedef name, or enumeration constant.

win — *Make a Window Active*

Syntax **win** *window name*

Menu selection View menu options

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

Description The WIN command allows you to make a window active by name. You can spell out the entire window name, but you really need to specify only enough letters to identify the window.

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

wr — *Close Watch Window*

Syntax **wr** [{***** | *window name*}]

Menu selection Configure→Watch Add

Toolbar selection none

Environments ☑ basic debugger □ PDM □ profiling

Description The WR command deletes all items from a Watch window and closes the window.

- To close the default Watch window, enter:

  ```
  wr ↵
  ```

- To close one of the additional Watch windows, use this syntax:

  wr *window name*

- To close all Watch windows, enter:

  ```
  wr * ↵
  ```

zoom — *Zoom a Window*

Syntax **zoom** [*window name*]

Menu selection none

Toolbar selection none

Environments ☑ basic debugger □ PDM ☑ profiling

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

You can spell out the entire *window name*, but you really need to specify only enough letters to identify the window.

13.3 Summary of Profiling Commands

The following tables summarize the profiling commands that are used for marking, enabling, disabling, and unmarking areas and for changing the display in the Profile window. These commands are easiest to use from the Tools→Profile menu and associated dialog boxes, so they are not included in the alphabetical command summary. The syntaxes for these commands are provided here so that you can include them in batch files.

Marking areas

To mark this area...	In C only	In 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**	

Disabling marked areas

To disable this area...	In C only	In disassembly only	In C *and* disassembly
Lines			
❑ By line number, address	**DCLE** *filename*, *line number*	**DALE** *address*	not applicable
❑ All lines in a function	**DCLF** *function*	**DALF** *function*	**DBLF** *function*
❑ All lines in a module	**DCLM** *filename*	**DALM** *filename*	**DBLM** *filename*
❑ All lines everywhere	**DCLG**	**DALG**	**DBLG**
Ranges			
❑ By line number, address	**DCRE** *filename*, *line number*	**DARE** *address*	not applicable
❑ All ranges in a function	**DCRF** *function*	**DARF** *function*	**DBRF** *function*
❑ All ranges in a module	**DCRM** *filename*	**DARM** *filename*	**DBRM** *filename*
❑ All ranges everywhere	**DCRG**	**DARG**	**DBRG**

Disabling marked areas (Continued)

To disable this area...	In C only	In disassembly only	In C *and* disassembly
Functions			
❑ By function name	**DCFE** *function*	not applicable	not applicable
❑ All functions in a module	**DCFM** *filename*		**DBFM** *filename*
❑ All functions everywhere	**DCFG**		**DBFG**
All areas			
❑ All areas in a function	**DCAF** *function*	**DAAF** *function*	**DBAF** *function*
❑ All areas in a module	**DCAM** *filename*	**DAAM** *filename*	**DBAM** *filename*
❑ All areas everywhere	**DCAG**	**DAAG**	**DBAG**

Enabling disabled areas

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

Unmarking areas

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

Changing the profile window display

(a) Viewing specific areas

To view this area...	In C only	In disassembly only	In C *and* disassembly
Lines			
□ By line number, address	**VFCLE** *filename*, *line number*	**VFALE** *address*	not applicable
□ All lines in a function	**VFCLF** *function*	**VFALF** *function*	**VFBLF** *function*
□ All lines in a module	**VFCLM** *filename*	**VFALM** *filename*	**VFBLM** *filename*
□ All lines everywhere	**VFCLG**	**VFALG**	**VFBLG**
Ranges			
□ By line number, address	**VFCRE** *filename*, *line number*	**VFARE** *address*	not applicable
□ All ranges in a function	**VFCRF** *function*	**VFARF** *function*	**VFBRF** *function*
□ All ranges in a module	**VFCRM** *filename*	**VFARM** *filename*	**VFBRM** *filename*
□ All ranges everywhere	**VFCRG**	**VFARG**	**VFBRG**
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**

Chapter 14

Basic Information About C Expressions

Many of the debugger commands take C expressions as parameters. This allows the debugger to have a relatively small, yet powerful, instruction set. Because C expressions can have side effects—that is, the evaluation of some types of expressions can affect existing values—you can use the same command to display or to change a value. This reduces the number of commands in the command set.

This chapter contains basic information that helps you use C expressions as debugger command parameters.

Topic **Page**

14.1 C Expressions for Assembly Language Programmers

It is not necessary for you to be an experienced C programmer to use the debugger. However, to use the debugger's full capabilities, you should be familiar with the rules governing C expressions. You should obtain a copy of *The C Programming Language* (first or second edition) by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey. This book is referred to in the C community, and in Texas Instruments documentation, as *K&R*.

Note:

A single value or symbol is a legal C expression.

K&R contains a complete description of C expressions; to get you started, here is a summary of the operators that you can use in expression parameters.

❑ **Reference operators**

–>	indirect structure reference	.	direct structure reference
[ ]	array reference	*	indirection (unary)
&	address (unary)		

❑ **Arithmetic operators**

+	addition (binary)	–	subtraction (binary)
*	multiplication	/	division
%	modulo	–	negation (unary)
(*type*)	type cast		

❑ **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, the parameters they are used with have side effects.

- **Bitwise operators**

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

- **Assignment operators**

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

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

14.2 Using Expression Analysis in the Debugger

The debugger's expression analysis is based on C expression analysis. This includes all mathematical, relational, pointer, and assignment operators. However, a few limitations, as well as a few additional features, are not described in K&R.

Restrictions

The following restrictions apply to the debugger's expression analysis features.

- ❑ The sizeof operator is not supported.
- ❑ The comma operator (,) is not supported (commas are used to separate parameter values for the debugger commands).
- ❑ Function calls and string constants are currently not supported in expressions.
- ❑ The debugger supports a limited capability of type casts; the following forms are allowed:

  (*basic type* **)**
  (*basic type* ***** *...***)**
  ([*structure/union/enum***]** *structure/union/enum tag* **)**
  ([*structure/union/enum***]** *structure/union/enum tag* ***** *...* **)**

  You can use up to six ***** characters in a cast.

Additional features

- ❑ All floating-point operations are performed in double precision using standard widening. (This is transparent.) Floats are represented in IEEE floating-point format.
- ❑ All registers can be referenced by name. The 'C54x auxiliary registers are treated as integers and/or pointers.
- ❑ Void expressions are legal (treated like integers).
- ❑ The specification of variables and functions can be qualified with context information. Local variables (including local statics) can be referenced with the expression form:

  *function name***.***local name*

This expression format is useful for examining the automatic variables of a function that is not currently being executed. Unless the variable is static, however, the function must be somewhere in the current call stack. If you want to see local variables from the currently executing function, you need not use this form; you can simply specify the variable name (just as in your C source).

File-scoped variables (such as statics or functions) can be referenced with the following expression form:

*filename***.***function name*
or *filename***.***variable name*

This expression format is useful for accessing a file-scoped static variable (or function) that may share its name with variables in other files.

In this expression form, *filename* **does not include** the file extension; the debugger searches the object symbol table for any source filename that matches the input name, disregarding any extension. Thus, if the variable *ABC* is in file source.c, you can specify it as source.ABC.

These expression forms can be combined into an expression of the form:

*filename***.***function name***.***variable name*

❑ Any integral or void expression can be treated as a pointer and used with the indirection operator (*). Here are several examples of valid use of a pointer in an expression:

```
*123
*A5
*(A2 + 123)
*(I*J)
```

By default, the values are treated as integers (that is, these expressions point to integer values).

❑ Any expression can be type cast to a pointer to a specific type (overriding the default of pointing to an integer, as described above).

Hint: You can use casting with the WA and DISP commands to display data in a desired format.

For example, the expression:

```
*(float *)10
```

treats 10 as a pointer to a floating-point value at location 10 in memory. In this case, the debugger fetches the contents of memory location 10 and treats the contents as a floating-point value. If you use this expression as a parameter for the DISP command, the debugger displays memory contents as an array of floating-point values within the DISP window, beginning with memory location 10 as array member [0].

Note how the first expression differs from the expression:

```
(float)*10
```

In this case, the debugger fetches an integer from address 10 and converts the integer to a floating-point value.

You can also type cast to user-defined types such as structures. For example, in the expression:

```
((struct STR *)10)–>field
```

The debugger treats memory location 10 as a pointer to a structure of type STR (assuming that a structure is at address 10) and accesses a field from that structure.

Appendix A

What the Debugger Does During Invocation

In some circumstances, you may find it helpful to know the steps that the debugger goes through during the invocation process. These are the steps, in order, that the debugger performs. If you are using the PDM to run multiple debuggers, the PDM executes the first step. (For more information on the environment variables mentioned below, see Chapter 2, *Getting Started With the Debugger.*)

The debugger:

1) Reads options from the operating system's command line.

2) Reads any information specified with the D_OPTIONS environment variable.

3) Reads information from the D_DIR and D_SRC environment variables.

4) Looks for the init.clr screen-configuration file.

   (The debugger searches for the screen-configuration file in directories named with D_DIR.)

5) Initializes the debugger screen and windows.

6) Finds the batch file that defines your memory map by searching in directories named with D_DIR. The debugger expects this file to set up the memory map and follows these steps to look for the batch file:

   ❑ When you invoke the debugger, it checks to see if you have used the –t debugger option. If it finds the –t option, the debugger reads and executes the specified file.

   ❑ If you have not used the –t option, the debugger looks for the default initialization batch file for the emulator. The batch file name differs for each version of the debugger:

      ■ For the emulator, this file is named *emuinit.cmd.*
      ■ For the simulator, this file is named *siminit.cmd.*

If the debugger finds the file corresponding to your tool, it reads and executes the file. If the debugger does not find the –t option or the initialization batch file, it looks for a file called init.cmd. This allows you to have one initialization batch file for more than one debugger tool. To set up this file, you can use the IF/ELSE/ENDIF commands (see page 3-8 for more information) to indicate which memory map applies to each tool.

7) Loads any object files specified with D_OPTIONS or specified on the command line during invocation.

8) Determines the initial mode (auto, assembly, or mixed) and displays the appropriate windows on the screen.

At this point, the debugger is ready to process any commands that you enter.

Where the debugger looks for files

You can perform all load-type commands by using menu options. However, if you choose to use the command-line equivalents to these menu options, you need to know where the debugger looks for source files.

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

❏ If you are using LOAD, RELOAD, or SLOAD, you have only two choices for supplying the path information:

  ■ Specify the path as part of the filename.

  ■ Alternatively, you can use the CD command before you enter the LOAD, RELOAD or SLOAD command to change the current directory from within the debugger. The format for this command is:

  cd *directory name*

❏ If you are using the FILE command, you have several options:

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

  SET D_SRC=*pathname;pathname* For PCs

  setenv D_SRC "*pathname;pathname***"** For SPARCstations

  You can name several directories for the debugger to search.

- When you invoke the debugger, you can use the – i option to name additional source directories for the debugger to search. The format for this option is **–i** *pathname.*

  You can specify multiple pathnames by using several –i options (one pathname per option). The list of source directories that you create with –i options is valid until you quit the debugger.

- Within the debugger environment, you can use the USE command to name additional source directories. The format for this command is:

  use *directory name*

  You can specify only one directory at a time.

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

Appendix B

Customizing the Emulator Analysis Interface

The interface to the 'C54x emulator analysis module is register based. You can set up hardware breakpoints or counter events through the Tools→Analysis menu dialog boxes. In some cases, however, you may want to define more complex conditions for the processor to detect. Or, you may want to write a batch file that defines breakpoint and/or counter conditions. In either case, you can accomplish these tasks by accessing the analysis registers through the debugger. This appendix explains how to access these registers.

Topic **Page**

B.1 Loading the Analysis Pseudoregisters With Their Default Values

The analysis.cmd file supplied with the debugger defines a basic set of analysis commands. These commands, like the analysis dialog boxes, load the analysis registers with specified values. Before you can use these command aliases, you must execute the analysis.cmd file. To execute the analysis.cmd file, follow these steps:

1) From the File menu, select Execute Take File...

2) In the Open Take File dialog box, browse until you find analysis.cmd.

3) Select analysis.cmd and click Open.

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 B–1 shows the predefined commands along with their menu equivalents.

B.2 Summary of Predefined Analysis Commands

Table B–1 shows the predefined commands along with their dialog box equivalents. These 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 a foundation for you to build upon to create your own commands.

Table B–1. The Analysis Commands Found in the analysis.cmd File

Command	Analysis Events Dialog Box Tab→ Selection	Description	Page
asys_emu0out	Emulator pins → EMU0 trigger out	Set EMU0 pin to output	B-6
asys_emu1out	Emulator pins → EMU1 trigger out	Set EMU1 pin to output	B-6
asys_extcnt	Emulator pins → External clock	Use the external counter on the emulator	B-6
asys_off	Analysis → Enable Events (Use Enable Events on the Tools→Analysis menu rather than Setup Events.)	Turn off the analysis interface	B-5
asys_on	Analysis → Enable Events (Use Enable Events on the Tools→Analysis menu rather than Setup Events.)	Turn on the analysis interface	B-5
asys_reset	None	Reset the analysis interface	B-11
cnt_br	Analysis Count Events → Branch taken	Count any branches detected	B-7
cnt_call	Analysis Count Events → Call taken	Count any calls detected	B-7
cnt_clock	Analysis Count Events → CPU clock	Count CPU clock cycles	B-7
cnt_data	Analysis Count Events → Data bus	Count any data accesses	B-7
cnt_ins	Analysis Count Events → Instruction fetch	Count any instructions fetched	B-7
cnt_intr	Analysis Count Events → Interrupt/trap taken	Count any interrupts/traps detected	B-7
cnt_load *value*	Analysis Count Events → Event counter	Load the analysis counter	B-6
cnt_pclk	Analysis Count Events → Pipeline clock	Count CPU pipeline execution clocks	B-7
cnt_prog1 cnt_prog2	Analysis Count Events → Program bus 1 and 2	Count any program address accesses	B-7
cnt_ret	Analysis Count Events → Return taken	Count any returns from an interrupt/trap detected	B-7
data_bus_qual	Analysis Count/Break Events → Data Bus radio	Count or break on data bus events	B-10
data_io_address *address* or *symbol name*	Analysis Count/Break Events → Data or I/O bus: Address field	Set a breakpoint or a count event on a data address	B-8
data_io_mask *value*	Analysis Count/Break Events → Data or I/O bus: Mask field	Set a breakpoint or a count event on data value with mask	B-8
data_io_val *value*	Analysis Count/Break Events → Data or I/O bus: Value field	Set a breakpoint or a count event on data value	B-8

Table B–1. The Analysis Commands Found in the analysis.cmd File (Continued)

Command	Analysis Events Dialog Box Tab→ Selection	Description	Page
data_io_qual_r	Analysis Count/Break Events → Data or I/O Bus: Data bus: Read	Data or I/O read qualifier	B-10
data_io_qual_rw	Analysis Count/Break Events → Data or I/O Bus: Data bus: Access	Data or I/O read/write qualifier	B-10
data_io_qual_w	Analysis Count/Break Events → Data or I/O Bus: Data bus: Write	Data or I/O write qualifier	B-10
io_bus_qual	Analysis Count/Break Events → IO Bus radio	Count or break on I/O bus events	B-10
prog1_address *address or function name*	Analysis Count/Break Events → Program bus 1: Address field	Break or count Program Bus 1 events at *address*	B-8
prog2_address *address or function name*	Analysis Count/Break Events → Program bus 2: Address field	Break or count Program Bus 2 events at *address*	B-8
prog1_ext_add	Analysis Count/Break Events → Program bus 1: Extended Address	Enable Program Bus 1 extended addressing	B-8
prog2_ext_add	Analysis Count/Break Events → Program bus 2: Extended Address	Enable Program Bus 2 extended addressing	B-8
prog1_page	Analysis Count/Break Events → Program Bus 1: Page field	Specify count and break page field for Program Bus 1	B-8
prog2_page	Analysis Count/Break Events → Program Bus 2: Page field	Specify count and break page field for Program Bus 2	B-8
prog1_qual_iaq prog2_qual_iaq	Analysis Count/Break Events → Program bus 1 and 2: Fetch	Program instruction acquisition	B-10
prog1_qual_r prog2_qual_r	Analysis Count/Break Events → Program bus 1 and 2: Read	Program read qualifier	B-10
prog1_qual_rw prog2_qual_rw	Analysis Count/Break Events → Program bus 1 and 2: Access	Program read/write qualifier	B-10
prog1_qual_w prog2_qual_w	Analysis Count/Break Events → Program bus 1 and 2: Write	Program write qualifier	B-10
prog_win_on	Analysis Count/Break Events → Program window	Program windowing enabled	B-5
prog_win_off	Analysis Count/Break Events → Program window	Program windowing disabled	B-5
stop_br	Analysis Break Events → Branch taken	Halt the processor when a branch is detected	B-9
stop_call	Analysis Break Events → Call taken	Halt the processor when a call is detected	B-9
stop_cnt	Analysis Count Events → Event counter < 0	Halt the processor when the counter passes 0	B-6
stop_data	Analysis Break Events → Data bus	Halt the processor on a data bus access	B-8
stop_disc	Analysis Break Events → Discontinuity	Halt any discontinuity	B-9

Table B–1. The Analysis Commands Found in the analysis.cmd File (Continued)

Command	Analysis Events Dialog Box Tab→ Selection	Description	Page
stop_emu0	Analysis Break Events → EMU0 driven low	Halt the processor when the EMU0 pin is low	B-9
stop_emu1	Analysis Break Events → EMU1 driven low	Halt the processor when the EMU1 pin is low	B-9
stop_intr	Analysis Break Events → Interrupt/trap taken	Halt the processor when an interrupt/trap is detected	B-9
stop_off	None	Disable break events	B-9
stop_pclk	Analysis Break Events → Pipeline clock	Halt the processor on a pipe-clock (instruction fetched)	B-9
stop_prog1 stop_prog2	Analysis Break Events → Program bus 1 and 2	Halt the processor on a program bus access	B-8
stop_ret	Analysis Break Events → Return taken	Halt the processor when a return from an interrupt/trap is detected	B-9

In addition to the predefined commands listed in Table B–1, you can create your own analysis commands using the ALIAS and EVAL commands. Refer to sections 3.1, page 3-2, and 8.2, page 8-3, for more information on ALIAS and EVAL. The following subsections briefly describe the use of the analysis commands.

Enabling the analysis interface

The basic syntax for the command to enable the analysis module is:

asys_on

The syntax for the command to disable the analysis interface is:

asys_off

Enabling the program window

The basic syntax for the command to enable the analysis program window is:

prog_win_on

The syntax for the command to disable the program window 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 B–2 shows the command for counting each of the nine events.

Table B–2. The Analysis Commands

Command	Dialog Box 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                    Set a program address breakpoint on
prog2_brk_add main                                     function_name.
stop_prog                             Enable the processor to stop on the
                                                  breakpoint condition.
run                                                    Run the program.
prog1_brk_add My_Function                  Set a new program address
prog2_brk_add My_Function              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 B–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 B–3. Breakpoint Commands for Program and Data Addresses

Command	**Dialog Box Selection**	**Description**
data_io_address *address* or *symbol name*	Data bus	Set a data breakpoint address
data_io_val *value*	Data bus	Set a data value
data_io_mask *value*	Data bus	Set a data value mask
prog1_address *address* or *function name*	Program bus 1	Set a program breakpoint address
prog2_address *address* or *function name*	Program bus 2	Set a program breakpoint address
prog1_ext_add prog2_ext_add	Program bus	Enable Program Bus 1or 2 extended addressing
prog1_page prog2_page	Program bus	Specify count and break page field for Program bus
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 B–4 shows the commands for stopping the processor when an event occurs. You can set breakpoints on any combination of these events.

Table B–4. Breakpoint Commands for Event Occurrences

Command	Dialog Box 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
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 B–5 shows the qualifier commands for data and program break events. You can use only one of these commands at a time.

Table B–5. Read and Write Qualifying Commands for Data and Program Accesses

Command	**Dialog Box Selection**	**Description**
data_bus_qual	Data bus: Radio	Count or break on Data bus events
data_io_qual_r	Data bus: Read	Look only at data reads
data_io_qual_rw	Data bus: Access	Look at both data reads and writes
data_io_qual_w	Data bus: Write	Look only at data writes
io_bus_qual	IO bus: Radio	Count or break on I/O bus events
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
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 can 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 module, type:

```
asys_reset ↵
```

Note:

To clear conditions or qualifier expressions previously defined via the Tools→Analysis menu, you must open the Analysis count events and Analysis break events dialog boxes and deselect each defined condition.

B.3 Creating Customized Analysis Commands

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 EVAL 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 module, enter:

```
alias analysis_on, ”eval anaenbl = 1”
```

To create a new command for counting branches detected, enter:

```
alias cb, ”e evtselt = 12”
```

B.4 Summary of Analysis Pseudoregisters

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.

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 as follows:

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 do not interfere with the status bits because they are cleared before command execution.

datbrkp (data breakpoint address)

The datbrkp pseudoregister allows you to specify a breakpoint address for each of the major buses in the ’C54x 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 is 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 are 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 bits are defined as follows:

Qualifier Code	Definition
0	Read
1	Write
2	Reserved
3	Read/write

evtcntr (event counter)

This register represents a true value in the ’C54x 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 displays 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) contains 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 ’C54x 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 ’C54x 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 ’C54x 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 bits are defined as follows:

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 address 2. 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 ’C54x 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 B–1. Program Discontinuity

```
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 shows 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 C

Describing Your Target System to the Debugger

For the debugger to understand how you have configured your target system, you must supply the target configuration information in a file for the debugger to read.

- ❏ If you are using an emulation scan path that contains only one ’C54x and no other devices, you can use the *board.dat* file that comes with the ’C54x emulator kit. This file describes to the debugger the single ’C54x in the scan path and gives the ’C54x the name CPU_A. Because the debugger automatically looks for a file called board.dat in the current directory and in the directories specified with the D_DIR environment variable, you can skip this appendix.
- ❏ If you plan to use a target system that has multiple ’C54x devices or that includes devices other than the ’C54x, 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. Each entry of the file describes one device on your scan path and the entries follow the order of the devices in the scan path. The text version of the configuration file is referred to as *board.cfg* in this book.

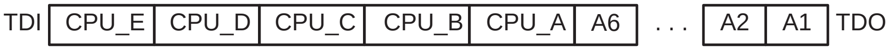
Example C–1 shows a board.cfg file that describes a possible ’C54x device chain. It lists six octals named A1–A6, followed by five ’C54x devices named CPU_A, CPU_B, CPU_C, CPU_D, and CPU_E.

Example C–1. A Sample TMS320C54x 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″	TMS320C54x	;the first ’C54x
″CPU_B″	TMS320C54x	
″CPU_C″	TMS320C54x	
″CPU_D″	TMS320C54x	
″CPU_E″	TMS320C54x	;the last ’C54x nearest TDI ;(test data in)

(b) A sample ’C54x device chain



The order in which you list each device is important. The emulator scans the devices, assuming that the data from one device is followed by the data of the next device on the chain. Data from the device that is closest to the emulation header’s TDO (test data out) reaches the emulator first. The device whose data reaches the emulator first is listed first in the board.cfg file; the device whose data reaches the emulator last is listed last in the board.cfg file.

The board.cfg file can have any number of each of these three types of entries:

- ❑ **Debugger devices** such as the ’C54x. 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 (’C54xs) 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 ’C54xs.

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 ’C54x. 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:

  - ■ **TMS320C54x** is an example of a debugger-device type. TMS320C54x describes the ’C54x.
  - ■ **SPL** specifies the scan path linker and must be followed by four subpaths, as in this syntax:

    ”*device name***”** **SPL {***subpath0***} {***subpath1***} {***subpath2***} {***subpath3***}**

    Each *subpath* can contain any number of devices. However, an SPL subpath *cannot* contain another SPL. A subpath that contains no devices must still be listed.

Example 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″        TMS320C54x     ;the first ′C54x
″HUB″          SPL            ;the scan path linker
  {                           ;the first subpath
  ″B1″         BYPASS08
  ″B2″         BYPASS08
  ″CPU_B″      TMS320C54x     ;the second ′C54x
  }
  {                           ;the second subpath
  ″C1″         BYPASS08
  ″C2″         BYPASS08
  ″CPU_C″      TMS320C54x     ;the third ′C54x
  }
  {                           ;the third subpath (contains nothing)
  }
  {                           ;the fourth subpath
  ″D1″         BYPASS08
  ″D2″         BYPASS08
  ″CPU_D″      TMS320C54x     ;the fourth ′C54x
  }
″CPU_E″        TMS320C54x     ;the last ′C54x 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 is not 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: Specifying the Configuration File When Invoking the Debugger

When you invoke a debugger (either from the PDM or at the system prompt), the debugger must be able to find the board.dat file so that it knows how you have set up your scan path. The debugger looks for the board.dat file in the current directory and in the directories named with the D_DIR environment variable.

If you used a name other than board.dat or if the board.dat file is not in the current directory or in a directory named with D_DIR, you must use the –f option when you invoke the debugger. The –f option allows you to specify a board configuration file (and pathname) to be used instead of board.dat. The format for this option is:

–f *filename*

Appendix D

Debugger Messages

This appendix contains an alphabetical listing of the progress and error messages that the debugger or PDM might display in the display area of the Command window or in the PDM display area. Each listing contains both a description of the situation that causes the message and an action to take if the message indicates a problem or error.

Topic **Page**

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.

If you are using the debugger with Windows 95 or Windows NT, you must be sure that you have sound enabled in the control panel.

D.2 Alphabetical Summary of Debugger Messages

‘]’ expected

Description This is an expression error—it means that the parameter contained an opening bracket symbol but did not contain a closing bracket symbol.

Action See section D.4, *Additional Instructions for Expression Errors*, page D-25.

‘)’ expected

Description This is an expression error—it means that the parameter contained an opening parenthesis symbol but did not contain a closing parenthesis symbol.

Action See section D.4, *Additional Instructions for Expression Errors*, page D-25.

A

Aborted by user

Description The debugger halted a long Command display listing because you pressed the ESC key.

Action None required; this is normal debugger behavior.

B

Breakpoint already exists at *address*

Description During single-step execution, the debugger attempted to set a breakpoint where one already existed. (This is not necessarily a breakpoint that you set—it may have been an internal breakpoint that the debugger set for single-stepping).

Action None should be required; you may want to reset the program entry point (Debug→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 Open the Breakpoint Control dialog box by selecting Breakpoints from the Configure menu. 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 can invoke the debugger with the –v option so that fewer symbols may be loaded, or you can relink your program and link in fewer modules at a time.

Cannot allocate system memory

Description This is a fatal error—it means that the debugger is running out of memory.

Action You can invoke the debugger with the –v option so that fewer symbols may be loaded, or you can relink your program and link in fewer modules at a time.

Cannot connect file to program memory

Description An attempt has been made to connect a file to program memory using the MC command.

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 the emurst command is reset. Follow the steps described below and then restart your emulator.

Action

- ❑ 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 the –p option used with the D_OPTIONS environment variable matches the I/O address defined by your switch settings. (See page 2-5 for more information on the D_OPTIONS environment variable.)
  - ■ Check to see if you have a conflict in address space with another bus setting. If you have a conflict, change the switches on your board to one of the alternate settings listed in the installation guide. Modify the –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 is automatically updated.

Cannot find/open initialization file

Description The debugger cannot find the init.cmd file.

Action Be sure that init.cmd is in the appropriate directory. If it is not, copy it from the debugger product diskette. If the file is already in the correct directory, verify that the D_DIR environment variable is set up to identify the directory. See the information about setting up the debugger environment information included with your installation instructions.

Cannot halt the processor

Description This is a fatal error—for some reason, pressing ESC did not halt program execution.

Action Exit the debugger. Invoke the autoexec.bat file, then invoke the debugger again.

Cannot initialize target system

Description This error occurs while you are invoking the debugger with the emulator. A variety 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 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. See section 2.5 on page 2-6.

For more information on setting up the D_OPTIONS environment variable, see page 2-5 .

Cannot map into reserved memory: ?

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 cannot find the screen-customization file that you specified. The debugger also displays this message when you try to load a screen-customization file that was saved by an older version of the debugger.

Action

❑ Be sure that the filename was typed correctly. If it was not, reenter the command with the correct name. If it was, reenter the command and specify full path information with the filename.

❑ Be sure that the screen-customization file was saved using the current version of the debugger rather than an older version of the debugger.

Cannot open “*filename*”

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 is not possible.

Action Close any unnecessary windows. Windows that can be closed include Watch, File, Calls, and Memory windows. To close any of these windows, make the desired window active and press CONTROL F4 .

Cannot open object file: "*filename*"

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 are loading an actual object file. Be sure that the file was linked. You may want to run cl500 (with the –z option) or lnk500 again to create an executable object file.

Cannot read processor status

Description This is a fatal error—for some reason, pressing ESC did not halt program execution.

Action Exit the debugger. Invoke the autoexec.bat file, then invoke the debugger again. If you are using the emulator, check the cable connections.

Cannot reset the processor

Description This is a fatal error—for some reason, pressing ESC did not halt program execution.

Action Exit the debugger. Invoke the autoexec.bat file, then invoke the debugger again. If you are using the emulator, check the cable connections.

Cannot restart processor

Description The debugger attempted to reset the PC to the program entry point, but the debugger could not find an entry point.

Action Either define an entry point in your program, or do not use Debug→Restart or RESTART when your program does not 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. 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 was not in ROM, see section D.5, *Additional Instructions for Hardware Errors*, page D-25.

Cannot step

Description There is a problem with the target system.

Action See section D.5, *Additional Instructions for Hardware Errors*, page D-25.

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, *Additional Instructions for Expression Errors*, page D-25.

Command “*command*” not found

Description The debugger did not recognize the command that you typed.

Action Reenter the correct command. See Chapter 13, *Summary of Commands*.

Command timed out, emulator busy

Description There is a problem with the target system.

Action See section D.5, *Additional Instructions for Hardware Errors*, page D-25.

Conflicting map range

Description A block of memory specified with the Configure→Memory Maps menu option or the MA command overlaps an existing memory map entry. Blocks cannot overlap.

Action Use Configure→Memory Maps or the ML command to list the existing memory map; this helps you find the existing block that the new block would overlap. If the existing block is not necessary, delete it with the Memory Map Control dialog box or with the MD command. Use the Memory Map Control dialog box or the MA command to redefine the block of memory. If the existing block is necessary, use the Memory Map Control dialog box or the MA command to define a range that does not overlap the existing block.

Corrupt call stack

Description The debugger tried to update the Calls window and could not. This message is displayed in the following situations:

- A function was called that did not return.
- The program stack was overwritten in target memory.
- You are debugging code that has optimization enabled (for example, you did not use the –g compile option); if this is the case, ignore this message—code execution is not affected.

Action If your program called a function that did not return, then this is normal behavior (as long as you intended for the function not to return). Otherwise, you may be overwriting program memory.

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, see page 2-13.

EOF reached –connected at port: <*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.

Error in expression

Description This is an expression error.

Action See section D.4, *Additional Instructions for Expression Errors*, page D-25.

Execution error

Description There is a problem with the target system.

Action See section D.5, *Additional Instructions for Hardware Errors*, page D-25.

File already tied to port

Description You attempted to connect to an address that already has a file connected to it.

Action Connect the file to a mapped port that is not connected to a file.

File already tied to this pin

Description You attempted to connect an input file to an interrupt pin that already has a file connected to it.

Action Use the PINC command to connect the file to another interrupt pin that is not connected to a file.

File does not exist

Description The port file could not be opened for reading.

Action Be sure that the file exists as named. If it does, enter the USE command to identify the file’s directory.

Files must be disconnected from ports

Description You attempted to delete a memory map that has files connected to it.

Action You must disconnect a port with the MI command before you can delete it from the memory map.

File not found

Description The filename specified for the FILE command was not found in the current directory or any of the directories identified with D_SRC.

Action Be sure that the filename was typed correctly. If it was, reenter the FILE command and specify full path information with the filename.

File not found : “*filename*”

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 too large (*filename*)

Description You attempted to load a file that exceeded the maximum loadable COFF file size.

Action Loading the file without the symbol table (SLOAD), or use cl500 (with the –z option) or lnk500 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, *Additional Instructions for Expression Errors*, page D-25.

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 cast

Description This is an expression error—the expression parameter uses a cast that does not meet the C language rules for casts.

Action See section D.4, *Additional Instructions for Expression Errors*, page D-25.

Illegal left hand side of assignment

Description This is an expression error—the left-hand side of an assignment expression does not meet C language assignment rules.

Action See section D.4, *Additional Instructions for Expression Errors*, page D-25.

Illegal memory access

Description Your program tried to access unmapped memory.

Action Modify your source code. Alternatively, you can check and modify your memory map.

Illegal operand of &

Description This is an expression error—the expression attempts to take the address of an item that does not have an address.

Action See section D.4, *Additional Instructions for Expression Errors*, page D-25.

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, *Additional Instructions for Expression Errors*, page D-25.

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, *Additional Instructions for Expression Errors*, page D-25.

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, *Additional Instructions for Expression Errors*, page D-25.

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, *Additional Instructions for Expression Errors*, page D-25.

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, *Additional Instructions for Expression Errors*, page D-25.

Integer not allowed

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

Action See section D.4, *Additional Instructions for Expression Errors*, page D-25.

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 13, *Summary of Commands*.

Invalid memory attribute

Description The third parameter of the MA command specifies the type, or attribute, of the block of memory that is added to the memory map. The parameter entered did not match one of the valid attributes.

Action Reenter the MA command. Use one of the following valid parameters to identify the memory type:

R, ROM	Read-only memory
W, WOM	Write-only memory
R|W, RAM	Read/write memory
RAM|EX, R|W|EX	Read/write external memory
P|R	Read-only peripheral frame
P|R|W	Read/write peripheral frame

Invalid object file

Description Either the file specified with File→Load→Load Program, File→Load→Reload Program, File→Load→Program Symbols, the LOAD, the SLOAD, or the RELOAD command is not an object file that the debugger can load, or it has been corrupted.

Action Be sure that you are loading an actual object file. Be sure that the file was linked. You may want to run cl500 (with the –z option) or lnk500 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 cl500.

Invalid watch delete

Description The debugger cannot delete the parameter supplied with the WD command.

Action Reenter the WD command. Be sure to specify the symbol name that matches the item you want to delete.

Invalid window position

Description The debugger cannot move the window to the XY position specified with the MOVE command. Either the XY parameters are not within the screen limits, or the active window may be too large to move to the desired position.

Action Reenter the MOVE command. Enter the X and Y parameters in pixels.

Invalid window size

Description The width and length specified with the SIZE or MOVE command may be too large or too small. If valid width and length were specified, then the active window is already at the far right or bottom of the screen and so cannot be made larger.

Action Reenter the SIZE command. Enter the width and length in pixels.

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, *Additional Instructions for Hardware Errors*, page D-25.

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, *Additional Instructions for Hardware Errors*, page D-25.

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, *Additional Instructions for Expression Errors*, page D-25.

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, *Additional Instructions for Hardware Errors*, page D-25.

Memory map table full

Description Too many blocks have been added to the memory map. This rarely happens unless blocks are added word by word (which is inadvisable).

Action Stop adding blocks to the memory map. Consolidate any adjacent blocks that have the same memory attributes.

N

Name “*name*” not found

Description The command cannot find the object named *name*.

Action If *name* is a symbol, be sure that it was typed correctly. If it was not, reenter the command with the correct name. If it was, then be sure that the associated object file is loaded.

Nesting of repeats cannot exceed 100

Description The debugger cannot simulate more than 100 levels of repeat nesting in an input data file. If more than 100 instances are requested, the debugger disconnects the input file from the pin.

Action Correct the input file so that the data does not include nesting repetition exceeding 100. Use the PINC command to reconnect the input file to the desired pin.

No file connected to this pin

Description You tried to disconnect the input file from a pin that was not previously connected to that pin.

Action Use the PINL command to list all of the pins and the files connected to them. Use the PIND command to reenter the correct pinname and filename.

P

Pinname not valid for this chip

Description You attempted to connect or disconnect an input file to an invalid interrupt pin.

Action Reconnect or disconnect the input file to an unused interrupt pin ().

Pointer not allowed

Description This is an expression error.

Action See section D.4, *Additional Instructions for Expression Errors*, page D-25.

Processor is already running

Description One of the RUN commands was entered while the debugger was running free from the target system.

Action Enter the HALT command to stop the free run, then reenter the desired RUN command.

R

Read not allowed for port

Description You attempted to connect a file for input operation to an address that is not configured for read.

Action Remap the port of correct the access in your source code.

Register access error

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

Action See section D.5, *Additional Instructions for Hardware Errors*, page D-25.

S

Specified map not found

Description The MD command was entered with an address or block that is not in the memory map.

Action Use the ML command to verify the current memory map. When using MD, you can specify only the first address of a defined block.

Structure member name required

Description This is an expression error—a symbol name is followed by a period but no member name.

Action See section D.4, *Additional Instructions for Expression Errors*, page D-25.

Structure member not found

Description This is an expression error—an expression references a nonexistent structure member.

Action See section D.4, *Additional Instructions for Expression Errors*, page D-25.

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, *Additional Instructions for Expression Errors*, page D-25.

T

Take file stack too deep

Description Batch files can be nested up to ten levels deep. The batch file that you tried to execute with File→Execute Take File or the TAKE command calls batch files that are nested more than ten levels deep.

Action Edit the batch file that caused the error. Instead of calling another batch file from within the offending file, you can copy the contents of the second file into the first. This will removes a level of nesting.

Too many breakpoints

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 Open the Breakpoint Control dialog box by selecting Breakpoints from the Configure menu. Delete individual software breakpoints.

Too many paths

Description More than 20 paths have been specified cumulatively with the USE command, D_SRC environment variable, and –i debugger option.

Action Do not enter the USE command before entering another command that has a *filename* parameter. Instead, enter the second command and specify full path information for the *filename*.

U

Undeclared port address

Description You attempted to do a connect/disconnect on an address that is not declared as a port.

Action Verify the address of the port to be connected or disconnected.

User halt

Description The debugger halted program execution because you clicked the Halt icon on the toolbar, you selected Halt! from the Debug menu, or 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. Here are the valid window names; the bold letters show the smallest acceptable abbreviations:

Calls
CPU
Command
Disassembly
Memory
Profile
Watch

Write not allowed for port

Description You attempted to connect a file for output operation to an address that is not configured for write.

Action Either change the software to write a port that is configured for write, or change the attributes of the port.

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 take.

Note:

If errors are detected in a TAKE file, the PDM aborts the batch file execution, and the file line number of the invalid command is displayed along with the error message.

C

Cannot communicate with *“name”*

Description The PDM cannot communicate with the named debugger, because the debugger either crashed or was exited.

Action Spawn the debugger again.

Cannot communicate with the child debugger

Description This error occurs when you are spawning a debugger. The PDM was able to find the debugger executable file, but the debugger could not be invoked for some reason, and the communication between the debugger and PDM was never established. This usually occurs when you have a problem with your target system.

Action Exit the PDM and go back 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 are not using all of them, close some of them. If you are under a UNIX environment, use the ipcs command to check your message queues; use ipcrm to clean up the message queues.

Cannot open log file

Description The PDM cannot find the filename that you supplied when you entered the DLOG command.

Action Be sure that the file resides in the current directory or in one of the directories specified by the D_DIR environment variable.

- ❑ Check to see if you mistyped the filename.
- ❑ Be sure that the file has readable rights.

Cannot open take file

Description The PDM cannot find the batch filename supplied for the TAKE command. You will also see this message if you try to execute a batch file that does not have a .pdm extension.

Action Be sure that the file resides in the current directory or in one of the directories specified by the D_DIR environment variable.

- ❑ Check to see whether you mistyped the filename.
- ❑ Be sure that the batch filename has a .pdm extension.
- ❑ Be sure that the file has executable rights.

Cannot open temporary file

Description The PDM is unable create a temporary file in the current directory.

Action Change the permissions of the current directory.

Cannot seek in file

Description While the PDM was reading a file, the file was deleted or modified.

Action Be sure that the files the PDM reads are not deleted or modified during the read.

Cannot spawn child debugger

Description The PDM could not spawn the debugger that you specified, because the PDM could not find the debugger executable file (emu54x). 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 2048. 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 do not 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 13, *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 12.7, *Understanding the PDM's Expression Analysis*, page 12-17, 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 10 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 10 other batch files. The TAKE command can handle batch files that are nested up to 10 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 have 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 ’C54x must be reset before you can use the emulator. Most target systems reset the ’C54x 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 does not show the File window, no matter what type of code is currently running.

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

auto mode: A context-sensitive debugging mode that automatically switches between showing assembly language code in the Disassembly window and C code in the File window, depending on what type of code is currently running.

B

batch file: One of two types of files. The first type contains DOS commands for the PC to execute. The second type of batch file contains debugger commands for the debugger to execute. The PC does not execute debugger batch files, and the debugger does not execute PC batch files.

benchmarking: A type of program execution that allows you to track the number of CPU cycles consumed by a specific section of code.

big endian: An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *little endian*

breakpoint: A point within your program where execution will halt because of a previous request from you.

C

Calls window: A window that lists the functions called by your program.

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

cl500: A shell utility that invokes the 'C54x compiler, assembler, and linker to create an executable object file version of your program.

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

code-display windows: Windows that show code, text files, or code-specific information. This category includes the Disassembly, File, and Calls windows.

command line: The portion of the Command window where you can enter commands.

Command window: A window that provides an area for you to enter commands and for the debugger to echo command entry, show command output, and list progress or error messages.

common object file format (COFF): A binary object file format that promotes modular programming by supporting the concept of *sections*. All COFF sections are independently relocatable in memory space; you can place any section into any allocated block of target memory.

CPU window: A window that displays the contents of 'C54x on-chip registers, including the program counter, status register, A-file registers, and B-file registers.

cursor: An icon on the screen (such as an arrow or a horizontal line) that is used as a pointing device. The cursor is usually under mouse or keyboard control.

D

D_DIR: An environment variable that identifies the directory containing the commands and files necessary for running the debugger.

D_OPTIONS: An environment variable that you can use for identifying often-used debugger options.

D_SRC: An environment variable that identifies directories containing program source files.

data-display windows: Windows for observing and modifying various types of data. This category includes the Memory, CPU, and Watch windows.

debugger: A window-oriented software interface that helps you to debug ’C54x programs running on a ’C54x emulator or simulator.

disassembly: Assembly language code formed from the reverse-assembly of the contents of memory.

Disassembly window: A window that displays the disassembly (reverse assembly) of memory contents.

display area: The portion of the Command window or PDM window where the debugger/PDM echoes command entry, shows command output, and lists progress or error messages.

dock (a window): To anchor a floating window to an outer edge of the debugger application window. A docked window has no title bar and cannot be moved. However, a docked window can be resized.

drag: To move an object on the debugger display by pressing one of the mouse buttons and moving the mouse.

E

EISA: *Extended Industry Standard Architecture.* A standard for PC buses.

emulator: A debugging tool that is external to the target system and provides direct control over the ’C54x processor that is on the target system.

emurst: The command that involves the utility that resets the emulator; also, the utility itself.

environment variable: A special system symbol that the debugger uses for finding directories or obtaining debugger options.

F

File window: A window that displays the contents of the current C code. The File window is intended primarily for displaying C code but can be used to display any text file.

float (a window): To cause a debugger window to sit on top of the debugger application window outside the edges of the debugger application window. A floating window always appears active.

I

init.cmd: A batch file that contains debugger-initialization commands. If this file is not present when you first invoke the debugger, then all memory is invalid.

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

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

L

little endian: An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*

M

memory map: A map of memory space that tells the debugger which areas of memory can and cannot be accessed.

Memory window: A window that displays the contents of memory.

menu bar: A row of pulldown menu selections found at the top of the debugger display.

mixed mode: A debugging mode that simultaneously shows both assembly language code in the Disassembly window and C code in the File window.

O

open-collector output: An output circuit that actively drives both high and low logic levels.

P

PC: Personal computer or program counter, depending on the context and where it is used in this book: 1) In installation instructions or information relating to hardware and boards, *PC* means *personal computer*. 2) In general debugger and program-related information, *PC* means *program counter*, which is the register that identifies the current statement in your program.

PDM: *Parallel Debug Manager.* A program used for creating and controlling multiple debuggers for the purpose of debugging code in a parallel-processing environment.

point: To move the mouse cursor until it overlays the desired object on the screen.

port address: The PC I/O memory space that the debugger uses for communicating with the emulator. The port address is selected via switches on the emulator board and communicated to the debugger with the –p debugger option.

pulldown menu: A command menu that is accessed by name or with the mouse from the menu bar at the top of the debugger display.

R

ripple-carry output signal: An output signal from a counter indicating that the counter has reached its maximum value.

S

scalar type: A C type in which the variable is a single variable, not composed of other variables.

scroll bar: A bar on the right side or bottom of a window that allows you to adjust the contents of the window to display hidden information.

scroll bar handle: The rectangular box in the center of the right scroll bar in the Disassembly or Memory window that marks the center of disassembled code or memory contents.

scrolling: A method of moving the contents of a window up, down, left, or right to view contents that were not originally shown.

section: A relocatable block of code or data that ultimately occupies contiguous space in the memory map.

side effects: A feature of C expressions in which using an assignment operator in an expression affects the value of one of the components used in the expression.

simulator: A development tool that simulates the operation of the ’C54x and lets you execute and debug applications programs by using the C source debugger.

single-step: A form of program execution that allows you to see the effects of each statement. The program is executed statement by statement; the debugger pauses after each statement to update the data-display windows.

status bar: An area at the bottom of the debugger application window that displays context-sensitive help and the status of the processor.

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

T

target system: A ’C54x board that works with the emulator; the emulator does not contain a ’C54x device, so it must use a ’C54x target board. Usually, the target system is a board that you have designed; you use the emulator and debugger to help you debug your design.

totem-pole output: An output circuit that actively drives both high and low logic levels.

W

Watch window: A window that displays the values of selected expressions, symbols, addresses, and registers.

window: A defined rectangular area of space on the display.

Index