TEXAS INSTRUMENTS

TMS320C6x C Source Debugger

For SPARCstations

User's Guide

1997 Digital Signal Processing Solutions

Printed in U.S.A., January 1997
D426029-9761 revision*

SPRU224

User's Guide

TMS320C6x C Source Debugger

For SPARCstations

1997

TMS320C6x C Source Debugger User's Guide

For SPARCstations

Literature Number: SPRU224
Manufacturing Part Number: D426029-9761 revision *
January 1997

Preliminary

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

What Is This Book About?

This book tells you how to use the TMS320C6x C source debugger with the simulator.

There are two debugger environments: the basic debugger environment and the profiling environment. The basic debugger environment is a general-purpose debugging environment. The profiling environment is a special environment for collecting statistics about code execution.

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

How to Use This Manual

The goal of this book is to help you learn to use the Texas Instruments standard programmer's interface for debugging. This book is divided into three parts:

- **Part I: Hands-On Information** is presented first so that you can start using your debugger the same day you receive it.
  - Chapter 1 lists the key features of the debugger, describes additional 'C6x software tools, tells you how to prepare a 'C6x program for debugging, and provides instructions and options for invoking the debugger.
  - Chapter 2 is a tutorial that introduces you to many of the debugger features.
- **Part II: Debugger Description** contains detailed information about using the debugger.

  The chapters in Part II detail the individual topics that are introduced in the tutorial. For example, Chapter 3 describes all of the debugger's windows and tells you how to move and size them; Chapter 4 describes everything you need to know about entering commands.

- **Part III: Reference Material** provides supplementary information.
  - Chapter 11 gives a complete summary of all the tasks introduced in Parts I and II. This includes a functional and an alphabetical summary of the debugger commands and a topical summary of function key actions.
  - Chapter 12 provides information about C expressions. The debugger commands are powerful because they accept C expressions as parameters; however, you can also use the debugger to debug assembly language programs. The information about C expressions aids assembly language programmers who are unfamiliar with C.
  - Part III also includes a glossary and an index.

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

- If you have used TI development tools or other debuggers before, you may want to:
  - Read the introductory material in Chapter 1.
  - Complete the tutorial in Chapter 2.
  - Read the alphabetical command reference in Chapter 11.
- If this is the first time that you have used a debugger or similar tool, you may want to:
  - Read the introductory material in Chapter 1.
  - Complete the tutorial in Chapter 2.
  - Read all of the chapters in Part II.

Notational Conventions

This document uses the following conventions.

- The TMS320C6x family of devices is referred to as ’C6x.
- The C source debugger has a very flexible command-entry system; there are usually a variety of ways to perform any specific action. For example, you may be able to perform the same action by typing in a command, using the mouse, or using function keys. This document uses three symbols to identify the methods that you can use to perform an action:

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

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

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

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

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

Command	Result displayed in the COMMAND window
whatis 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 font,** and parameters are in *italics*. Portions of a syntax that are in **bold face** should be entered as shown; portions of a syntax that are in *italics* describe the kind of information 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 specify the information within the brackets; you don't enter the brackets themselves. Here's an example of a command that has an optional parameter:

  run [*expression*]

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

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

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

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

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

Related Documentation From Texas Instruments

The following books describe the TMS320C6x DSPs and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C6x Assembly Language Tools User's Guide (literature number SPRU186) 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 'C6x generation of devices.

TMS320C6x Optimizing C Compiler User's Guide (literature number SPRU187) describes the 'C6x C compiler. This C compiler accepts ANSI standard C source code and produces assembly language source code for the 'C6x generation of devices. This book also describes the assembly optimizer, which helps you optimize your assembly code.

TMS320C6x Software Tools Getting Started Guide (literature number SPRU185) describes how to install the TMS320C6x assembly language tools, the C compiler, the simulator, and the C source debugger. Installation instructions for SunOS™, Solaris™, Windows™ 95, and Windows NT™ systems are given.

TMS320C62xx CPU and Instruction Set Reference Guide (literature number SPRU189) describes the 'C62xx CPU architecture, instruction set, pipeline, and interrupts for the TMS320C62xx digital signal processors.

TMS320C62xx Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C62xx digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port, serial ports, direct memory access (DMA), clocking and phase-locked loop (PLL), and the power-down modes.

TMS320C62xx Programmer's Guide (literature number SPRU198) describes ways to optimize C and assembly code and includes application program examples.

TMS320C62xx Technical Brief (literature number SPRU197) gives an introduction to the 'C62xx digital signal processor, development tools, and third-party support.

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

Trademarks

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/mirrors/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: comments@books.sc.ti.com

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

Contents

Part I: Hands-On Information

Part III: Reference Material

Figures

Tables

Chapter 1

Overview of a Code Development and Debugging System

The TMS320C6x C source debugger is an advanced programmer's interface that helps you to develop, test, and refine 'C6x C programs (compiled with the 'C6x optimizing ANSI C compiler) and assembly language programs. The debugger is the interface to the 'C6x simulator.

This chapter gives an overview of the programmer's interface, describes the 'C6x code development environment, and provides instructions and options for invoking the debugger.

Topic **Page**

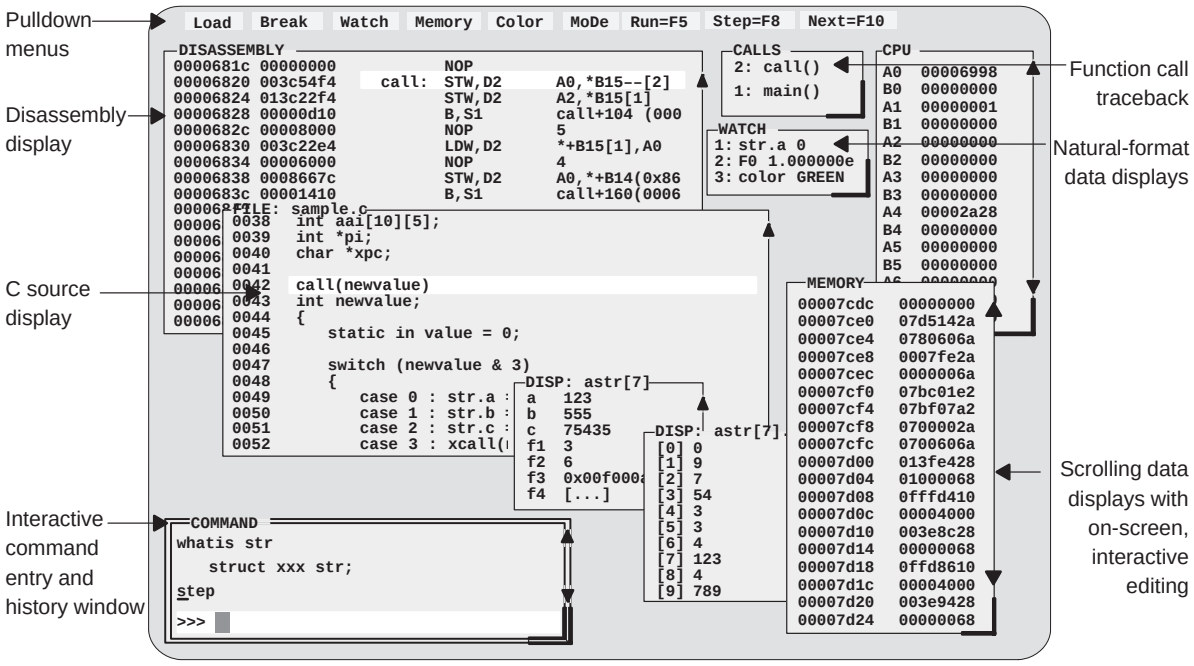
1.1 Description of the C Source Debugger

The ’C6x C source debugging interface improves productivity by allowing you to debug a program in the language it was written in. You can choose to debug your programs in C, assembly language, or both. And, unlike many other debuggers, the ’C6x debugger’s higher level features are available even when you’re debugging assembly language code.

The Texas Instruments advanced programmer’s interface is easy to learn and use. Its friendly window-, mouse-, and menu-oriented interface reduces learning time and eliminates the need to memorize complex commands. The debugger’s customizable displays and flexible command entry let you develop a debugging environment that suits your needs. A shortened learning curve and increased productivity reduce the software development cycle, so you’ll get to market faster.

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

Figure 1–1. The Basic Debugger Display

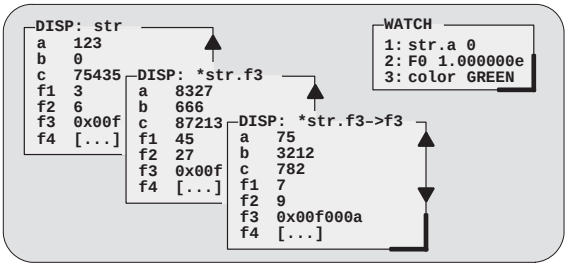


Key features of the debugger

- **Multilevel debugging**. The debugger allows you to debug both C and assembly language code. If you're debugging a C program, you can choose to view 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.

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

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



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

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

- **Powerful command set.** Unlike many other debugging systems, this debugger doesn't force you to learn a large, intricate command set. The 'C6x C source debugger supports a small but powerful command set that makes full use of C expressions. One debugger command performs actions that would take several commands in another system.

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


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

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

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

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

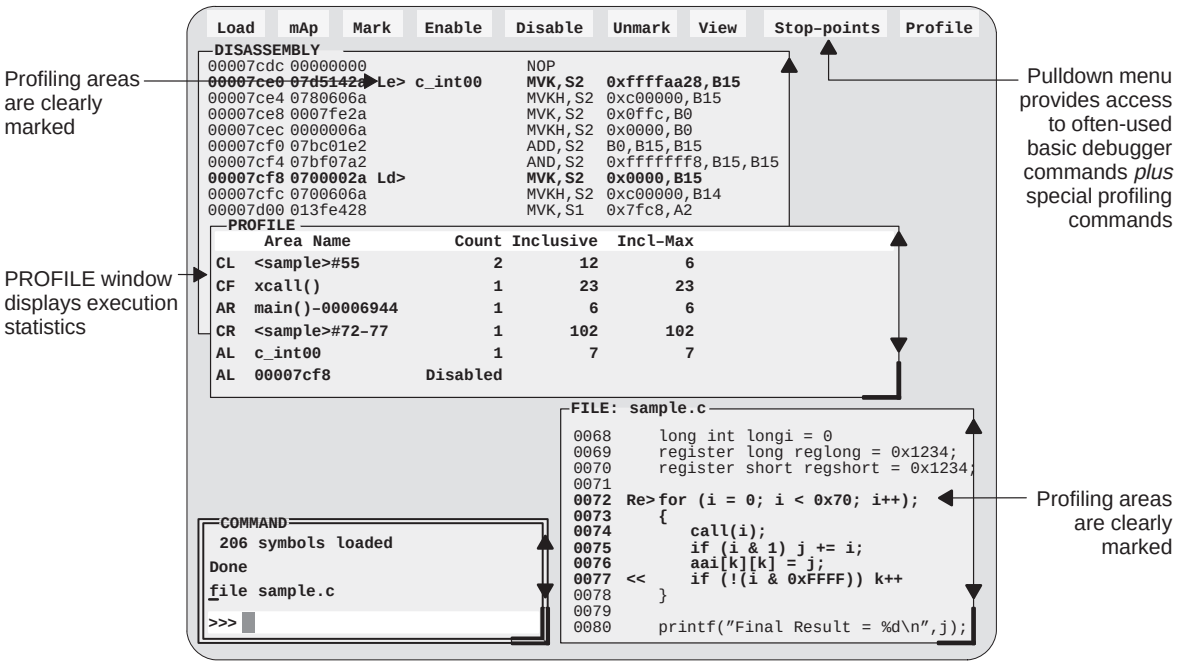
- **All the standard features you expect in a world-class debugger.** The debugger provides you with complete control over program execution with features like conditional execution and single-stepping (including single-stepping into or over function calls). You can set or clear a breakpoint with a click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

1.2 Description of the Profiling Environment

In addition to the basic debugging environment, a second environment—the *profiling environment*—is available. The profiling environment provides a method for collecting execution statistics about specific areas in your code. This gives you immediate feedback on your application's performance.

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

Figure 1–2. The Profiling-Environment Display



Key features of the profiling environment

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

- **More efficient code.** Within the profiling environment, you can quickly identify busy sections in your programs. This helps you to direct valuable development time toward 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're profiling, or display a selected subset of the areas.

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

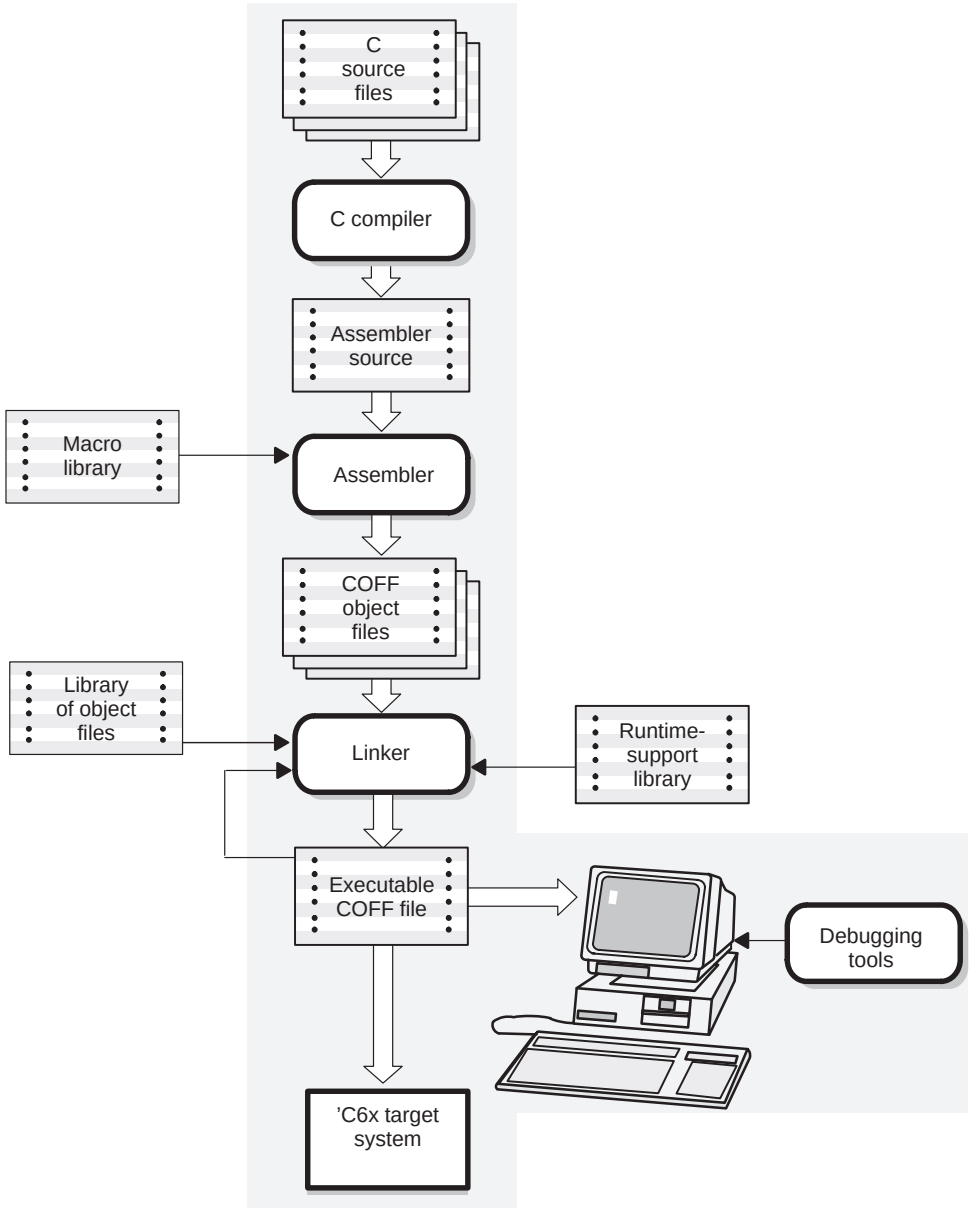
- **Disabled areas.** In addition to identifying areas that you can collect statistics on, you can also identify areas that you don't want to affect the statistics. This removes the timing impact from code such as a standard library function or a fully optimized portion of code.

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

1.3 Developing Code for the TMS320C6x

The ’C6x is well supported by a complete set of hardware and software development tools, including a C compiler, assembler, and linker. Figure 1–3 illustrates the ’C6x code development flow. The most common paths of software development are highlighted in grey; the other portions are optional.

Figure 1–3. TMS320C6x 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 1–3.

C compiler

The 'C6x optimizing ANSI **C compiler** is a full-featured optimizing compiler that translates standard ANSI C programs into 'C6x assembly language source. Key characteristics include:

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

assembler

The **assembler** translates ’C6x assembly language source files into machine language object files.

linker

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

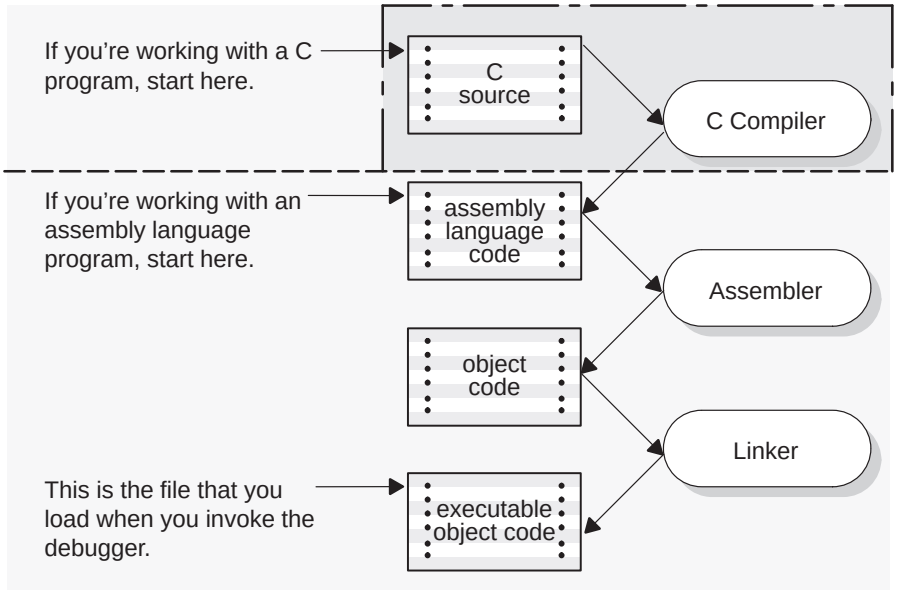
debugging tools

The main purpose of the development process is to produce a module that can be executed in a **’C6x target system.** You can use the debugger as an interface for the software simulator to refine and correct your code.

1.4 Preparing Your Program for Debugging

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

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



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

You can compile, assemble, and link a program by invoking the compiler, assembler, and linker in separate steps; or you can perform all three actions in a single step by using the cl60 shell program. The *TMS320C6x Assembly Language Tools User's Guide* and the *TMS320C6x Optimizing C Compiler User's Guide* contain complete instructions for invoking the tools individually and for using the shell program.

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

cl60 [**–***options*] **–g** [*filenames*] [**–z** [*link options*]]

cl60 invokes the compiler and assembler.

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

–g tells the C compiler to produce symbolic debugging information. When preparing a C program for debugging, you must use the –g option, or you won't be able to access symbolic debugging information (such as C labels, variables, etc.).

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

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

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

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

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

Extension	File type	The shell will...
.c	C source	Compile, assemble, and link the file
.asm	Assembly language source	Assemble and link the file
.s* (any extension that begins with s)	Assembly language source	Assemble and link the file
.o* (extension begins with o)	Object file	Link the file
none (.c assumed)	C source	Compile, assemble, and link the file

Note: The shell links files only if you specify the –z option.

1.5 Invoking the Debugger

Enter the following command on the command-line to invoke the standalone debugger:

sim6x [*filename*] [*–options*]

sim6x invokes the debugger.

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

–options supply the debugger with additional information.

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

Table 1–1. Summary of Debugger Options

Option	Brief Description
–b	Select a preset screen size (80 characters by 40 lines)
–bb	Select a slightly larger preset screen size (80 characters by 50 lines)
–c	Clear the .bss section
–d *machinename*	Display the debugger on different machine (X Window System™ only)
–i *pathname*	Identify additional directories
–min	Select the minimal debugging mode
–o	Enable C I/O
–profile	Enter the profiling environment
–s	Load the symbol table only
–t *filename*	Identify a new initialization file
–v	Load without the symbol table
–x	Ignore D_OPTIONS

Selecting the screen size (–b, –bb options)

By default, the debugger uses an 80-character-by-25-line screen.

When you run multiple debuggers, the default screen size is a good choice because you can more easily view multiple default-size debuggers on your screen. However, you can change the screen size by using one of the –b options, which provides a preset screen size, or by resizing the screen at run time. (Note that when you are running a standalone debugger, you can also change the screen size by using one of these methods.)

- **Using a preset screen size.** Use the –b or –bb option to select one of these preset screen sizes:
  - **–b** Screen size is 80 characters by 43 lines.
  - **–bb** Screen size is 80 characters by 50 lines.
- **Resizing the screen at run time.** You can resize the screen at run time by using your mouse to change the size of the operating-system window that contains the debugger. The maximum size of the debugger screen is 132 characters by 60 lines.

Clearing the .bss section (–c option)

The –c option clears the .bss section when the debugger loads code. You can use this option when you have C programs that use the RAM initialization model (specified with the –cr linker option).

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. For example, if you are running a debugger on a machine called opie and you want the debugger display to appear on a machine called barney, use the following command to invoke the debugger:

```
sim6x –d barney:0 ↵
```

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

Identifying additional directories (–i option)

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

sim6x –i *pathname*$_1$ **–i** *pathname*$_2$ **–i** *pathname*$_3$ **. . .**

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

Selecting the minimal debugging mode (–min option)

By default, the debugger automatically displays whatever code is currently running: assembly language or C. By default, the MEMORY, COMMAND, DISASSEMBLY, and CPU windows are displayed. You may also display other windows, such as the DISP and WATCH windows.

The debugger has a *minimal* debugging mode that displays the COMMAND, WATCH, and DISP windows only. The WATCH and DISP windows are displayed only if you cause them to display (by entering the WA or DISP commands). Minimal mode may be useful when you need to debug a memory problem.

To invoke the debugger and enter minimal mode, use the –min option:

sim6x –min . . .

For more information about the windows in the debugger interface, see Section 3.2, *Descriptions of the Different Kinds of Windows and Their Contents*.

Entering the profiling environment (–profile option)

The –profile option allows you to bring up the debugger in a profiling environment so that you can collect statistics about code execution. Note that only a subset of the basic debugger features is available in the profiling environment.

Loading the symbol table only (–s option)

If you supply a *filename* when you invoke the debugger, you can use the –s option to tell the debugger to load only the file's symbol table (without the file's object code). This option is useful in a debugging environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). Using this option is similar to loading a file by using the debugger's SLOAD command.

Identifying a new initialization file (–t option)

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

–t *filename*

Loading without the symbol table (–v option)

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

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

Ignoring D_OPTIONS (–x option)

The –x option tells the debugger to ignore any information supplied with the D_OPTIONS environment variable (described in the installation guide).

1.6 Exiting the Debugger

To exit the debugger, enter the following command from the COMMAND window:

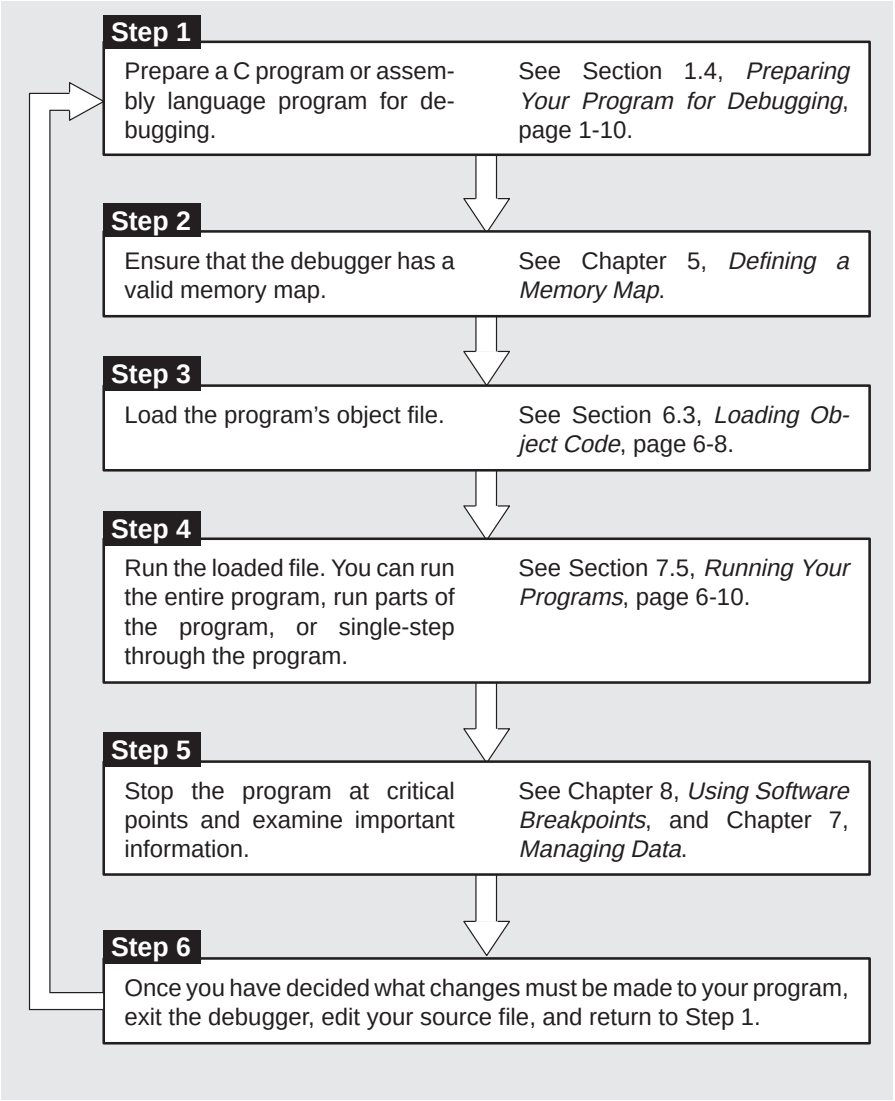
```
quit ↵
```

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

You can also exit the debugger by selecting the close option from the Windows menu bar.

1.7 Debugging Your Programs

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



Chapter 2

An Introductory Tutorial to the C Source Debugger

This chapter provides a step-by-step, hands-on demonstration of the 'C6x C source debugger's basic features. This is not the kind of tutorial that you can take home to read—it is effective only if you're sitting at your terminal, performing the lessons in the order that they're presented. The tutorial contains two sets of lessons (11 in the first, 13 in the second) and takes about 1 hour to complete.

Topic **Page**

How to use this tutorial

This tutorial contains three basic types of information:

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

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

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

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

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

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

Important! This tutorial assumes that you have correctly and completely installed your debugger (including invoking any files or operating-system commands as instructed in the installation guide).

A note about entering commands

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

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

An escape route (just in case)

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

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

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

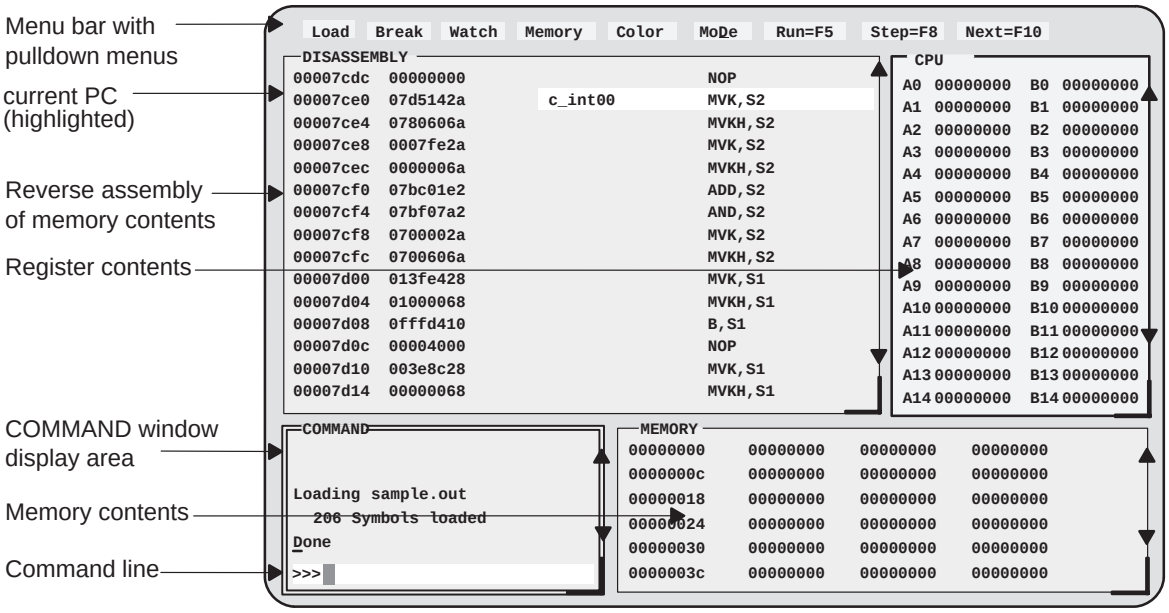
Included with the debugger is a demonstration program named *sample*. This lesson shows you how to invoke the debugger and load the sample program.

Invoke the debugger and load the sample program; enter:

```
sim6x sample.out ↵
```

Take a look at the display. . .

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



- ❑ If you **don't** see a display, then your debugger may not be installed properly. Go back through the installation instructions and be sure that you followed each step correctly; then reinvoke the debugger.

- ❑ If you **do** see a display, *check the first few lines of the DISASSEMBLY window*. If these lines aren't the same—if, for example, they show ADD instructions or say *Invalid address*—then enter the following commands on the debugger command line. (Just type; you don't have to worry about where the cursor is.)

  1) Reset the 'C6x processor:

     `reset` ↵

  2) Load the sample program again:

     `load sample.out` ↵

What's in the DISASSEMBLY window?

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

This tutorial step demonstrates that the code shown in the DISASSEMBLY window corresponds to memory contents. Initially, memory is displayed starting at address 0; if you look at the first line of the DISASSEMBLY window, you'll see that its display starts at address 0x7cdc.

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

```
mem 0x7cdc
```

Notice that the addresses in the first column of the DISASSEMBLY window correspond to the addresses in the first column of the MEMORY window; the values in the second column of the DISASSEMBLY window correspond to the memory contents displayed in the second, third, and fourth columns of the MEMORY window.

Select the active window

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

Make the CPU window the active window:

```
win CPU
```

lesson continues on the next page →

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

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

Try This: Press the F6 key to "cycle" through the windows in the display, making each one active in turn.

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

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

2) Click the left mouse button.

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

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

  Press ESC to get out of this.

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

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

Resize the active window

This lesson shows you how to resize the active window.

Important! Be sure the CPU window is still active.

Make the CPU window as small as possible:

`size 4,3` ↵

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

Make the CPU window larger:

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

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

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

1) Point to the lower right corner of the CPU window.

2) Press the left mouse button but don't release it; move the mouse while you're holding in the button. This resizes the window.

3) Release the mouse button when the window reaches the desired size.

Zoom the active window

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

Important! Be sure the CPU window is still active.

Make the active window as large as possible:

`zoom` ↵

The window should now be as large as possible, taking up the entire display (except for the menu bar) and hiding all the other windows. Notice that you can now see additional registers that were not displayed before. For example, you can now see the current value of the PC register.

"Unzoom" or return the window to its previous size by entering the ZOOM command again:

`zoom` ↵ *The ZOOM command will be recognized, even though the COMMAND window is hidden by the CPU window.*

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

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

Zoom the DISASSEMBLY window:

1) Point to the upper left corner of the DISASSEMBLY window.

2) Click the left mouse button. This makes the DISASSEMBLY window active and zooms it at the same time.

Now, you can see the complete disassembly code of each line in the sample program. Return the window to its previous size by repeating steps 1 and 2.

Move the active window

This lesson shows you how to move the active window.

Important! Be sure the CPU window is still active.

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

`move 0,1` ↵ *The debugger doesn't let you move the window to the very top—that would hide the menu bar*

The MOVE command's first parameter identifies the window's new X position on the screen. The second parameter identifies the window's new Y position on the screen. The maximum X and Y positions depend on which screen-size option you used when you invoked the debugger and on the position of the window before you tried to move it.

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

`move` ↵
→→→→ *Press → until the CPU window is back where it was (it may seem as if only the border is moving—this is normal)*
ESC *Press ESC when you finish moving the window*

You can use ↑ to move the window up, ↓ to move the window down, and ← to move the window left.

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

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

2) Press the left mouse button, but don't release the button; move the mouse while you're holding in the button.

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

Scroll through a window's contents

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

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

Scroll through the contents of the DISASSEMBLY window:

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

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

Make the MEMORY window the active window:

`win MEMORY` ↵

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

↓ ↑ PAGE DOWN PAGE UP

These keys don't work the same for all windows; Section 11.5, *Summary of Special Keys*, on page 11-50 summarizes the functions of all the special keys and key sequences and how they affect different windows.

Display the C source version of the sample file

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

Display the contents of a C source file:

```
file sample.c ↵
```

This opens a FILE window that displays the contents of the file sample.c (sample.c was one of the files that contributed to making the sample object file). You can always tell which file you're displaying by looking at the label in the FILE window. Right now, the label says FILE: sample.c. If you can't see the label, press F6 until the FILE window becomes the active window.

The CALLS window is displayed also. The CALLS window tracks the C functions as they are called. Right now, the CALLS window lists **UNKNOWN as the first function, because it is waiting for a function to be called.

Execute some code

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

Important! You will be looking at the contents of the PC register in this lesson. If you cannot see the contents of the PC register in your CPU window, either resize your CPU window or scroll down until you can see the PC value.

Execute a portion of the sample program:

`go main` ↵ *The label in the COMMAND window changes to COMMAND [RUNNING...] to indicate that your program is executing.*

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

- The current PC is highlighted in both the DISASSEMBLY and FILE windows.
- The object codes of the first several statements in the DISASSEMBLY window have changed color because these statements are associated with the current C statement (which is highlighted in the FILE window).
- The CALLS window, which tracks functions as they're called, now lists main().
- The color for the value of the PC in the CPU window has changed because the PC's value changed during program execution.

Become familiar with the four debugging modes

The debugger has four basic debugging modes:

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

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

The following steps show you how to switch debugging modes.

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

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

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

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

4) Release the button.

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

Switch to auto mode:

1) Press [ALT][D]. This displays and freezes the MoDe menu.

2) Now select C(auto). To do so, choose one of these methods:

   - Press the arrow keys to move up/down through the menu; when C(auto) is highlighted, press [↵].
   - Type **C**.
   - Point the mouse cursor at C(auto), then click the left mouse button.

You should be in auto mode now, and you should see the FILE window. The statement that defines the main() label should still be highlighted. You should not see the DISASSEMBLY window, because the processor is in the C portion of your program. Auto mode automatically switches between an assembly and a C display, depending on where you are in your program. Here's a demonstration of that:

Restart your program, so it is at a point that executes assembly language code:

```
restart [↵]
```

lesson continues on the next page →

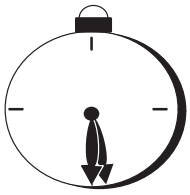
You're still in auto mode, but you should now see the DISASSEMBLY window. The current PC should be at the statement that defines the c_int00 label (the first statement in the sample program).

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

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

Switch back to mixed mode before continuing:

```
mix ↵
```



Halfway Point

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

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

Open another text file, then redisplay a C source file

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

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

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

```
file init.cmd
```

This replaces sample.c in the FILE window with your init.cmd file.

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

Redisplay another C source file (sample.c):

```
func call
```

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

Use the basic RUN command

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

Run your entire program:

`run` ↵ *The label in the COMMAND window changes to COMMAND [RUNNING...] to indicate that your program is executing.*

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

This isn’t very exciting; halt program execution:

ESC

Set some breakpoints

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

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

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

`file sample.c` ↵

Set a software breakpoint and run your program:

1) Scroll to line 42 in the FILE window (the call(newvalue) statement) and set a breakpoint at that line:

   a) Point the mouse cursor at the statement on line 42.

   b) Click the left mouse button. *Notice that BP> (for breakpoint) appears at the beginning of the line and that the line is highlighted.*

2) Reset the program entry point:

   `restart` ↵

3) Enter the run command:

   `run` ↵

Once again, you should see that some statements are highlighted in the CPU window, showing that they were changed by program execution. But this time, you know that the changes were caused by code from the beginning of the program to line 42 in the FILE window.

Clear the breakpoint:

1) Point the mouse cursor at the statement on line 42. (It should still be highlighted from setting the breakpoint.)

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

Watch some values and single-step through code

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

Set up for the single-step example:

```
restart ↵
go main ↵
```

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

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

Open a WATCH window and change to mixed mode:

```
wa b15, Stack Pointer, x ↵
wa pc,, x ↵
wa *0x6820, Call ↵
wa i ↵
mix ↵
```

If the WATCH window isn't wide enough to display the value of the stack pointer, resize the window.

You may have noticed that the WA (watch add) command has three parameters. The first parameter is the item that you're watching. The second parameter is an optional label. The third parameter is the format for the data display. For example, you displayed the contents of the B15 register with the label Stack Pointer in hexadecimal format. (You specified hexadecimal with an x in the third parameter.) You also displayed the contents of the PC register without a label in hexadecimal format. (You specified that you wanted to use the expression instead of a label by inserting an extra comma.)

Now try out the single-step commands. **Hint:** Watch the PC in the FILE and DISASSEMBLY windows; watch the values that you set up in the WATCH window.

Single-step through the sample program:

```
step 20 ↵
```

Try This: Notice that the step command single-stepped each assembly language statement (in fact, you single-stepped through 20 assembly language statements). The debugger supports additional single-step commands that have a slightly different flavor.

- For example, if you enter:

  ```
  cstep 20 ↵
  ```

  you'll single-step 20 *C statements,* not assembly language statements (notice how the PC "jumps" in the DISASSEMBLY window).

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

  ```
  restart ↵
  go main ↵
  ```

  Now enter the NEXT command, as shown below. You'll be single-stepping 20 assembly language statements.

  ```
  next 20 ↵
  ```

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

Run code conditionally

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

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

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

```
wd 3 ↵
wd 1 ↵
wd 1 ↵
```

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

Set up for the conditional run examples:

1) Set software breakpoints at lines 72 and 74.

2) Reset the program entry point:

```
restart ↵
```

3) Run the first part of the program:

```
go main ↵
```

Now initiate the conditional run:

```
run i<10 ↵
```

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

When the conditional run completes, close the WATCH window.

Close the WATCH window:

wr ↵

WHATIS that?

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

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

```
whatis genum ↵
      enum yyy genum;            genum is an enumerated type
whatis tiny6 ↵
      struct {                                tiny6 is a structure
          int u;
          int v;
          int x;
          int y;
          int z;
      } tiny6;
whatis call ↵
      int call();          call is a function that returns an integer
whatis s ↵
      short s;                          s is a short unsigned integer
```

Clear the COMMAND window display area

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

Clear the COMMAND window display area:

`cls` ↵

Display the contents of an aggregate data type

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

Show a structure in a DISP window:

`disp small` ↵

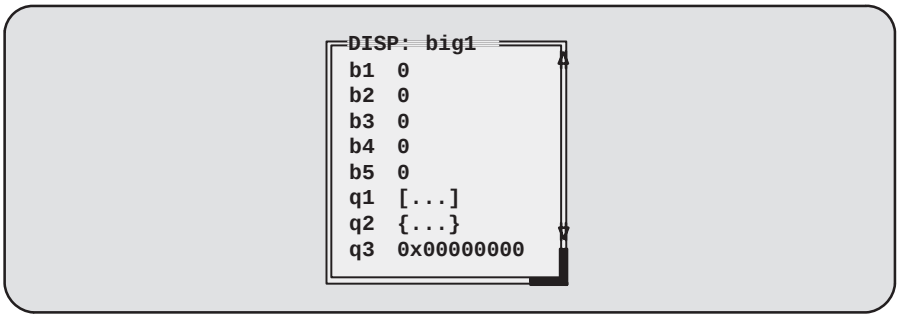
Close the DISP window:

F4

Show another structure in a DISP window:

`disp big1` ↵

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



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

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

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

Display what q3 is pointing to:

1) Point at the address displayed next to the q3 label in big1's display.

2) Click the left mouse button.

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

lesson continues on the next page →

Display array q1 in another DISP window:

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

2) Click the left mouse button.

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

Try This: Display structure q2 in another DISP window.

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

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

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

F9 4) Now press F9 .

Close all of the DISP windows:

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

2) Press F4.

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

Display data in another format

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

One way to display data in another format is through casting (which is part of the C language). In the expression below, the *(float *) portion of the expression tells the debugger to treat address 0x100 as type float (exponential floating-point format).

Display memory contents in floating-point format:

```
disp *(float *)0x100 ↵
```

This opens a DISP window to show memory contents in an array format. The array member identifiers don't necessarily correspond to actual addresses—they're relative to the first address you request with the DISP command. In this case, the item displayed as item [0] is the contents of address 0x0100—*it isn't memory location 0*. Note that you can scroll through the memory displayed in the DISP window; item [1] is at 0x0101, and item [–1] is at 0x0ffe.

lesson continues on the next page →

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

Change display formats according to data types by using the SETF (set format) command:

1) For comparison, watch the following variables. Their C data types are listed on the right.

```
wa i ↵                    Type int
wa ff ↵                   Type float
wa dd ↵                   Type double
```

2) You can list all the data types and their current display formats:

```
setf ↵
```

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

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

4) List the data types to display formats again. You might want to zoom the COMMAND window to compare the new listing of the type formats to the listing that you saw in step 2:

```
setf ↵
```

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

```
wa i, NEWi ↵
wa ff, NEWff ↵
wa dd, NEWdd ↵
```

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

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

```
setf * ↵
```

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

Use display formats with the ? and WA commands:

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

   `? small.ra[1],c` ↵

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

   `wa str.a,,o` ↵

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

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

Change some values

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

Important! Make sure no other windows are obscuring your view of the MEMORY window.

Change a value in memory:

1) Display memory beginning with address 0x0100:

   `mem 0x100` ↵

↖ 2) Point to the contents of memory location 0x0100. (The contents of memory location 0x0100 are in the second column of the MEMORY window.)

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

4) Type 00000000.

5) Press ↵ to enter the new value.

6) Press ESC to conclude editing.

Try This: Here's another method for editing data.

1) Make the CPU window the active window:

   `win CPU` ↵

↑↓ →← 2) Press the arrow keys until the field cursor (_) points to the contents of register B15.

F9 3) Press F9 to highlight the contents of register B15.

4) Type ffff0000.

↵ 5) Press ↵ to enter the new value.

ESC 6) Press ESC to conclude editing.

Define a memory map

You can set up a memory map to tell the debugger which areas of memory it can and can't access. This is called *memory mapping*. When you invoked the debugger for this tutorial, the debugger automatically read a default memory map from the initialization batch file included in the sim6x directory. For the purposes of the sample program, that's fine (which is why this lesson was saved for the end).

View the default memory map settings:

`ml` ↵

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

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

Change the memory map:

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

   `md 0x0080 0000` ↵

   This deletes the block of memory beginning at address 0x0080 0000.

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

   `ma 0x0080 0000,0x20,ROM` ↵
   `ma 0x0080 0100,0x7f,RAM` ↵

Define your own command string

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

This lesson shows you how you can define an alias to set up a memory map, defining the same map that was defined in the previous lesson.

Define an alias for setting up the memory map:

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

```
alias mymap,"mr;ma 0x0080 0000,0x20,ROM;
   ma 0x0080 0100,0x7f,RAM;ml" ↵
```

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

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

```
mymap ↵
```

This is equivalent to entering the following four commands:

```
mr
ma 0x0080 0000,0x20,ROM
ma 0x0080 0100,0x7f,RAM
ml
```

Close the debugger

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

Close the debugger and return to the operating system:

```
quit ↵
```

Chapter 3

The Debugger Display

The ’C6x C source debugger has a window-oriented display. This chapter shows what windows look like and describes the basic types of windows that you’ll use.

Topic **Page**

3.1 Debugging Modes and Default Displays

The debugger has four debugging modes:

- Auto
- Assembly
- Mixed
- Minimal

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

- When the debugger is running assembly language code, you'll see an assembly display similar to the one in Figure 3–1. The DISASSEMBLY window displays the reverse assembly of memory contents.
- When the debugger is running C code, you'll see a C display similar to the one in Figure 3–2. (This assumes that the debugger can find your C source file to display in the FILE window. If the debugger can't find your source, it switches to mixed mode.)

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

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

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

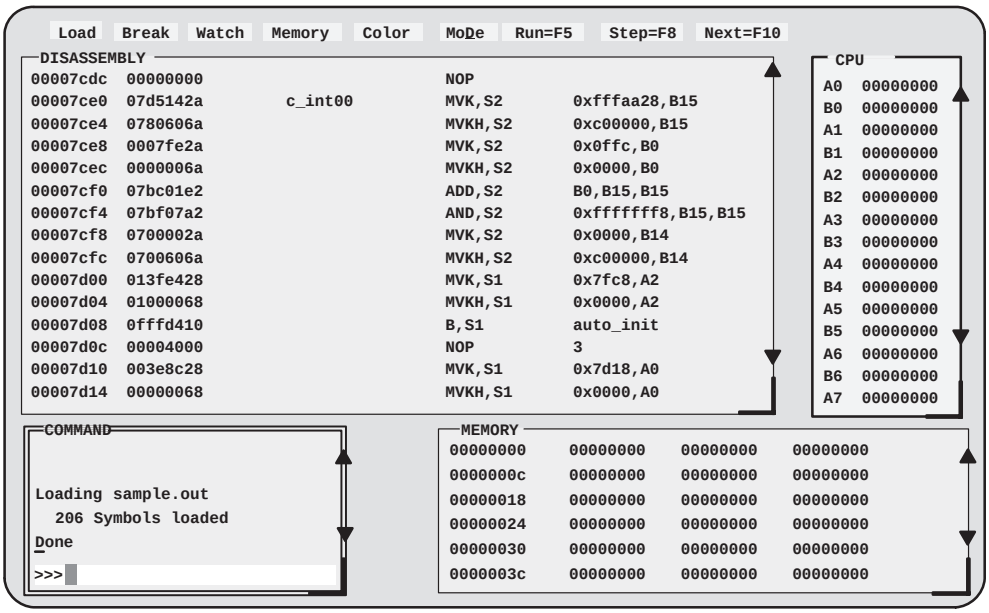
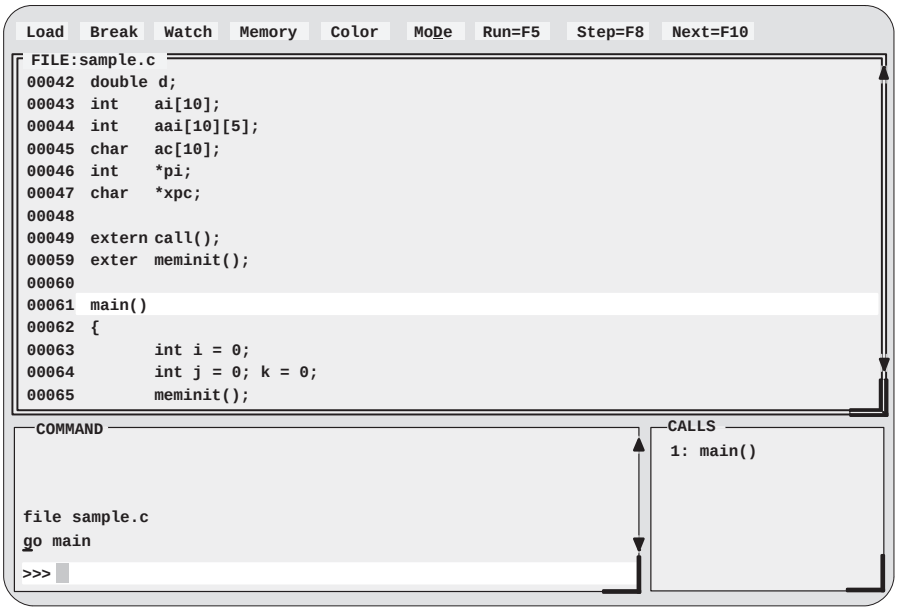


Figure 3–2. Typical C Display (for Auto Mode Only)



Assembly mode

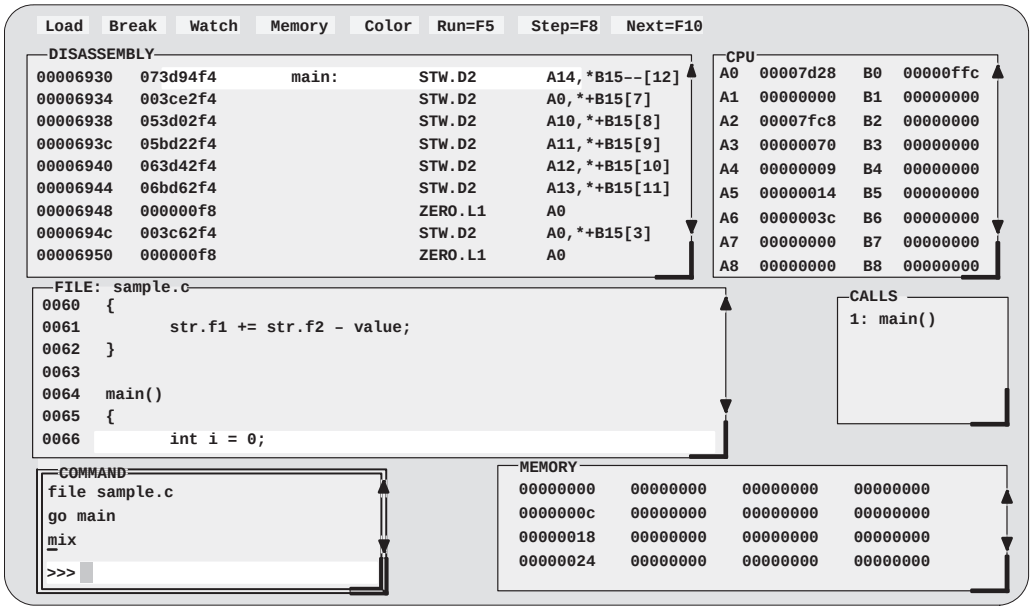
Assembly mode is for viewing assembly language programs only. In this mode, you'll see a display similar to the one shown in Figure 3–1. When you're in assembly mode, you'll always see the assembly display, regardless of whether C or assembly language is currently running.

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

Mixed mode

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

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



In mixed mode, the debugger displays all windows that can be displayed in auto and assembly modes, regardless of whether you're currently running assembly language or C code. This is useful for finding bugs in C programs that exploit specific architectural features of the 'C6x.

Minimal mode

Minimal mode allows you to query the target system without displaying any additional information. You can display the contents of CPU registers, memory addresses, or symbols within the COMMAND window by using the WA, DISP, and ?/EVAL commands (described on page 7-3). You can use any of the standard debugger commands in the COMMAND window. If you use the C, ASM, or MIX commands, the debugging mode changes to the auto, assembly, or mixed mode, respectively. To return to minimal mode, use the MINIMAL command.

Restrictions associated with debugging modes

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

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

dasm	func	mem
calls	file	disp

3.2 Descriptions of the Different Kinds of Windows and Their Contents

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

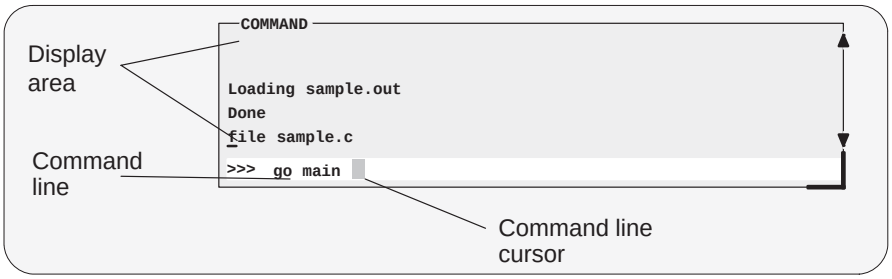
Every window is identified by a name in its upper left corner. Each type of window serves a specific purpose and has unique characteristics. There are eight different windows, divided into these general categories:

- The COMMAND window provides an area for typing in commands and for displaying various types of information, such as progress messages, error messages, or command output.
- Code-display windows display assembly language or C code. There are three code-display windows:
  - The DISASSEMBLY window displays the disassembly (assembly language version) of memory contents.
  - The FILE window displays any text file that you want to display; its main purpose, however, is to display C source code.
  - The CALLS window identifies the current function and previous function calls (when C code is running).
- The PROFILE window displays statistics about code execution.
- Data-display windows are for observing and modifying various types of data. There are four data-display windows:
  - A MEMORY window displays the contents of a range of memory. You can display multiple MEMORY windows at one time.
  - The CPU window displays the contents of ’C6x registers.
  - A DISP window displays the contents of an aggregate type such as an array or structure, showing the values of the individual members. You can display up to 120 DISP windows at one time.
  - A WATCH window displays selected data such as variables, specific registers, or memory locations. You can display multiple WATCH windows at one time.

You can move or resize any of these windows; you can also edit any value in a data-display window. Before you can perform any of these actions, however, you must select the window you want to move, resize, or edit and make it the *active window*. For more information about making a window active, see Section 3.4, *The Active Window*.

The remainder of this section describes the individual windows.

COMMAND window



Purpose
- ❑ Provides an area for entering commands
- ❑ Provides an area for echoing commands and displaying command output, errors, and messages

Editable? Command line is editable; command output isn't

Modes All modes

Created Automatically

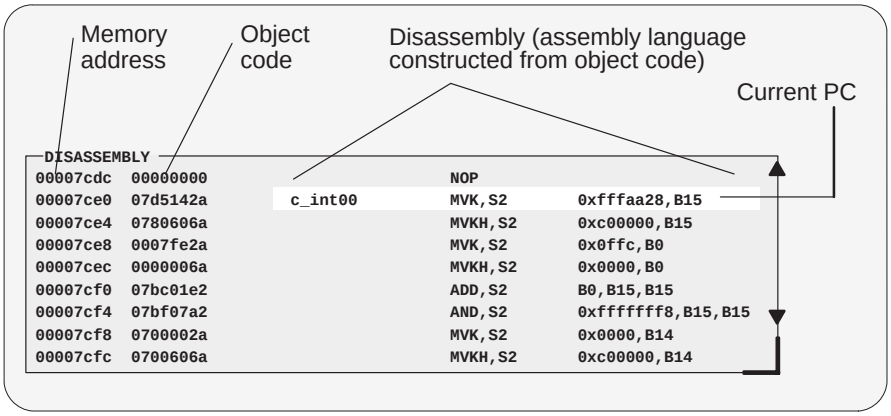
Affected by
- ❑ All commands entered on the command line
- ❑ All commands that display output in the display area
- ❑ Any condition or input that creates an error

The COMMAND window has two parts:

- ❑ *Command line.* This is where you enter commands. When you want to enter a command, just type—no matter which window is active. The debugger keeps a list of the last 50 commands that you entered. You can select and reenter commands from the list without retyping them. (For more information, see *Using the command history,* page 4-5.)
- ❑ *Display area.* This area of the COMMAND window echoes the command that you entered, shows any output from the command, and displays debugger messages.

For more information about the COMMAND window and entering commands, refer to Chapter 4, *Entering and Using Commands*.

DISASSEMBLY window



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

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

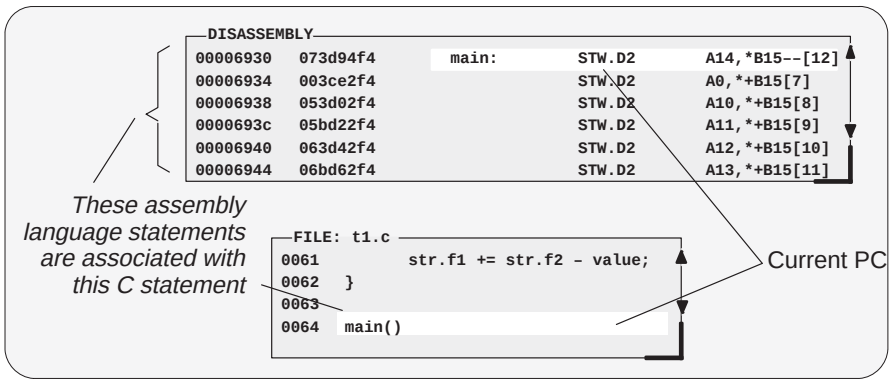
Modes Auto (assembly display only), assembly, and mixed

Created Automatically

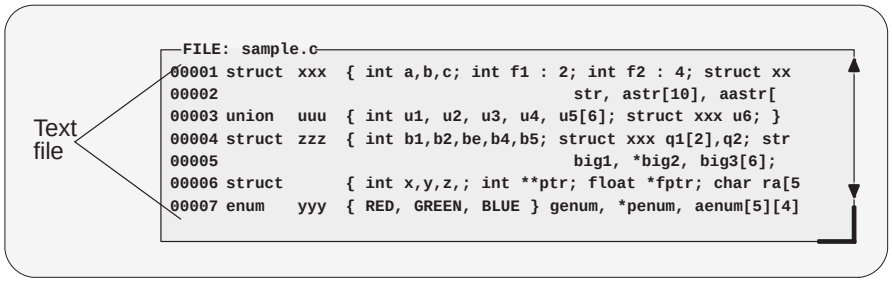
Affected by
- ❑ DASM and ADDR commands
- ❑ Breakpoint and run commands

Within the DISASSEMBLY window, the debugger highlights:

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



FILE window



Purpose Shows any text file you want to display

Editable? No; if the FILE window displays C code, pressing the edit key (F9) or the left mouse button sets a software breakpoint on a C statement

Modes Auto (C display only) and mixed

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

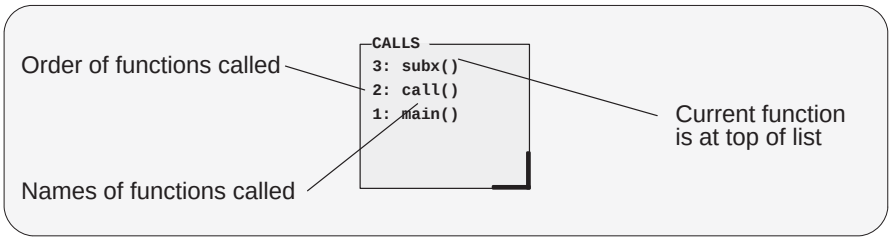
Affected by
- ❑ FILE, FUNC, and ADDR commands
- ❑ Breakpoint and run commands

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

Within the FILE window, the debugger highlights:

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

CALLS window



Purpose Lists the function you're in, its caller, and its caller, etc., as long as each function is a C function

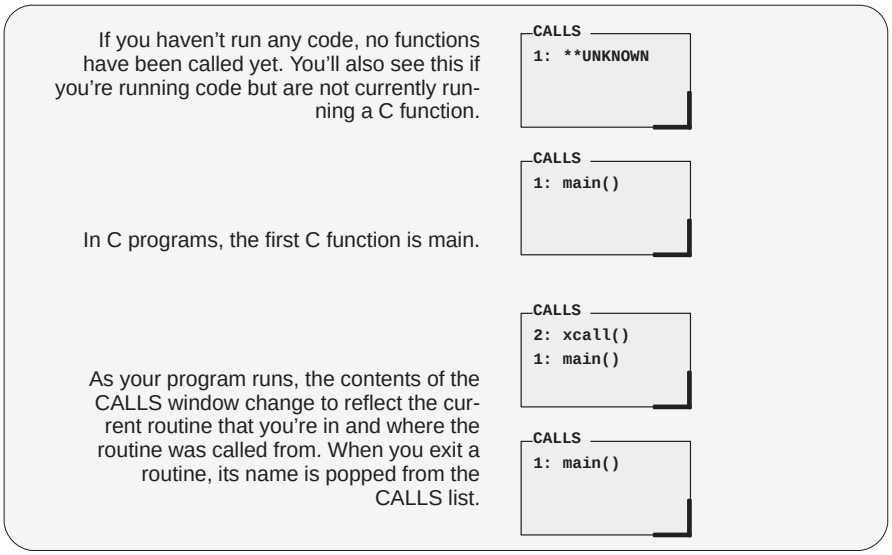
Editable? No; pressing the edit key (F9) or the left mouse button changes the FILE display to show the source associated with the called function

Modes Auto (C display only) and mixed

Created
- ❑ Automatically when you're displaying C code
- ❑ With the CALLS command if you closed the CALLS window

Affected by Run and single-step commands

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



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

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

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

1) Make the CALLS window the active window (see Section 3.4 on page 3-21).

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

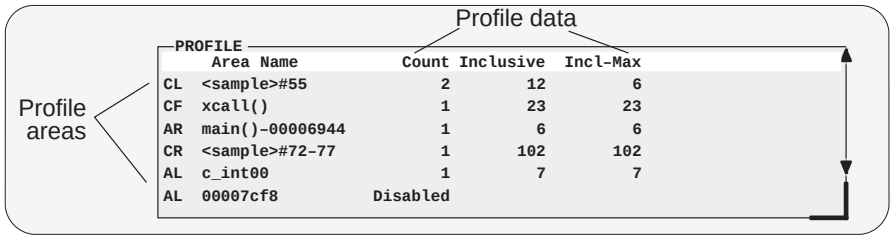
3) Press F9. This displays the selected function in the FILE window.

You can close and reopen the CALLS window.

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

  calls

PROFILE window



Purpose Displays statistics collected during a profiling session

Editable? No

Modes Auto

Created By invoking the debugger with the –profile option

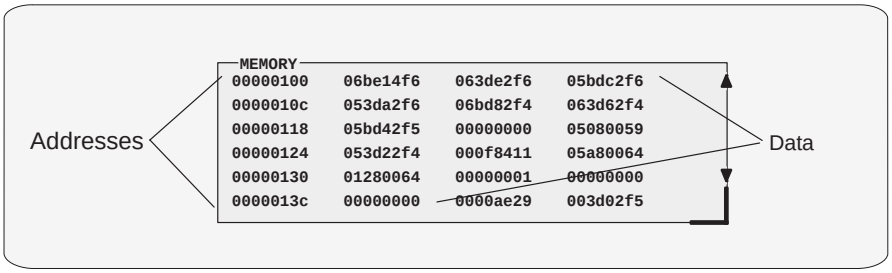
Affected by
- The PF and PQ commands
- Any commands on the View menu
- Clicking in the header area of the window

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

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

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

MEMORY windows



Purpose	Displays the contents of memory
Editable?	Yes—you can edit the data (but not the addresses)
Modes	Auto (assembly display only), assembly, and mixed
Created	❑ Automatically (the default MEMORY window only) ❑ With the MEM command and a unique *window name*
Affected by	MEM command

A MEMORY window has two parts:

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

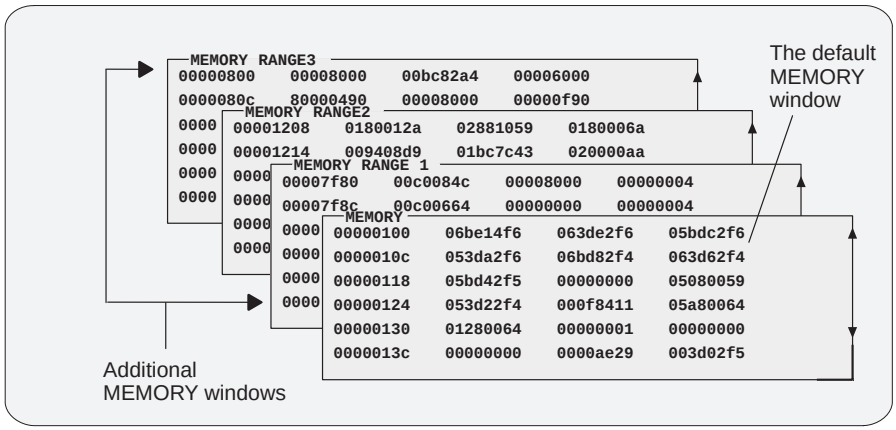
- *Data.* The remaining columns display the values at the listed addresses. The data is shown in hexadecimal format as 32-bit words. You can display more data by making the window wider and/or longer.

  The MEMORY window above has three columns of data, and each new address is incremented by 12 (0xC). Although the window shows three columns of data, there is still only one column of addresses; the first value is at address 0x0000 0100, the second at address 0x0000 0104, etc.; the fourth value (first value in the second row) is at address 0x0000 010C, the fifth at address 0x0000 0110, etc.

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

The debugger opens one MEMORY window by default. You can open any number of additional MEMORY windows to display different ranges of memory. Refer to Figure 3–4.

Figure 3–4. Default and Additional MEMORY Windows



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

❑ **Opening an additional MEMORY window**

To open an additional MEMORY window, enter the MEM command with a unique *window name*:

mem *address***,** [*display format*] **,** *window name*

For example, if you want to open a new MEMORY window starting at address 0x8000 named RANGE1, enter:

`mem 0x8000,,RANGE1` ↵

This displays a new window, labeled MEMORY RANGE1, showing the contents of memory starting at the address 0x8000.

❑ **Displaying a new memory range in a MEMORY window**

You can use the MEM command to display a different memory range in a window:

mem *address***,** [*display format*] **,** *window name*

The debugger displays the contents of memory at *address* in the first data position in your MEMORY window. The end of the range is defined by the size of the window.

The *window name* parameter is optional if you are displaying a different memory range in the default MEMORY window. Use the *window name* parameter when you want display a new memory range in one of the additional MEMORY windows.

The *display format* parameter for the MEM command is optional. When used, the data is displayed in the selected format as shown in Table 7–1 on page 7-17.

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

❑ **Closing a MEMORY window**

Closing a window is a two-step process:

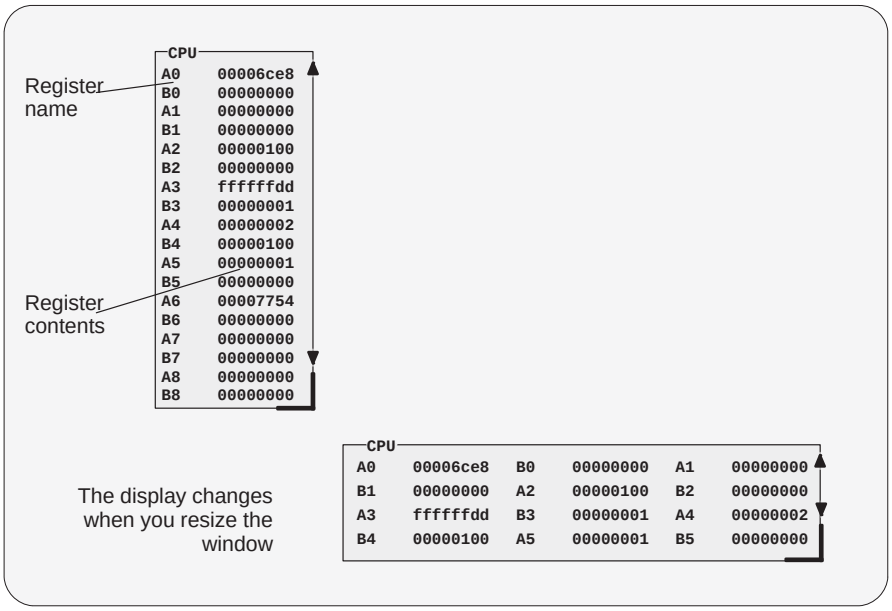
1) Make the appropriate MEMORY window the active window (see Section 3.4, on page 3-21).

2) Press F4.

❑ **Reopening a MEMORY window**

To reopen an additional MEMORY window after you've closed it, enter the MEM command with a unique window name. To reopen the default MEMORY window, use the MEM command with no window name.

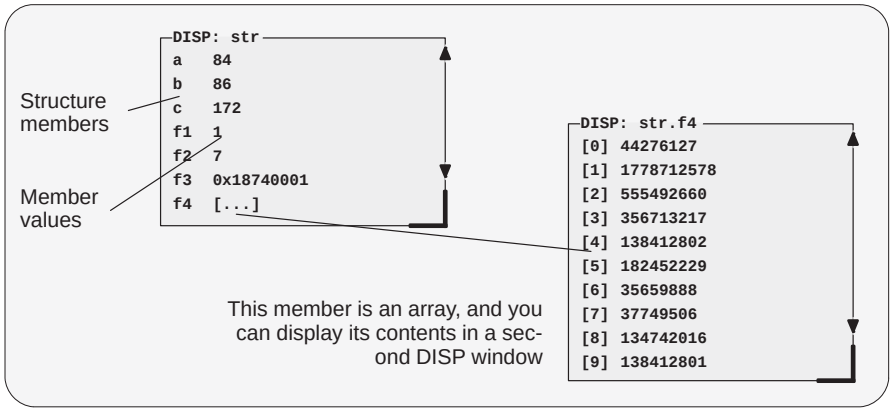
CPU window



Purpose	Shows the contents of the ’C6x registers
Editable?	Yes—you can edit the value of any displayed register
Modes	Auto (assembly display only), assembly, and mixed
Created	Automatically
Affected by	Data-management commands

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

DISP windows



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

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

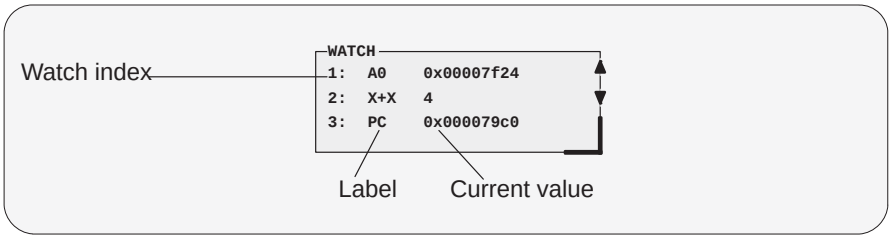
disp *expression*

Data is displayed in its natural format:

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

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

WATCH windows



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

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

To display the values of expressions, variables, or registers, use the WA command; the syntax is:

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

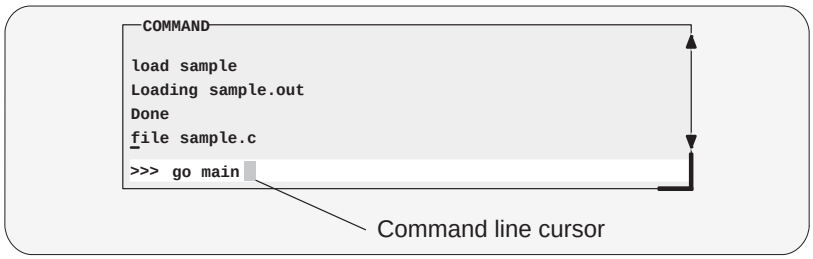
- WA adds *expression* to the WATCH window. (If there's no WATCH window, then WA also opens a WATCH window.)
- The *label* parameter is optional. When used, it provides a label for the watched entry. If you don't use a *label*, the debugger displays the *expression* in the label field.
- The *display format* parameter is optional. When used, the data is displayed in the selected format as shown in Table 7–1 on page 7-17.
- If you omit the *window name* parameter, the debugger displays the expression in the default WATCH window (labeled WATCH). 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.

To delete individual entries from a WATCH window, use the WD command with the appropriate *window name*. To delete all entries at once and close a WATCH window, use the WR command with the appropriate *window name*. Note that you don't need to specify a *window name* if you are deleting items from the default WATCH window.

3.3 Cursors

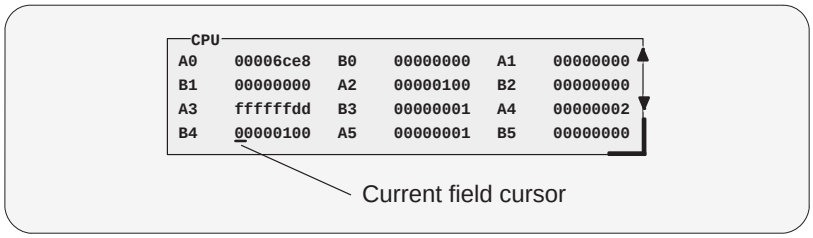
The debugger display has three types of cursors:

- The *command-line cursor* is a block-shaped cursor that identifies the current character position on the command line. When the COMMAND window is active (see Section 3.4, *The Active Window*), arrow keys affect the position of this cursor.



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

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



3.4 The Active Window

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

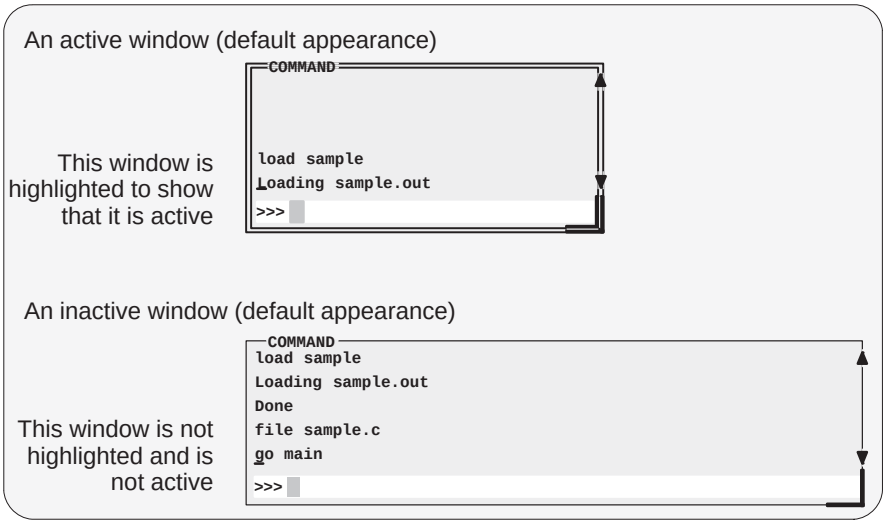
You can move, resize, zoom, or close *only one window at a time*; thus, only one window at a time can be the *active window*. Whether or not a window is active doesn't affect the debugger's ability to update information in a window—it affects only your ability to manipulate a window.

Identifying the active window

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

You can alter the active window's border style and colors if you wish; Figure 3–5 illustrates the default appearance of an active window and an inactive window.

Figure 3–5. Default Appearance of an Active and an Inactive Window



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

Selecting the active window

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

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

2) Click the left mouse button.

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

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

  Press ESC *to get out of this.*

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

  Press the button again to clear the breakpoint.

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

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

win *WINDOW NAME*

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

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

```
win  DISASSEMBLY  ↵
or   win  DISA    ↵
```

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

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

3.5 Manipulating a Window

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

Note:

You can resize or move any window, but first the window must be *active*. For information about selecting the active window, see Section 3.4 on page 3-21.

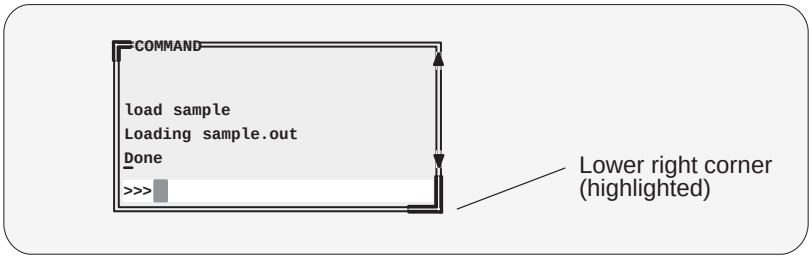
Resizing a window

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

There are two basic ways to resize a window:

- By using the mouse
- By using the SIZE command

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



2) Grab the highlighted corner by pressing one of the mouse buttons; while pressing the button, move the mouse in any direction. This resizes the window.

3) Release the mouse button when the window reaches the desired size.

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

size [*width*, *length*]

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

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

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

SIZE, method 1: Use the *width* and *length* parameters. Valid values for the width and length depend on the screen size and the window position on the screen. If the window is in the upper left corner of the screen, the maximum size of the window is the same as the screen size minus one line. (The extra line is needed for the menu bar.) For example, if the screen size is 80 characters by 25 lines, the largest window size is 80 characters by 24 lines.

If a window is in the middle of the display, you can't size it to the maximum height and width—you can size it only to the right and bottom screen borders. The easiest way to make a window as large as possible is to zoom it, as described on page 3-26.

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

```
win  CALLS ↵
size 8, 20 ↵
```

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

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

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

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

```
win  CPU ↵
size ↵
↓ ↓ ↓     ← ←     ESC
```

Zooming a window

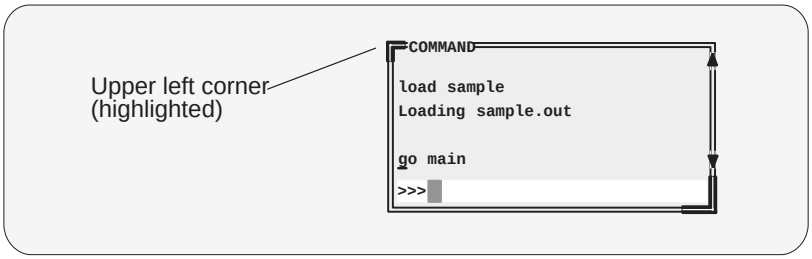
Another way to resize the active window is to zoom it. Zooming a window makes it as large as possible, so that it takes up the entire display (except for the menu bar) and hides all the other windows. Unlike the SIZE command, zooming is not affected by the window's position in the display.

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

There are two basic ways to zoom or unzoom a window:

- By using the mouse
- By using the ZOOM command

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



2) Click the left mouse button.

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

zoom

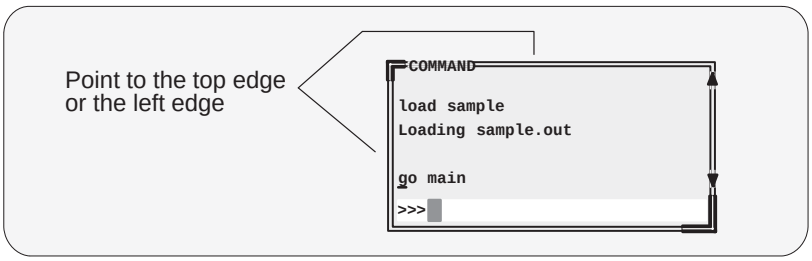
Moving a window

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

There are two ways to move a window:

- By using the mouse
- By using the MOVE command

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



2) Press the left mouse button but don't release it; now move the mouse in any direction.

3) Release the mouse button when the window is in the desired position.

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

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

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

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

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

MOVE, method 1: Use the *X position* and *Y position* parameters. You can move a window by defining a new XY position for the window's upper left corner. Valid X and Y positions depend on the screen size and the window size. X positions are valid if the X position plus the window width in characters is less than or equal to the screen width in characters. Y positions are valid if the Y position plus the window height is less than or equal to the screen height in lines.

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

Note:

If you choose, you can resize a window at the same time you move it. To do this, use the *width* and *length* parameters in the same way that they are used for the SIZE command (see page 3-25).

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

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

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

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

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

3.6 Manipulating a Window's Contents

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

Note:

You can scroll and edit only the *active window*. For information, see Section 3.4 on page 3-21.

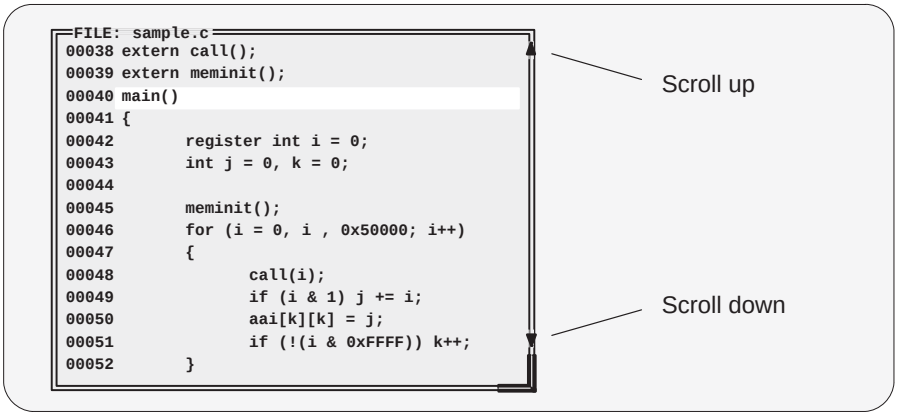
Scrolling through a window's contents

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

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

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

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



To scroll window contents up or down:

1) Point to the appropriate scroll arrow.

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

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

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

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

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

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

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

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

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

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

← → — When a field is selected for editing, the ← and → keys move the cursor within the field. You can use CONTROL ← or CONTROL → to move to the next field except when the COMMAND window is active; in this case, the cursor moves to the beginning of the preceding or next word.

Editing the data displayed in windows

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

Note:

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

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

3.7 Closing a Window

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

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

1) Make the appropriate window active.

2) Press F4 .

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

wr [*window name*]

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

Chapter 4

Entering and Using Commands

The debugger provides you with several methods for entering commands:

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

Mouse use and function key use differ from situation to situation and are described throughout this book whenever applicable. This chapter includes specific rules that apply to entering commands and using pulldown menus.

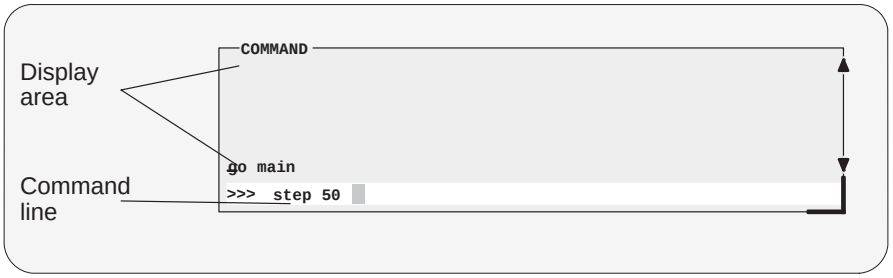
Topic **Page**

4.1 Entering Commands From the Command Line

The debugger supports a complete set of commands that help you to control and monitor program execution, customize the display, and perform other tasks. These commands are discussed in the various sections throughout this book, as they apply to the topic that is being discussed. Chapter 11, *Summary of Commands and Special Keys*, summarizes all of the debugger commands with an alphabetic reference.

Although there are a variety of methods for entering most of the commands, *all* of the commands can be entered by typing them on the command line in the COMMAND window. Figure 4–1 shows the COMMAND window.

Figure 4–1. The COMMAND Window



The COMMAND window serves two purposes:

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

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

Typing in and entering commands

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

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

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

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

To...	Press...
Move back over text without erasing characters	←†
Move forward through text without erasing characters	CONTROL L or →†
Move to the beginning of previous word without erasing characters	CONTROL ←†
Move to the beginning of next word without erasing characters	CONTROL →†
Move to the beginning of the line without erasing characters	ALT ←†
Move to the end of the line without erasing characters	ALT →†
Move back over text while erasing characters	CONTROL H or BACK SPACE or DEL
Move forward through text while erasing characters	SPACE
Insert text into the characters that are already on the command line	INSERT

† You can use the arrow keys only when the COMMAND window is selected.

Notes:

1) When the COMMAND window is not active, you cannot use the arrow keys to move through or edit text on the command line.

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

Sometimes, you can't type a command

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

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

Using the command history

The debugger keeps an internal list, or *command history*, of the commands that you enter. It remembers the last 50 commands that you entered. If you want to reenter a command, you can move through this list, select a command that you've already executed, and reexecute it.

Use these keystrokes to move through the command history.

To...	Press...
Move forward through the list of executed commands, one by one	SHIFT TAB
Move backward through the list of executed commands, one by one	TAB
Repeat the last command that you entered	F2

As you move through the command history, the debugger displays the commands, one by one, on the command line. When you see a command that you want to execute, simply press ↵ to execute the command. You can also edit these displayed commands in the same manner that you can edit new commands, as described on page 4-3.

Clearing the display area

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

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

cls

Recording information from the display area

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

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

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

  dlog *filename*

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

- To end the recording session, enter:

  dlog close ↵

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

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

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

- **Creating a new log file.** If you use the DLOG command without one of the optional parameters, the debugger creates a new file that it records the information into. If you are recording to a log file already, entering a new DLOG command and filename closes the previous log file and opens a new one.

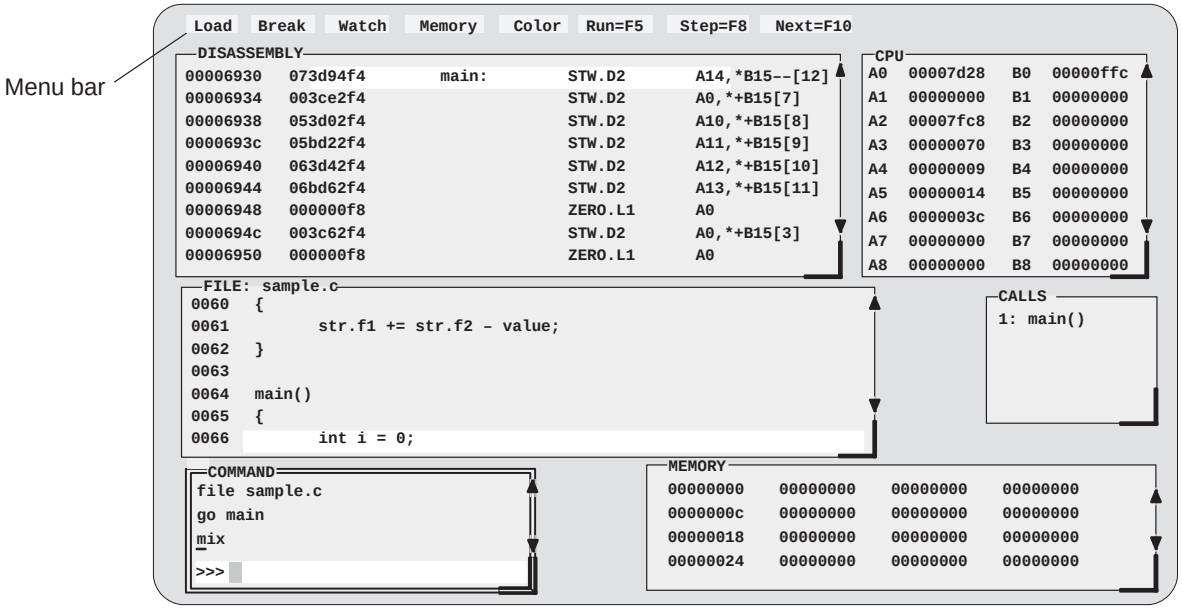
- **Appending to an existing file.** Use the **a** parameter to open an existing file to which to append the information in the display area.

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

4.2 Using the Menu Bar and the Pulldown Menus

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

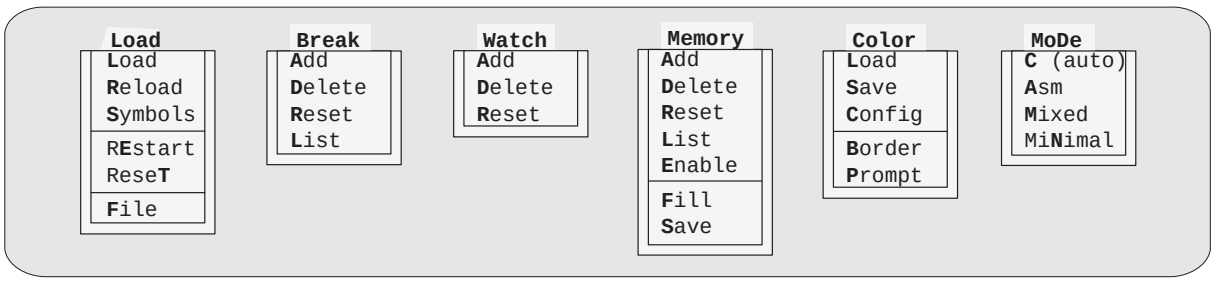
Figure 4–2. The Menu Bar in the Basic Debugger Display



Several of the selections on the menu bar have pulldown menus; if they could all be pulled down at once, they'd look like Figure 4–3.

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

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



Pulldown menus in the profiling environment

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

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

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

Using the pulldown menus

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

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

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

Mouse method 1

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

2) Press the left mouse button, but don't release the button.

3) While pressing the mouse button, move the mouse downward until your selection is highlighted on the menu.

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

Mouse method 2

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

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

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

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

key

Keyboard method 1

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

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

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

Keyboard method 2

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

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

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

↵ 4) When your selection is highlighted, press ↵.

Escaping from the pulldown menus

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

- Press ESC

  or

- Point the mouse outside of the menu; press and then release the left mouse button.

❑ If you pull down a menu and see that it's not the menu you wanted, you can point the mouse at another entry and press the left mouse button, or you can use the ← and → keys to display adjacent menus.

Using menu bar selections that don't have pulldown menus

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

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

There are two ways to execute these choices.

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

2) Click the left mouse button.

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

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

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

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

For more information about the RUN, STEP, and NEXT commands, see Section 6.5, *Running Your Programs*, page 6-10.

4.3 Using Dialog Boxes

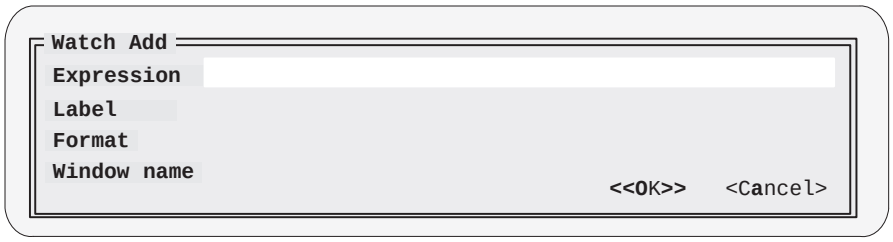
Many of the debugger commands have parameters. When you execute these commands from pulldown menus, you must have some way of providing parameter information. The debugger allows you to do this by displaying a *dialog box* that asks for this information.

Entering text in a dialog box

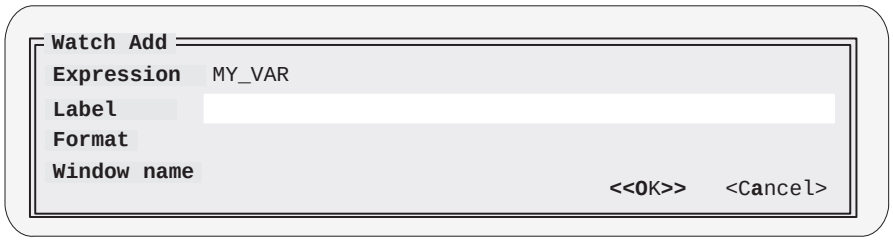
Entering text in a dialog box is much like entering commands on the command line. For example, the Add entry on the Watch menu is equivalent to entering the WA command. This command has four parameters:

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

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



You can enter an *expression* just as you would if you typed the WA command. After you enter an *expression*, press TAB or ↓. The cursor moves down to the next parameter:



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

In the case of the WA command, the *label*, *format*, and *window name* parameters are optional. If you want to enter one of these parameters, you can do so; if you don't want to use these optional parameters, don't type anything in their fields—just continue to the next parameter.

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

- When you display a dialog box for the first time during a debugging session, the parameter fields are empty. When you bring up the same dialog box again, the box displays the last values that you entered. (This is similar to having a command history.) If you want to use the same value, just press TAB or ↓ to move to the next parameter.
- You can edit what you type (or values that remain from a previous entry) in the same way that you can edit text on the command line. See Section 4.1 for more information on editing text on the command line.

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

4.4 Entering Commands From a Batch File

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

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

The format for the TAKE command is:

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

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

    setenv D_DIR "*pathname;pathname***"**

    This allows you to name several directories that the debugger can search.
- By default, the debugger echoes the commands in the display area of the COMMAND window and updates the display as it reads commands from the batch file.
  - If you don't use the *suppress echo flag* parameter, or if you use it but supply a nonzero value, the debugger behaves in the default manner.
  - If you want to suppress the echoing and updating, use the value 0 for the *suppress echo flag* parameter.

Echoing strings in a batch file

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

echo *string*

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

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

```
echo Creating new memory map
```

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

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

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

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

Controlling command execution in a batch file

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

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

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

The debugger includes some predefined constants for use with IF. These constants evaluate to 0 (false) or 1 (true). Table 4–1 shows the constants and their corresponding tools.

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

Constant	Debugger Tool
$$SIM$$	Simulator

If the Boolean expression evaluates to true (1), the debugger executes all commands between the IF and ELSE or ENDIF. Note that the ELSE portion of the command is optional. (See Chapter 12, *Basic Information About C Expressions*, for more information.)

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

  loop *expression*
  debugger command
  debugger command
  .
  .
  endloop

  These looping commands evaluate using the same method as the run conditional command expression. (See Chapter 12, *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:

    ```
    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 the following:

```
loop !0
step
? PC
? A0
endloop
```

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

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

4.5 Defining Your Own Command Strings

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

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

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

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

- **Aliasing several commands.** The *command string* can contain more than one debugger command—just separate the commands with semicolons. Be sure to enclose the *command string* in quotes.

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

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

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

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

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

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

  You could enter:

  ```
  mfil 0x2ff80,0x18,0x1122
  ```

  In this example, the first value (0x2ff80) 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).

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

  For example, assume that the init and mfil aliases have been defined as shown on page 4-17. If you enter:

  `alias` ↵

  you'll see:

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

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

  For example, if you have defined the init alias as shown on page 4-17, you could enter:

  `alias init` ↵

  Then you'd see:

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

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

- **Redefining an alias.** To redefine an alias, reenter the ALIAS command with the same alias name and a new command string.

- **Deleting aliases.** To delete a single alias, use the UNALIAS command:

  unalias *alias name*

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

  `unalias *`

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

Notes:

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

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

Chapter 5

Defining a Memory Map

Before you begin a debugging session, you must supply the debugger with a memory map. The memory map tells the debugger which areas of memory it can and can't access. Note that the commands described in this chapter can also be entered by using the Memory pulldown menu (see Section 4.2, *Using the Menu Bar and the Pulldown Menus*, page 4-7).

Topic **Page**

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

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

Note:

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

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

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

Defining the memory map in a batch file

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

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

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

1) It checks to see whether you've used the –t debugger option. The –t option allows you to specify a batch file other than the initialization batch file shipped with the debugger. If it finds the –t option, the debugger reads and executes the specified file.

2) If you don't use the –t option, the debugger looks for the default initialization batch file. The batch filename for the simulator is called *siminit.cmd*. If the debugger finds the 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*.

Potential memory map problems

You may experience these problems if the memory map isn't correctly defined and enabled:

- **Accessing invalid memory addresses.** If you don't supply a batch file containing memory-map commands, the debugger is initially unable to access any target memory locations. Invalid memory addresses and their contents are highlighted in the data-display windows. (On color monitors, invalid memory locations, by default, are displayed in red.)
- **Accessing an undefined or protected area.** When memory mapping is enabled, the debugger checks each of its memory accesses against the memory map. If you attempt to access an undefined or protected area, the debugger displays an error message.
- **Loading a COFF file with sections that cross a memory range.** Be sure that the map ranges you specify in a COFF file match those that you define with the MA command (described on page 5-5). Alternatively, you can turn memory mapping off during a load by using the MAP OFF command (described on page 5-8).

5.2 A Sample Memory Map

Because you must define a memory map before you can run any programs, it's convenient to define the memory map in the initialization batch files. Figure 5–1 *(a)* shows the memory map commands that are defined in the initialization batch file that accompanies the 'C6x simulator. You must edit the file to your configuration. You can use the file as is, edit it, or create your own memory map batch file to match your own configuration.

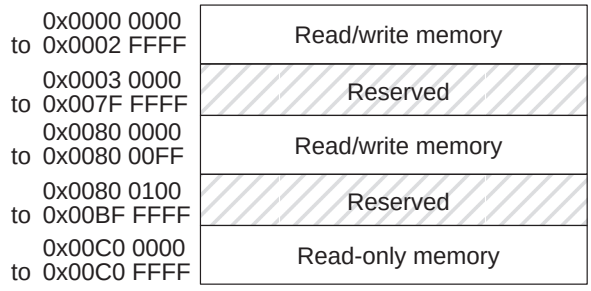
The MA (map add) commands define valid memory ranges and identify the read/write characteristics of the memory ranges. By default, mapping is enabled when you invoke the debugger. Figure 5–1 *(b)* illustrates the memory map defined by the MA commands in Figure 5–1 *(a)*.

Figure 5–1. Sample Memory Map for Use With a TMS320C6x Simulator

(a) Memory map commands

```
ma 0x000000, 0x30000,  RAM
ma 0x800000, 0x100,    RAM
ma 0xC00000, 0x10000,  RAM
```

(b) Memory map for 'C6x local memory



5.3 Identifying Usable Memory Ranges

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

ma *address, length, type*

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

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

  ```
  Conflicting map range
  ```

- The *length* parameter defines the length of the range. This parameter can be any C expression.
- The *type* parameter identifies the read/write characteristics of the memory range. The *type* must be one of these keywords:

To identify this kind of memory . . .	Use this keyword as the *type* parameter . . .
Read-only	**R** or **ROM**
Write-only	**W** or **WOM**
Read/write	**R|W** or **RAM**
No-access	**PROTECT**
Input port	**INPORT** or **P|R**
Output port	**OUTPORT** or **P|W**
Input/output port	**IOPORT** or **P|R|W**

Notes:

- The debugger caches memory that is not defined as a port type (INPORT, OUTPORT, or IOPORT). For ranges that you don't want cached, be sure to map them as ports.
- Be sure that the map ranges that you specify in a COFF file match those that you define with the MA command. Moreover, a command sequence such as:

  ```
  ma x,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 spanned the length of y + z, use the second MA command example. Alternatively, you could turn memory mapping off during a load by using the MAP OFF command.

Memory mapping with the simulator

The 'C6x 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 'C6x, do not allocate more than 20K words of memory in your memory map. For example, the following memory map allocates 64K words of 'C6x program memory.

Example 5–1. Sample Memory Map for the TMS320C6x Using Memory Cache Capabilities

```
MA 0,0x5000,R|W
MA 0x5000,0x5000,R|W
MA 0xa000,0x5000,R|W
MA 0xf000,0x1000,R|W
```

The simulator creates temporary files in a separate directory on your disk. For example, when you enter an MA 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.

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, your disk should have at least 40K bytes of free space available. To do this, enter:

```
ma 0x0, 0x5000, ram ↵
```

Note:

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

5.4 Enabling Memory Mapping

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

map on
or
map off

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

Note:

When memory mapping is enabled, you cannot:

- ❑ Access memory locations that are not defined by an MA command
- ❑ Modify memory areas that are defined as read only or protected

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

```
Error in expression
```

5.5 Checking the Memory Map

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

ml

The ML command lists the starting address, ending address, and read/write characteristics of each defined memory range. Here is an example of the results shown in the display area of the COMMAND window when you enter the ML command:

```
Memory range_______          Attributes
00000000 – 0000ffff         READ WRITE
00800000 – 008000ff         READ WRITE
00c00000 – 00c0ffff         READ WRITE
```

Starting address Ending address

5.6 Modifying the Memory Map During a Debugging Session

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

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

md *address*

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

  ```
  Specified map not found
  ```

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

mr

This resets the debugger memory map.

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

ma *address, length, type*

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

Returning to the original memory map

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

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

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

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

Chapter 6

Loading, Displaying, and Running Code

The main purpose of a debugging system is to allow you to load and run your programs in a test environment. This chapter tells you how to load your programs into the debugging environment, run them on the target system, and view the associated source code. Many of the commands described in this chapter can also be executed from the Load pulldown menu (see Section 4.2, *Using the Menu Bar and the Pulldown Menus*, page 4-7).

Topic **Page**

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

The debugger has three code-display windows:

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

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

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

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

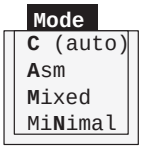
Use this mode	To view	The debugger uses these code-display windows
assembly	*assembly language code only* (even if your program is executing C code)	DISASSEMBLY
auto	*assembly language code* (when that's what your program is running)	DISASSEMBLY
auto	*C code only* (when that's what your program is running)	❑ FILE ❑ CALLS
mixed	*both* assembly language and C code	❑ DISASSEMBLY ❑ FILE ❑ CALLS
minimal	no code	none

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

Selecting a debugging mode

Unless you use the –min command-line option (which selects minimal mode and is discussed on page 1-14), when you first invoke the debugger, it automatically comes up in auto mode. You can then choose assembly or mixed mode. There are several ways to do this.

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



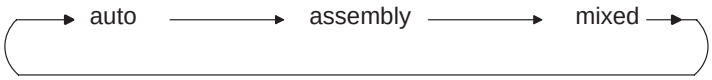
1) Point to the menu name.

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

3) Release the mouse button.

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

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



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

c	Changes from the current mode to auto mode
asm	Changes from the current mode to assembly mode
mix	Changes from the current mode to mixed mode
minimal	Changes from the current mode to minimal mode

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

6.2 Displaying Your Source Programs (or Other Text Files)

The debugger displays two types of code:

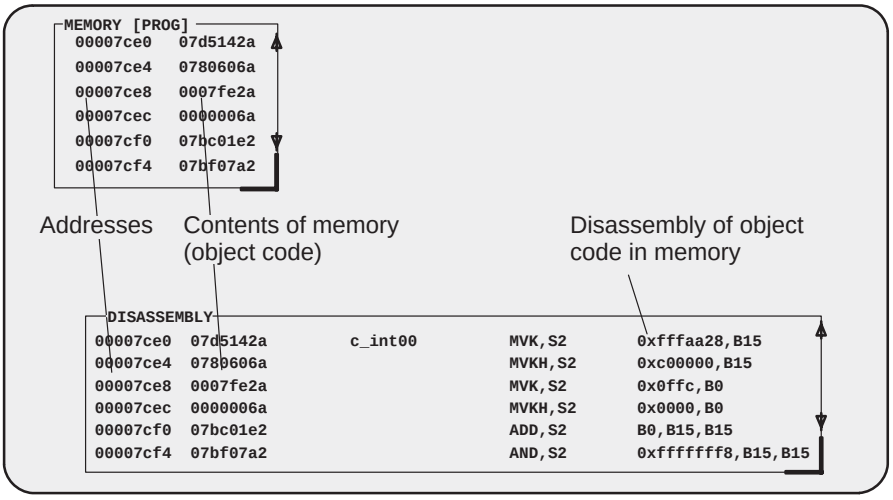
- It displays *assembly language code* in the DISASSEMBLY window in auto, assembly, or mixed mode.
- It displays *C code* in the FILE window in auto and mixed modes.

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

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

Displaying assembly language code

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



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

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

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

dasm *address*
or
dasm *function name*

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

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

addr *address*
or
addr *function name*

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

Displaying C code

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

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

These commands are valid in C and mixed modes:

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

file *filename*

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

(Displaying a file *doesn't* load that file's object code. If you want to be able to run the program, you must load the file's associated object code as described in Section 6.3, *Loading Object Code*.)

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

func *function name*
or
func *address*

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

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

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

addr *address*
or
addr *function name*

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

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

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

2) Click the left mouse button.

(If the CALLS window is active, you can also use the arrow keys and F9 to display the function; see the *CALLS window* discussion on page 3-10 for details.)

Displaying other text files

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

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

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

6.3 Loading Object Code

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

Loading code while invoking the debugger

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

If you want to load a file's symbol table only, use the –s option (this has the same effect as using the debugger's SLOAD command). To do this, enter the appropriate debugger-invocation command along with the name of the object file and specify –s (see page 1-14 for more information).

Loading code after invoking the debugger

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

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

load *object filename*

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

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

reload [*object filename*]

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

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

sload *object filename*

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

6.4 Where the Debugger Looks for Source Files

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

- If you're using LOAD, RELOAD, or SLOAD, you can specify the path as part of the filename.
- If you're 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 doing this is:

    setenv D_SRC "*pathname;pathname***"**

    This allows you to name several directories that the debugger can 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.

use

  - Within the debugger environment, you can use the USE command to name additional source directories. The format for this command is:

    use *directory name*

    You can specify only one directory at a time.

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

6.5 Running Your Programs

To debug your programs, you must execute them on a 'C6x debugging tool (the simulator). The debugger provides two basic types of commands to help you run your code:

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

Defining the starting point for program execution

All run and single-step commands begin executing from the current PC. 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 appropriately highlighted line in the FILE or DISASSEMBLY window. To do this, execute one of these commands:

  dasm PC
  or
  addr PC

Sometimes you may want to modify the PC to point to a different position in your program. There are two ways to do this:

rest

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

  restart
  or
  rest

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

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

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

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

Running code

The debugger supports several run commands.

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

run [*expression*]

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

- ❑ If you don't supply an *expression*, the program executes until it encounters a breakpoint or until you press ESC or the left mouse button.
- ❑ If you supply a logical or relational *expression*, this becomes a conditional run (see page 6-14).
- ❑ If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger executes *count* instructions, halts, then updates the display.

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

go [*address*]

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

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

return
or
ret

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

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

Single-stepping through code

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

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

Note that the debugger ignores interrupts when you use the STEP command to single-step through assembly language code.

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

- If you don't supply an *expression*, the program executes a single statement, then halts.
- If you supply a logical or relational *expression*, this becomes a conditional single-step execution (see page 6-14).
- If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger single-steps *count* C or assembly language statements (depending on the type of code you're in).

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

step [*expression*]

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

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

For more information about the compiler's –g option, see the *TMS320C6x Optimizing C Compiler User's Guide*.

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

cstep [*expression*]

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

next [*expression*]
cnext [*expression*]

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

F8 Acts as a STEP command.

F10 Acts as a NEXT command.

When you use the function keys to single-step through programs, you can't enter an expression for the command.

The debugger allows you to execute several single-step commands from the selections on the menu bar.

To execute a STEP:

1) Point to Step=F8 in the menu bar.

2) Press and release the left mouse button.

To execute a NEXT:

1) Point to Next=F10 in the menu bar.

2) Press and release the left mouse button.

When you use the menu-bar selections to single-step through programs, you can't enter an expression for the command.

Running code while disconnected from the target system

reset The RESET command resets the simulator and reloads the monitor. This is a *software* reset. The format for this command is:

reset

If you execute the RESET command, the simulator simulates the 'C6x processor and peripheral reset operation, putting the processor in a known state.

Running code conditionally

The RUN, STEP, CSTEP, NEXT, and CNEXT commands all have an optional *expression* parameter that can be a relational or logical expression. This type of expression 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 Chapter 8, *Using Software Breakpoints*.) Each time the debugger evaluates the conditional expression, it updates the screen. The debugger applies this algorithm:

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

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

6.6 Halting Program Execution

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

Click the left mouse button.

ESC Press the escape key.

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

Chapter 7

Managing Data

The debugger allows you to examine and modify many different types of data related to the 'C6x and to your program. You can display and modify the values of:

- Individual memory locations or a range of memory
- 'C6x registers
- Variables, including scalar types (ints, chars, etc.) and aggregate types (arrays, structures, etc.)

Topic **Page**

7.1 Where Data Is Displayed

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

Type of data	Window name and purpose
Memory locations	**MEMORY window** Displays the contents of a range of data memory, program memory, or I/O space
Register values	**CPU window** Displays the contents of 'C6x registers
Pointer data or selected variables of an aggregate type	**DISP window** Displays the contents of aggregate types and shows the values of individual members
Selected variables (scalar types or individual members of aggregate types) and specific memory locations or registers	**WATCH window** Displays selected data

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

7.2 Basic Commands for Managing Data

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

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

whatis *symbol*

This lists *symbol*'s data type in the 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.

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];`
	`}`

? 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, ? 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`
	etc.

`? j`	`4194425`
`? j=0x5a`	`90`

Note that the DISP command (described in detail on page 7-11) behaves like the ? command when its *expression* parameter does not identify an aggregate type.

eval The EVAL (evaluate expression) command behaves like the ? command *but does not show the result* in the 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's not necessary to display the result.

7.3 Basic Methods for Changing Data Values

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

Editing data displayed in a window

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

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

There are two similar methods for overwriting displayed data:

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

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

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

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

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

1) Select the window that contains the field you'd like to modify; make this the active window. (Use the mouse, the WIN command, or F6. For more detail, see Section 3.4, *The Active Window*, on page 3-21.)

2) Use arrow keys to move the cursor to the field you'd like to edit.

   - ↑ Moves up 1 field at a time.
   - ↓ Moves down 1 field at a time.
   - ← Moves left 1 field at a time.
   - → Moves right 1 field at a time.

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

ESC 4) Type the new information. If you make a mistake or change your mind, press ESC; this resets the field to its original value.

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

Note:

If you press ↵ when the cursor is in the middle of text, the debugger truncates the input text at the point where you press ↵. Likewise, if you use ← or → to move to the beginning of the previous or next field, the debugger truncates the input text at the point where you press the ← or →.

Advanced "editing"—using expressions with 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, it means that the value of some variable in the expression changes as the result of evaluating the expression.

This means that you can coerce many commands into changing values for you. Specifically, it's most helpful to use ? and EVAL to change data as well as display it.

For example, if you want to see what's in auxiliary register A3, you can enter:

```
? A3 ↵
```

You can also use this type of command to modify A3's contents. Here are some examples of how you might do this:

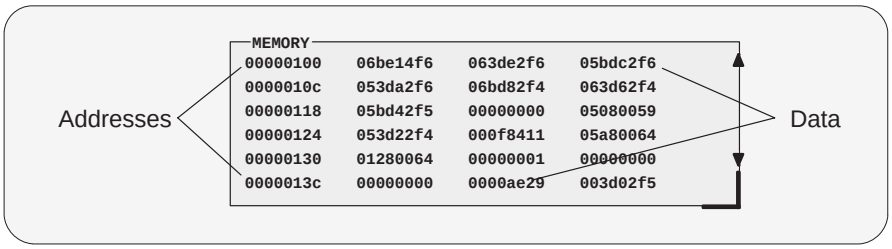
```
? A3++ ↵          Side effect: increments the contents of A3 by 1
eval --A3 ↵       Side effect: decrements the contents of A3 by 1
? A3 = 8 ↵        Side effect: sets A3 to 8
eval A3/=2 ↵      Side effect: divides contents of A3 by 2
```

Note that not all expressions have side effects. For example, if you enter **? A3+4**, the debugger displays the result of adding 4 to the contents of A3 but does not modify A3's contents. Expressions that have side effects must contain an assignment operator or an operator that implies an assignment. Operators that can cause a side effect are:

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

7.4 Managing Data in Memory

In mixed and assembly modes, the debugger maintains a MEMORY window that displays the contents of memory. For details concerning the MEMORY window, see *MEMORY windows* on page 3-13.



The debugger has commands that show the memory values at a specific location or that display a different range of memory in the MEMORY window. The debugger allows you to change the values at individual locations; for more information, refer to Section 7.3, *Basic Methods for Changing Data Values*.

Displaying memory contents

The main way to observe memory contents is to view the display in a MEMORY window. The debugger displays the default MEMORY windows automatically (labeled MEMORY). You can open any number of additional MEMORY windows to display different memory ranges.

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 the screen size).

mem You can use the MEM command to open an additional MEMORY window or to display a different memory range in a window. 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. For more information, see *Resizing a window* on page 3-24.

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

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

  `mem 0x00` ⏎

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

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

  `mem &SYM` ⏎

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

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

  `mem SP – A0 + label` ⏎

❑ The *display format* parameter is optional. When used, the data is displayed in the selected format as shown in Table 7–1 on page 7-17.

❑ The *window name* parameter is optional if you are displaying a different memory range in the default MEMORY window. Use the *window name* parameter when you want to open an additional MEMORY window or change the displayed memory range in an additional MEMORY window. When you open an additional MEMORY window, the debugger appends the *window name* to the MEMORY window label.

You can also change the display of any data-display window—including the MEMORY window—by scrolling through the window's contents. For more details, see *Scrolling through a window's contents* on page 3-29.

Displaying memory contents while you're debugging C

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

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

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

  ```
  ? *0x26
  ```

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

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

  ```
  wa *0x26
  ```

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

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

Saving memory values to a file

ms Sometimes it's useful to save a block of memory values to a file. You can use the MS (memory save) command to do this; the files are saved in COFF format. The syntax for the MS command is:

ms *address, length, filename*

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

For example, to save the values in data memory locations 0x0000 – 0x0010 to a file named memsave, you could enter:

```
ms 0x0,0x10,memsave
```

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

```
load memsave.obj
```

Filling a block of memory

fill Sometimes it's useful to be able to fill an entire block of memory at once. You can do this by using the FILL command. The syntax for this command is:

fill *address, length, data*

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

For example, to fill memory locations 0x10FF–0x110D with the value 0xABCD, you would enter:

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

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

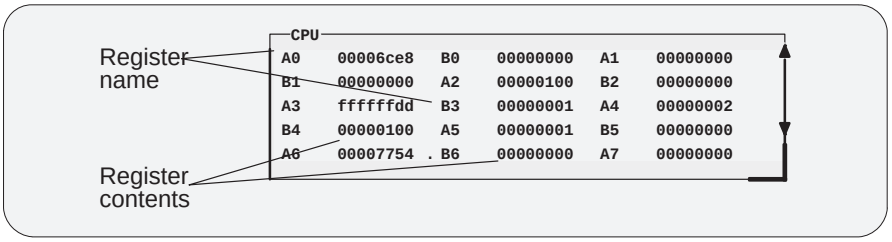
```
mem 0x10ff
```

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

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

7.5 Managing Register Data

In mixed and assembly modes, the debugger maintains a CPU window that displays the contents of individual registers. For details, see *CPU window* on page 3-16.



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, refer to Section 7.3, *Basic Methods for Changing Data Values*.

Displaying register contents

The main way to observe register contents is to view the display in the CPU window. However, you may not be interested in all of the registers; if you're interested in only a few registers, you might want to make the CPU window small and use the extra screen space for the DISASSEMBLY or FILE display. In this type of situation, there are several ways to observe the contents of the selected registers.

- If you have only a temporary interest in the contents of a register, you can use the ? command to display the register's contents. For example, if you want to know the contents of A0, you could enter:

  `? A0` ↵

  The debugger displays A0'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 WA command to display the register in a WATCH window. For example, if you want to observe the status register, you could enter:

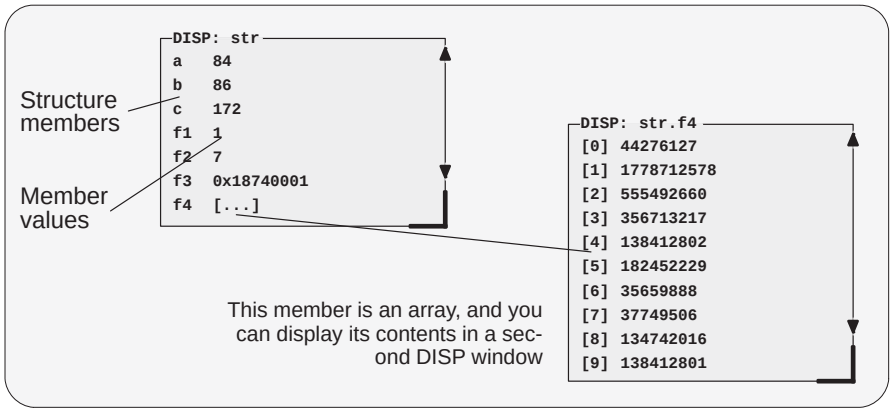
  `wa ST0,Status Register 0` ↵

  This adds the ST0 to the WATCH window and labels it as *Status Register 0*. The register's contents are continuously updated, just as if you were observing the register in the CPU window.

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

7.6 Managing Data in a DISP (Display) Window

The main purpose of the DISP window is to display members of complex, aggregate data types such as arrays and structures. The debugger shows DISP windows *only when you specifically request to see DISP windows* with the DISP command (described below). Note that you can have up to 120 DISP windows open at once. For more details, see *DISP windows* on page 3-17.



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

Displaying data in a DISP window

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

disp *expression*

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

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

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

A member that is an array looks like this	[. . .]
A member that is a structure looks like this	{. . .}
A member that is a pointer looks like an address	0x0000

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

Use the DISP command again; this time, *expression* must identify the member that has additional data. For example, if the first expression identifies a structure named *str* and one of str's members is an array named *f4*, you can display the contents of the array by entering this command:

`disp str.f4` ⏎

This opens a new DISP window that shows the contents of the array. If str has a member named *f3* that is a pointer, you could enter:

`disp *str.f3` ⏎

This opens a window to display what str.f3 points to.

Here's another method of displaying the additional data:

1) Point to the member in the DISP window.

2) Now click the left button.

Here's the third method:

↑ ↓ 1) Use the arrow keys to move the cursor up and down in the list of members.

F9 2) When the cursor is on the desired field, press F9 .

When the debugger opens a second DISP window, the new window may at first be displayed on top of the original DISP window; if so, you can move the windows so that you can see both at once. If the new windows also have members that are pointers or aggregate types, you can continue to open new DISP windows.

Closing a DISP window

Closing a DISP window is a simple, two-step process.

1) Make the DISP window that you want to close active (see Section 3.4, *The Active Window*, on page 3-21).

2) Press F4 .

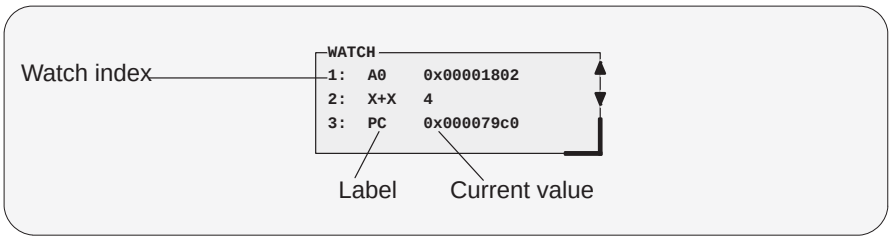
Note that you can close a window and all of its children by closing the original window.

Note:

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

7.7 Managing Data in a WATCH Window

The debugger doesn't maintain a dedicated window that tells you about the status of all the symbols defined in your program. Such a window might be so large that it wouldn't be useful. Instead, the debugger allows you to create a WATCH window that shows you how program execution affects specific expressions, variables, registers, or memory locations.



The debugger displays a WATCH window *only when you specifically request a WATCH window* with the WA command (described below). For additional details concerning the WATCH window, see *WATCH windows* on page 3-18.

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, refer to Section 7.3, *Basic Methods for Changing Data Values*.

Note:

All of the watch commands described can also be accessed from the Watch pulldown menu. For more information about using the the pulldown menus, refer to Section 4.2, *Using the Menu Bar and the Pulldown Menus*, on page 4-7.

Displaying data in the WATCH window

The debugger has one command for adding items to a WATCH window.

wa To open a WATCH window, use the WA (watch add) command. The syntax is:

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

When you first execute WA, the debugger opens a WATCH window. After that, executing WA adds additional values to the WATCH window, unless you open an additional watch window.

- The *expression* parameter can be any C expression, including an expression that has side effects. It's most useful to watch an expression whose value will change over time; constant expressions provide no useful function in the WATCH window.

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

  ```
  wa *0x26 ↵
  ```

- The *label* parameter is optional. When used, it provides a label for the watched entry. If you don't use a *label*, the debugger displays the *expression* in the label field.

- The *display format* parameter is optional. When used, the data is displayed in the selected format as shown in Table 7–1 on page 7-17.

- The *window name* parameter is optional. If you omit the *window name* parameter, the debugger displays the expression in the default WATCH window (labeled WATCH). 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.

Deleting watched values and closing the WATCH window

The debugger supports two commands for deleting items from the WATCH window.

wr If you'd like to close 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.

wd If you'd like to delete a specific item from a WATCH window, use the WD (watch delete) command. The syntax is:

wd *index number* [, *window name*]

Whenever you add an item to a WATCH window, the debugger assigns it an index number. (The illustration of the WATCH window on page 7-14 points to these watch indexes.) The WD command's *index number* parameter must correspond to one of the watch indexes in the named WATCH window.

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

Note:

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

7.8 Displaying Data in Alternative Formats

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

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

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

Changing the default format for specific data types

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

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

The *display format* parameter identifies the new display format for any data of type *data type*. Table 7–1 lists the available formats and the corresponding characters that can be used as the *display format* parameter.

Table 7–1. Display Formats for Debugger Data

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

Table 7–2 lists the C data types that can be used for the *data type* parameter. Only a subset of the display formats applies to each data type, so Table 7–2 also shows valid combinations of data types and display formats.

Table 7–2. Data Types for Displaying Debugger Data

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

Here are some examples:

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

  `setf short, u` ↵

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

  `setf short, *` ↵

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

  `setf` ↵

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

```
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 : Exponential floating point
ptr    : Hexadecimal
```

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

  `setf *` ↵

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

You can also use the ?, MEM, DISP, and WA commands to show data in alternative display formats. (The ? and DISP commands use alternative formats only for scalar types, arrays of scalar types, and individual members of aggregate types.)

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

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

Here are some examples:

- To watch the PC in decimal, enter:

  ```
  wa pc,,d ↵
  ```

- To display memory contents in octal, enter:

  ```
  mem 0x0,o ↵
  ```

- To display an array of integers as characters, enter:

  ```
  disp ai,c ↵
  ```

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

Chapter 8

Using Software Breakpoints

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

Software breakpoints are especially useful in combination with conditional execution (described on page 6-14).

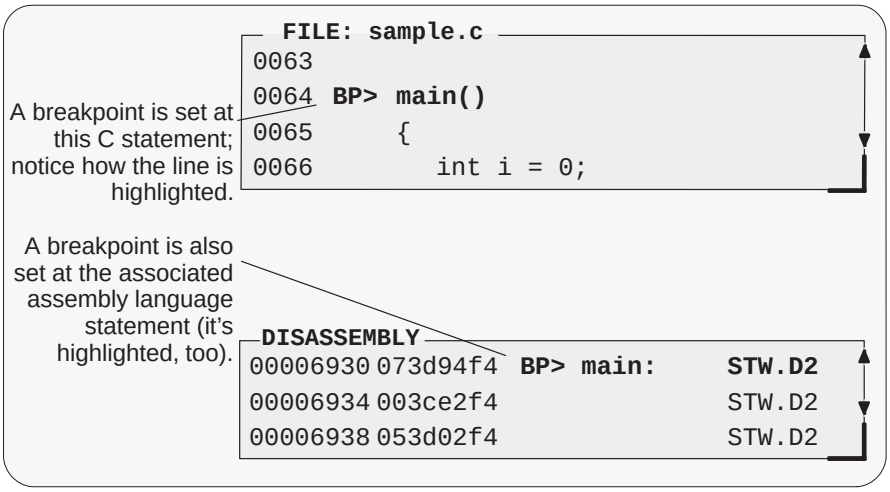
Topic **Page**

8.1 Setting a Software Breakpoint

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

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

If you set a breakpoint in the disassembly, the debugger also highlights the associated C statement. If you set a breakpoint in the C source, the debugger also highlights the associated statement in the disassembly. (If more than one assembly language statement is associated with a C statement, the debugger highlights the first of the associated assembly language statements.)



Notes:

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

2) Up to 200 breakpoints can be set.

There are several ways to set a software breakpoint:

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

2) Click the left button.

Repeating this action clears the breakpoint.

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

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

3) Press the F9 key.

Repeating this action clears the breakpoint.

ba If you know the address where you'd like to set a software breakpoint, you can use the BA (breakpoint add) command. This command is useful because it doesn't require you to search through code to find the desired line. The syntax for the BA command is:

ba *address*

This command sets a breakpoint at *address*. This parameter can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label. You cannot set multiple breakpoints at the same statement.

8.2 Clearing a Software Breakpoint

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

1) Point to a breakpointed assembly language or C statement.

2) Click the left button.

1) Use the arrow keys or the DASM command to move the cursor to a breakpointed assembly language or C statement.

2) Press the F9 key.

br If you want to clear *all* the software breakpoints that are set, use the BR (breakpoint reset) command. This command is useful because it doesn't require you to search through code to find the desired line. The syntax for the BR command is:

br

bd If you'd like to clear one specific software breakpoint and you know the address of this breakpoint, you can use the BD (breakpoint delete) command. The syntax for the BD command is:

bd *address*

This command clears the breakpoint at *address*. This parameter can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label. If no breakpoint is set at *address*, the debugger ignores the command.

8.3 Finding the Software Breakpoints That Are Set

bl Sometimes you may need to know where software breakpoints are set. For example, the BD command's *address* parameter must correspond to the address of a breakpoint that is set. The BL (breakpoint list) command provides an easy way to get a complete listing of all the software breakpoints that are currently set in your program. The syntax for this command is:

bl

The BL command displays a table of software breakpoints in the display area of the COMMAND window. BL lists all the software breakpoints that are set, in the order in which you set them. Here's an example of this type of list:

```
 Address   Symbolic Information
00006930   in main, at line 60, "/user/fred/c6xhll/sample.c"
```

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

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

Chapter 9

Customizing the Debugger Display

The debugger display is completely configurable; you can create the interface that is best suited for your use. Besides being able to size and position individual windows, you can change the appearance of many of the display features, such as window borders, how the current statement is highlighted, etc. In addition, if you're using a color display, you can change the colors of any area on the screen. Once you've customized the display to your liking, you can save the custom configuration for use in future debugging sessions.

Topic **Page**

9.1 Changing the Colors of the Debugger Display

You can use the debugger with a color or a monochrome display; the commands described in this section are most useful if you have a color display. If you are using a monochrome display, these commands change the shades on your display. For example, if you are using a black-and-white display, these commands change the shades of gray that are used.

color
scolor

You can use the COLOR or SCOLOR command to change the colors of areas in the debugger display. The format for these commands is:

color *area name*, *attribute*$_1$ [, *attribute*$_2$ [, *attribute*$_3$ [, *attribute*$_4$]]]
scolor *area name*, *attribute*$_1$ [, *attribute*$_2$ [, *attribute*$_3$ [, *attribute*$_4$]]]

These commands are similar. However, SCOLOR updates the screen immediately, and COLOR doesn't update the screen (the new colors/attributes take effect as soon as the debugger executes another command that updates the screen). You might use the COLOR command several times, followed by an SCOLOR command to put all of the changes into effect at once.

The *area name* parameter identifies the areas of the display that are affected. The *attributes* identify how the areas are affected. Table 9–1 lists the valid values for the *attribute* parameters.

Table 9–1. Colors and Other Attributes for the COLOR and SCOLOR Commands

(a) Colors

black	blue	green	cyan
red	magenta	yellow	white

(b) Other attributes

bright	blink

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

Table 9–2 lists valid values for the *area name* parameters. This is a long list; the subsections following the table further identify these areas.

Table 9–2. Summary of Area Names for the COLOR and SCOLOR Commands

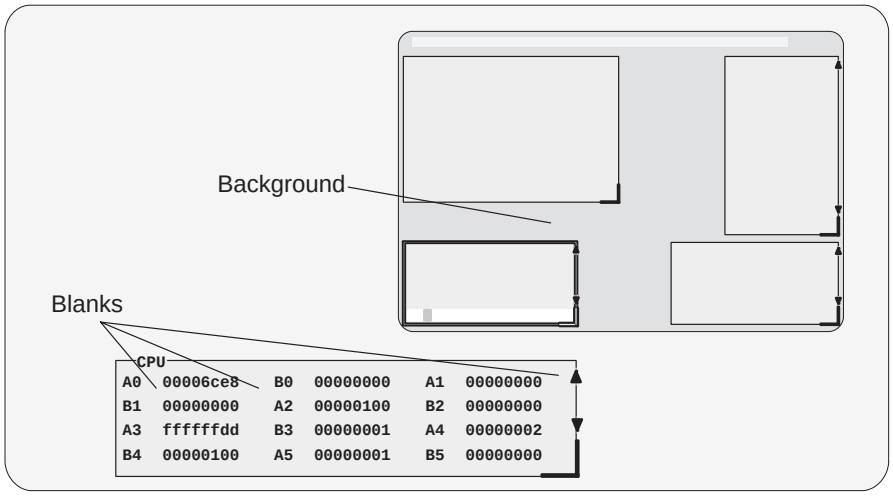
menu_bar	menu_border	menu_entry	menu_cmd
menu_hilite	menu_hicmd	win_border	win_hiborder
win_resize	field_text	field_hilite	field_edit
field_label	field_error	cmd_prompt	cmd_input
cmd_cursor	cmd_echo	asm_data	asm_cdata
asm_label	asm_clabel	background	blanks
error_msg	file_line	file_eof	file_text
file_brk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

Note: Listing order is left to right, top to bottom.

You don't have to type an entire *attribute* or *area name*; you need to type only enough letters to uniquely identify either parameter. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they're listed in Table 9–2 (left to right, top to bottom).

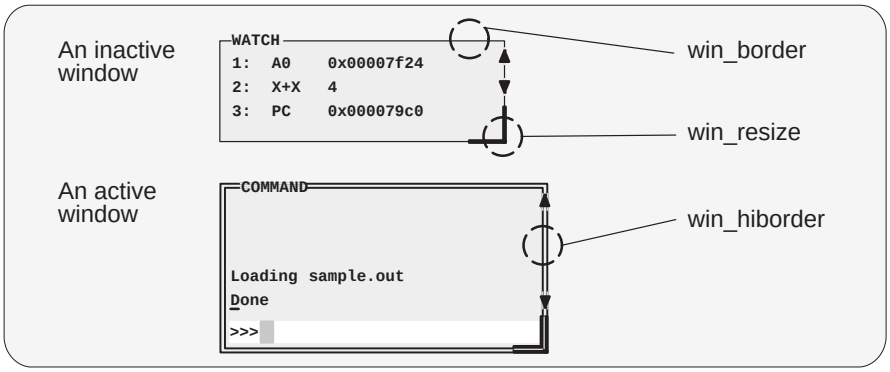
The remainder of this section identifies these areas.

Area names: common display areas



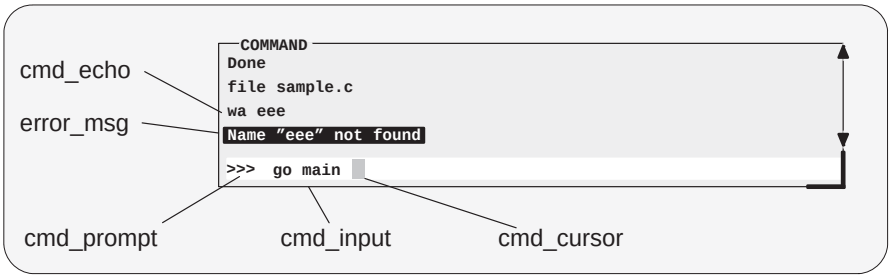
Area identification	Parameter name
Screen background (behind all windows)	background
Window background (inside windows)	blanks

Area names: window borders



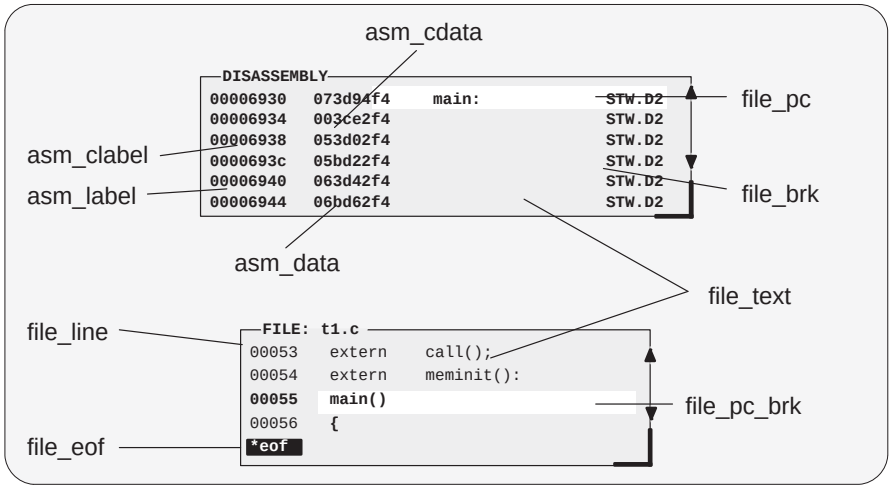
Area identification	Parameter name
Window border for any window that isn't active	win_border
The reversed L in the lower right corner of a resizable window	win_resize
Window border of the active window	win_hiborder

Area names: COMMAND window



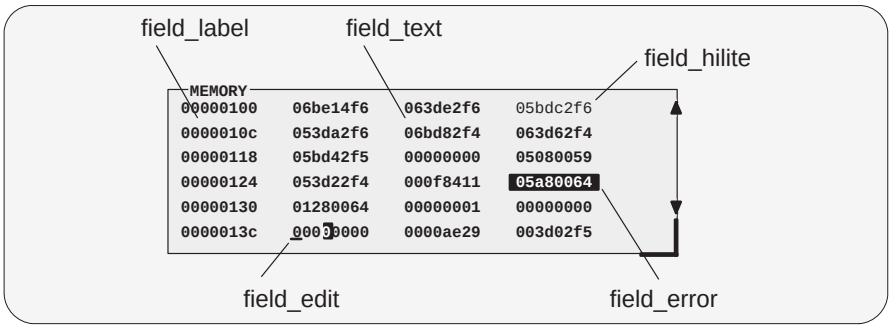
Area identification	Parameter name
Echoed commands in display area	cmd_echo
Errors shown in display area	error_msg
Command-line prompt	cmd_prompt
Text that you enter on the command line	cmd_input
Command-line cursor	cmd_cursor

Area names: DISASSEMBLY and FILE windows



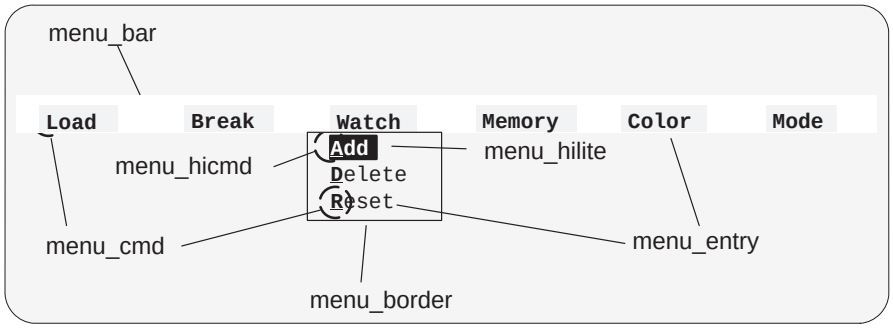
Area identification	Parameter name
Object code in DISASSEMBLY window that is associated with current C statement	asm_cdata
Object code in DISASSEMBLY window	asm_data
Addresses in DISASSEMBLY window	asm_label
Addresses in DISASSEMBLY window that are associated with current C statement	asm_clabel
Line numbers in FILE window	file_line
End-of-file marker in FILE window	file_eof
Text in FILE or DISASSEMBLY window	file_text
Breakpointed text in FILE or DISASSEMBLY window	file_brk
Current PC in FILE or DISASSEMBLY window	file_pc
Breakpoint at current PC in FILE or DISASSEMBLY window	file_pc_brk

Area names: data-display windows



Area identification	Parameter name
Label of a window field (includes register names in CPU window, addresses in MEMORY window, index numbers and labels in WATCH window, member names in DISP window)	field_label
Text of a window field (includes data values for all data-display windows) and of most command output messages in command window	field_text
Text of a highlighted field	field_hilite
Text of a field that has an error (such as an invalid memory location)	field_error
Text of a field being edited (includes data values for all data-display windows)	field_edit

Area names: menu bar and pulldown menus



Area identification	Parameter name
Top line of display screen; background to main menu choices	menu_bar
Border of any pulldown menu	menu_border
Text of a menu entry	menu_entry
Invocation key for a menu or menu entry	menu_cmd
Text for current (selected) menu entry	menu_hilite
Invocation key for current (selected) menu entry	menu_hicmd

9.2 Changing the Border Styles of the Windows

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

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

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

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

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

Here are some examples of the BORDER command. Note that you can skip parameters, if desired.

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

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

9.3 Saving and Using Custom Displays

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

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

The debugger supports two commands for saving and restoring custom screen configurations into files. The filenames that you use for restoring configurations must correspond to the filenames that you used for saving configurations. Note that these are binary files, not text files, so you can't edit the files with a text editor.

Changing the default display for monochrome monitors

The default display is most useful with color monitors. The debugger highlights changed values, messages, and other information with color; this may not be particularly helpful if you are using a monochrome monitor.

The debugger package includes another screen configuration file named mono.clr, which defines a screen configuration that can be used with monochrome monitors. The best way to use this configuration is to rename the file:

1) Rename the original init.clr file—you might want to call it color.clr.

2) Next, rename the mono.clr file. Call it init.clr. Now, whenever you invoke the debugger, it will automatically come up with a customized screen configuration for monochrome monitors.

If you aren't happy with the way that this file defines the screen configuration, you can customize it.

Saving a custom display

ssave Once you've customized the debugger display to your liking, you can use the SSAVE command to save the current screen configuration to a file. The format for this command is:

ssave [*filename*]

This saves the screen resolution, border styles, colors, window positions, window sizes, and (on PCs) video mode (EGA, VGA, etc.) for all debugging modes.

The *filename* parameter names the new screen configuration file. You can include path information (including relative pathnames); if you don't specify path information, the debugger places the file in the current directory. If you don't supply a filename, the debugger saves the current configuration into a file named init.clr.

Note that you can execute this command as the Save selection on the Color pulldown menu.

Loading a custom display

sconfig You can use the SCONFIG command to restore the display to a particular configuration. The format for this command is:

sconfig [*filename*]

This restores the screen resolution, colors, window positions, window sizes, border styles, and (on PCs) video mode (EGA, CGA, MDA, etc.) saved in *filename*. Screen resolution and video mode are restored either by changing the mode (on video cards with switchable modes) or by resizing the debugger screen (on other hosts).

If you don't supply a *filename*, the debugger looks for init.clr. The debugger searches for the file in the current directory and then in directories named with the D_DIR environment variable.

Note that you can execute this command as the Load selection on the Color pulldown menu.

Note:

The file created by the SSAVE command in this version of the debugger saves positional, screen size, and video mode information that was not saved by SSAVE in previous versions of the debugger. The format of this new information is not compatible with the old format. If you attempt to load an earlier version's SCONFIG file, the debugger will issue an error message and stop the load.

Invoking the debugger with a custom display

If you set up the screen in a way that you like and always want to invoke the debugger with this screen configuration, you have two choices for accomplishing this:

- Save the configuration in init.clr.
- Add a line to the batch file that the debugger executes at invocation time (init.cmd). This line should use the SCONFIG command to load the custom configuration.

Returning to the default display

If you saved a custom configuration into init.clr but don't want the debugger to come up in that configuration, rename the file or delete it. If you are in the debugger, have changed the configuration, and would like to revert to the default, just execute the SCONFIG command without a filename.

9.4 Changing the Prompt

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

prompt *new prompt*

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

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

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

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

Chapter 10

Profiling Code Execution

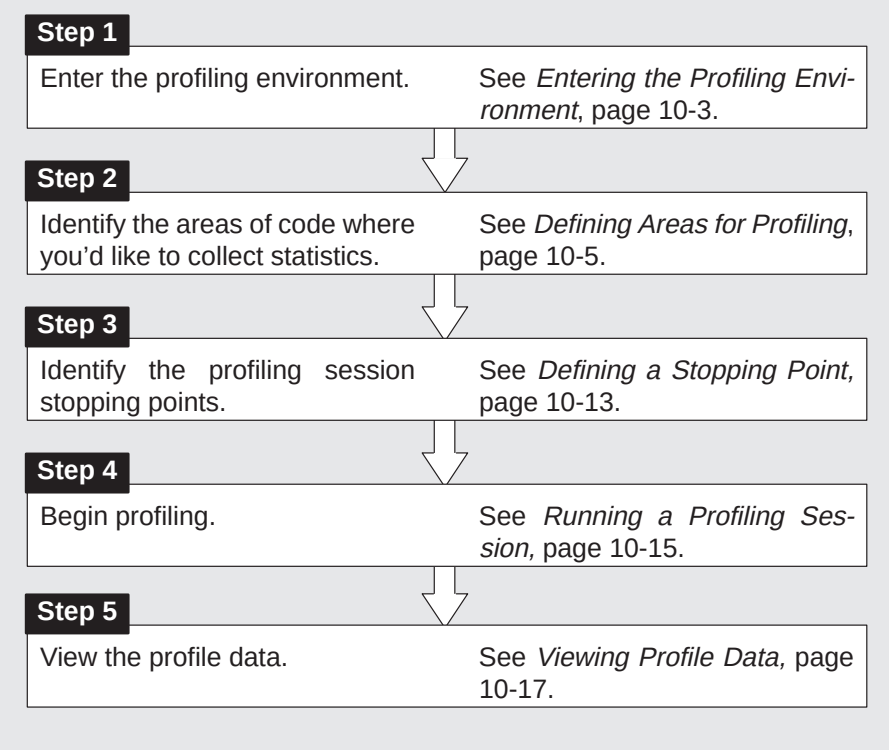
The profiling environment is a special debugger environment that lets you collect execution statistics for your code.

Note that the profiling environment is *separate* from the basic debugging environment; the only way to switch between the two environments is by exiting and then reinvoking the debugger.

Topic **Page**

10.1 An Overview of the Profiling Process

Profiling consists of five simple steps:



Note:

When you compile a program that will be profiled, you must use the –g and the –as options. The –g option includes symbolic debugging information; the –as option ensures that you will be able to include ranges as profile areas.

A profiling strategy

The profiling environment provides a method for collecting execution statistics about specific areas in your code. This gives you immediate feedback on your application's performance. Here's a suggestion for a basic approach to optimizing the performance of your program.

1) Mark all the functions in your program as profile areas.
2) Run a profiling session; find the busiest functions.
3) Unmark all the functions.
4) Mark the individual lines in the busy functions and run another profiling session.

10.2 Entering the Profiling Environment

To enter the profiling environment, invoke the debugger with the **–profile** option. At the system command line, enter the appropriate command:

```
sim6x –profile ↵
```

Use any additional debugger options that you desire (–b, –p, etc.).

Restrictions of the profiling environment

Some restrictions apply to the profiling environment:

- You'll always be in mixed mode.
- COMMAND, DISASSEMBLY, FILE, and PROFILE are the only windows available; additional windows, such as the WATCH window, cannot be opened.
- Breakpoints cannot be set. (However, you can use a similar feature called *stopping points* when you mark sections of code for profiling.)
- The profiling environment supports only a subset of the debugger commands. Table 10–1 lists the debugger commands that can and can't be used in the profiling environment.

Table 10–1. Debugger Commands That Can/Can't Be Used in the Profiling Environment

Can be used		Can't be used	
?	MR	ADDR	HALT
ALIAS	PROMPT	ASM	MEM
CD	QUIT	BA	MIX
CLS	RELOAD	BD	MS
DASM	RESET	BL	NEXT
DIR	RESTART	BORDER	RETURN
ECHO	SCONFIG	BR	RUN
EVAL	SIZE	C	SCOLOR
FILE	SLOAD	CALLS	SETF
FUNC	SSAVE	CNEXT	SOUND
IF/ELSE/ENDIF	SYSTEM	COLOR	STEP
LOAD	TAKE	CSTEP	WA
LOOP/ENDLOOP	UNALIAS	DISP	WD
MA	USE	FILL	WHATIS
MAP	VERSION	GO	WR
MD	WIN		
ML	ZOOM		
MOVE			

Be sure you don't use any of the "can't be used" commands in your initialization batch file.

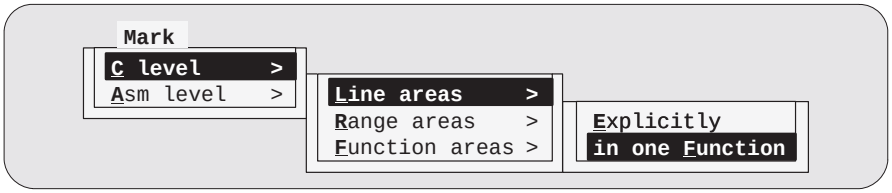
Using pulldown menus in the profiling environment

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

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

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

The profiling environment's pulldown menus operate like the basic debugger pulldown menus. However, several of the menus have additional submenus. A submenu is indicated by a > character following a menu item. For example, here's one of the submenus for the Mark menu:



Chapter 11, *Summary of Commands and Special Keys*, shows which debugger commands are associated with the menu items in the basic debugger pulldown menus. Because the profiling environment supports over 100 profile-specific commands, it's not practical to show the commands associated with the menu choices. Here's a tip to help you with the profiling commands: the highlighted menu letters form the name of the corresponding debugger command. For example, if you prefer the function-key approach to using menus, the highlighted letters in **M**ark→ **C** level→**L**ine areas→in one **F**unction show that you could press ALT M, C, L, F. This also shows that the corresponding debugger command is MCLF.

10.3 Defining Areas for Profiling

Within the profiling environment, you can collect statistics on three types of areas:

- **Individual lines** in C or disassembly
- **Ranges** in C or disassembly
- **Functions** in C only

To identify any of these areas for profiling, mark the line, range, or function. You can disable areas so that they won't affect the profile data, and you can reenable areas that have been disabled. You can also unmark areas that you are no longer interested in.

The mouse is the simplest way to mark, disable, enable, and unmark tasks. The pulldown menus also support these tasks and more complex tasks.

The following subsections explain how to mark, disable, reenable, and unmark profile areas by using the mouse or the pulldown menus. The individual commands are summarized in *Restrictions of the profiling environment* on page 10-3. *Restrictions on profiling areas* are summarized on page 10-12.

Marking an area

Marking an area qualifies it for profiling so that the debugger can collect timing statistics about the area.

Remember, to display C code, use the FILE or FUNC command; to display disassembly, use the DASM command.

Notes:

1) Marking an area in C *does not* mark the associated code in disassembly.
2) Areas can be nested; for example, you can mark a line within a marked range. The debugger will report statistics for both the line and the function.
3) Ranges cannot overlap, and they cannot span function boundaries.

Marking a line. These instructions apply to both C and disassembly.

4) Point to the line you want to mark.

5) Click the left mouse button.

The beginning of the line will be highlighted with >>.

6) Click the left mouse button again.

The beginning of the line will be highlighted with Le> *(line enabled).*

Marking a range. These instructions apply to both C and disassembly.

1) Point to the first line of the range you want to mark.

2) Click the left mouse button.

The beginning of the line will be highlighted with >>.

3) Point to the last line of the range.

4) Click the left mouse button again.

The beginning of the line will be highlighted with Re> *(range enabled), marking the beginning of the range. The last line will be highlighted with* <<, *marking the end of the range.*

Marking a function. These instructions apply to C only.

1) Point to the statement that declares the function you want to mark.

2) Click the left mouse button.

The beginning of the line will be highlighted with Fe> *(function enabled).*

Table 10–2 lists the menu selections for marking areas. The highlighted areas show the keys that you can use if you prefer to use the function-key method of selecting menu choices.

Table 10–2. Menu Selections for Marking Areas

To mark this area	**C only:** **M**ark→**C** level	**Disassembly only:** **M**ark→**A**sm level
Lines	→**L**ine areas	→**L**ine areas
❑ By line number†	→**E**xplicitly	→**E**xplicitly
❑ All lines in a function	→in one **F**unction	→in one **F**unction
Ranges	→**R**ange areas	→**R**ange areas
❑ By line numbers†	→**E**xplicitly	→**E**xplicitly
Functions	→**F**unction areas	not applicable
❑ By function name	→**E**xplicitly	
❑ All functions in a module	→in one **M**odule	
❑ All functions everywhere	→**G**lobally	

† C areas are identified by line number; disassembly areas are identified by address.

Disabling an area

At times, it is useful to identify areas that you don't want to impact profile statistics. To do this, you should *disable* the appropriate area. Disabling effectively subtracts the timing information of the disabled area from all profile areas that include or call the disabled area. Areas must be marked before they can be disabled.

For example, if you have marked a function that calls a standard C function such as malloc(), you may not want malloc() to affect the statistics for the calling function. You could mark the line that calls malloc(), and then disable the line. This way, the profile statistics for the function would not include the statistics for malloc().

Note:

If you disable an area after you've already collected statistics on it, that information will be lost.

The simplest way to disable an area is to use the mouse, as described below.

Disabling a range area:

1) Point to the marked line.

2) Click the left mouse button once.

   The beginning of the line will be highlighted with Rd> *(range disabled).*

Disabling a function area:

1) Point to the marked statement that declares the function.

2) Click the left mouse button once.

   The beginning of the line will be highlighted with Fd> *(function disabled).*

Table 10–3 lists the menu selections for disabling areas. The highlighted areas show the keys that you can use if you prefer to use the function-key method of selecting menu choices.

Table 10–3. Menu Selections for Disabling Areas

To disable this area	**C only:** **D**isable→**C** level	**Disassembly only:** **D**isable→**A**sm level	**C *and* disassembly:** **D**isable→**B**oth levels
Lines	→**L**ine areas	→**L**ine areas	→**L**ine areas
❑ By line number†	→**E**xplicitly	→**E**xplicitly	not applicable
❑ All lines in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All lines in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All lines everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Ranges	→**R**ange areas	→**R**ange areas	→**R**ange areas
❑ By line numbers†	→**E**xplicitly	→**E**xplicitly	not applicable
❑ All ranges in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All ranges in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All ranges everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Functions	→**F**unction areas	not applicable	→**F**unction areas
❑ By function name	→**E**xplicitly		not applicable
❑ All functions in a module	→in one **M**odule		→in one **M**odule
❑ All functions everywhere	→**G**lobally		→**G**lobally
All areas	→**A**ll areas	→**A**ll areas	→**A**ll areas
❑ All areas in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All areas in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All areas everywhere	→**G**lobally	→**G**lobally	→**G**lobally

† C areas are identified by line number; disassembly areas are identified by address.

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 use the mouse, just point to the line, the function, or the first line of a range, and click the left mouse button; the range will once again be highlighted in the same way as a marked area.

In addition to using the mouse, you can enable an area by using one of the commands listed in Table 10–4. However, the easiest way to enter these commands is by accessing them from the Enable menu.

Table 10–4. Menu Selections for Enabling Areas

To enable this area	**C only:** **D**isable→**C** level	**Disassembly only:** **E**nable→**A**sm level	**C *and* disassembly:** **E**nable→**B**oth levels
Lines	→**L**ine areas	→**L**ine areas	→**L**ine areas
❑ By line number†	→**E**xplicitly	→**E**xplicitly	not applicable
❑ All lines in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All lines in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All lines everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Ranges	→**R**ange areas	→**R**ange areas	→**R**ange areas
❑ By line numbers†	→**E**xplicitly	→**E**xplicitly	not applicable
❑ All ranges in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All ranges in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All ranges everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Functions	→**F**unction areas	not applicable	→**F**unction areas
❑ By function name	→**E**xplicitly		not applicable
❑ All functions in a module	→in one **M**odule		→in one **M**odule
❑ All functions everywhere	→**G**lobally		→**G**lobally
All areas	→**A**ll areas	→**A**ll areas	→**A**ll areas
❑ All areas in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All areas in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All areas everywhere	→**G**lobally	→**G**lobally	→**G**lobally

† C areas are identified by line number; disassembly areas are identified by address.

Unmarking an area

If you want to stop collecting information about a specific area, unmark it. You can use the mouse or key method.

Unmarking a line area:

1) Point to the marked line.

2) Click the right mouse button once.

The line will no longer be highlighted.

Unmarking a range area:

1) Point to the marked line.

2) Click the right mouse button once.

The line will no longer be highlighted.

Unmarking a function area:

1) Point to the marked statement that declares the function.

2) Click the right mouse button once.

The line will no longer be highlighted.

Table 10–5 lists the selections on the Unmark menu.

Table 10–5. Menu Selections for Unmarking Areas

To unmark this area	**C only:** **U**nmark→**C** level	**Disassembly only:** **U**nmark→**A**sm level	**C *and* disassembly:** **U**nmark→**B**oth levels
Lines	→**L**ine areas	→**L**ine areas	→**L**ine areas
❑ By line number†	→**E**xplicitly	→**E**xplicitly	not applicable
❑ All lines in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All lines in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All lines everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Ranges	→**R**ange areas	→**R**ange areas	→**R**ange areas
❑ By line numbers†	→**E**xplicitly	→**E**xplicitly	not applicable
❑ All ranges in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All ranges in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All ranges everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Functions	→**F**unction areas	not applicable	→**F**unction areas
❑ By function name	→**E**xplicitly		not applicable
❑ All functions in a module	→in one **M**odule		→in one **M**odule
❑ All functions everywhere	→**G**lobally		→**G**lobally
All areas	→**A**ll areas	→**A**ll areas	→**A**ll areas
❑ All areas in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All areas in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All areas everywhere	→**G**lobally	→**G**lobally	→**G**lobally

† C areas are identified by line number; disassembly areas are identified by address.

Restrictions on profiling areas

The following restrictions apply to profiling areas:

- ❑ There must be a minimum of three instructions between a delayed branch and the beginning of an area.
- ❑ An area cannot begin or end on the RPTS instruction or on the instruction to be repeated.
- ❑ An area cannot begin or end on the last instruction of a repeat block.

10.4 Defining a Stopping Point

Before you run a profiling session, you must identify the point where the debugger should stop collecting statistics. By default, C programs contain an *exit* label, and this is defined as the default stopping point when you load your program. (You can delete exit as a stopping point, if you wish.) If your program does not contain an exit label, or if you prefer to stop at a different point, you can define another stopping point. You can set multiple stopping points; the debugger will stop at the first one it finds.

Each stopping point is highlighted in the FILE or DISASSEMBLY window with a ***** character at the beginning of the line. Even though no statistics can be gathered for areas following a stopping point, the areas will be listed in the PROFILE window.

You can use the mouse or commands to add or delete a stopping point; you can also use commands to list or reset all the stopping points.

Note:

You cannot set a stopping point on a statement that has already been defined as a part of a profile area.

To set a stopping point:

1) Point to the statement that you want to add as a stopping point.

2) Click the right mouse button.

To remove a stopping point:

1) Point to the statement marking the stopping point that you want to delete.

2) Click the right mouse button.

The debugger supports several commands for adding, deleting, resetting, and listing stopping points (described below); all of these commands can also be entered from the Stop-points menu.

sa To add a stopping point, use the SA (stop add) command. The syntax for this command is:

sa *address*

This adds *address* as a stopping point. The *address* parameter can be a label, a function name, or a memory address.

sd To delete a stopping point, use the SD (stop delete) command. The syntax for this command is:

sd *address*

This deletes *address* as a stopping point. As for SA, the *address* can be a label, a function name, or a memory address.

sr To delete all the stopping points at once, use the SR (stop reset) command. The syntax for this command is:

sr

This deletes all stopping points, including the default *exit* (if it exists).

sl To see a list of all the stopping points that are currently set, use the SL (stop list) command. The syntax for this command is:

sl

10.5 Running a Profiling Session

Once you have defined profile areas and a stopping point, you can run a profiling session. You can run two types of profiling sessions:

- A **full profile** collects a full set of statistics for the defined profile areas.
- A **quick profile** collects a subset of the available statistics (it doesn't collect exclusive or exclusive max data, which are described in Section 10.6, *Viewing Profile Data*). This reduces overhead because the debugger doesn't have to track entering/exiting subroutines within an area.

The debugger supports commands for running both types of sessions. In addition, the debugger supports a command that helps you to resume a profiling session. All of these commands can also be entered from the Profile menu.

pf To run a full profiling session, use the PF (profile full) command. The syntax for this command is:

pf *starting point* [*, update rate*]

pq To run a quick profiling session, use the PQ (profile quick) command. The syntax for this command is:

pq *starting point* [*, update rate*]

The debugger will collect statistics on the defined areas between the *starting point* and the stopping point. The *starting point* parameter can be a label, a function name, or a memory address. There is no default starting point.

The *update rate* is an optional parameter that determines how often the statistics listed in the PROFILE window will be updated. The *update rate* parameter can have one of these values:

0 An *update rate* of 0 means that the statistics listed in the PROFILE window are not updated until the profiling session is halted. A "spinning wheel" character will be shown at the beginning of the PROFILE window label line to indicate that a profiling session is in progress. 0 is the default value.

≥1 If a number greater than or equal to 1 is supplied, the statistics in the PROFILE window are updated during the profiling session. If a value of 1 is supplied, the data will be updated as often as possible. When larger numbers are supplied, the data is updated less often.

<0 If a negative number is supplied, the statistics listed in the PROFILE window are not updated until the profiling session is halted. The "spinning wheel" character is not displayed.

No matter which *update rate* you choose, you can force the PROFILE window to be updated during a profiling session by pointing to the window header and clicking a mouse button.

After you enter a PF or PQ command, your program restarts and runs to the defined starting point. Profiling begins when the starting point is reached and continues until a stopping point is reached or until you halt the profiling session by pressing ESC.

pr Use the PR command to resume a profiling session that has halted. The syntax for this command is:

pr [*clear data* [, *update rate*]]

The optional *clear data* parameter tells the debugger whether or not it should clear out the previously collected data. The *clear data* parameter can have one of these values:

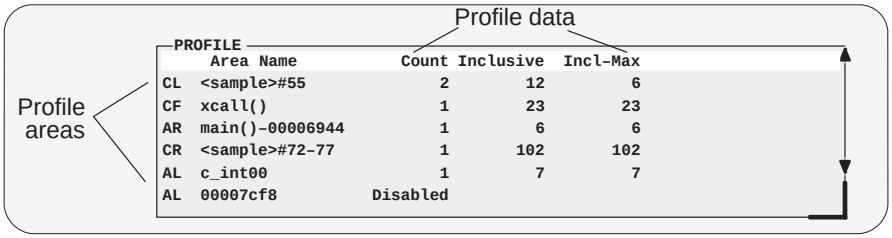
0	The profiler will continue to collect data (adding it to the existing data for the profiled areas) and to use the previous internal profile stacks. 0 is the default value.
nonzero	All previously collected profile data and internal profile stacks are cleared.

The *update rate* parameter is the same as for the PF and PQ commands.

10.6 Viewing Profile Data

The statistics collected during a profiling session are displayed in the PROFILE window. Figure 10–1 shows an example of this window.

Figure 10–1. An Example of the PROFILE Window



The example in Figure 10–1 shows the PROFILE window with some default conditions:

- Column headings show the labels for the default set of profile data, including *Count, Inclusive, Incl-Max, Exclusive*, and *Excl-Max*.
- The data is sorted on the address of the first line in each area.
- All marked areas are listed, including disabled areas.

You can modify the PROFILE window to display selected profile areas or different data; you can also sort the data differently. The following subsections explain how to do these things.

Note:

To reset the PROFILE display back to its default characteristics, use View→Reset.

Viewing different profile data

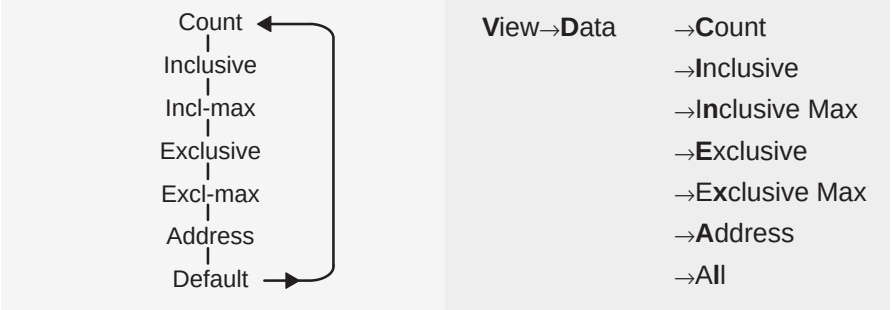
By default, the PROFILE window shows a set of statistics labeled as Count, Inclusive, Incl-Max, Exclusive, and Excl-Max. The Address field, which is not part of the default statistics, can also be displayed. Table 10–6 describes the statistic that each field represents.

Table 10–6. Types of Data Shown in the PROFILE Window

Label	Profile data
Count	The number of times a profile area is entered during a session.
Inclusive	The total execution time (cycle count) of a profile area, including the execution time of any subroutines called from within the profile area.
Incl-Max (inclusive maximum)	The maximum inclusive time for one iteration of a profile area. If the profiled code contains no flow control (such as conditional processing), inclusive-maximum will equal the inclusive timing divided by the count.
Exclusive	The total execution time (cycle count) of a profile area, excluding the execution time of any subroutines called from within the profile area. In general, the exclusive data provides the best statistics for comparing the execution time of one profile area to another area.
Excl-Max (exclusive maximum)	The maximum exclusive time for one iteration of a profile area.
Address	The memory address of the line. If the area is a function or range, the Address field shows the memory address of the first line in the area.

In addition to viewing this data in the default manner, you can view each of these statistics individually. The benefit of viewing them individually is that in addition to a cycle count, you are also supplied with a percentage indication and a histogram.

In order to view the fields individually, you can use the mouse—just point to the header line in the PROFILE window and click a mouse button. You can also use the View→Data menu to select the field you'd like to display. When you use the left mouse button to click on the header, fields are displayed individually in the order listed below on the left. (Use the right mouse button to go in the opposite direction.) On the right are the corresponding menu selections.



One advantage of using the mouse is that you can change the display while you're profiling.

Data accuracy

During a profiling session, the debugger sets many internal breakpoints and issues a series of RUNB commands. As a result, the processor is momentarily halted when entering and exiting profiling areas. This stopping and starting can affect the cycle count information (due to pipeline flushing and the mechanics of software breakpoints) so that it varies from session to session. This method of profiling is referred to as *intrusive profiling*.

Treat the data as *relative*, not absolute. The percentages and histograms are relevant only to the cycle count from the starting point to the stopping point—not to overall performance. Even though the cycle counts may change if you profiled the same area twice, the relationship of that area to other profiled areas should not change.

Sorting profile data

By default, the data displayed in the PROFILE window is sorted 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).

You can sort the data on any of the data fields by using the View→Sort menu. For example, to sort all the data on the basis of the values of the Inclusive field, use View→Sort→Inclusive; the area with the highest Count field will display first, and the area with the lowest Count field will display last. This applies even when you are viewing individual fields.

Viewing different profile areas

By default, all marked areas are listed in the PROFILE window. You can modify the window to display selected areas. To do this, use the selections on the View→Filter pulldown menu; these selections are summarized in Table 10–7.

Table 10–7. Menu Selections for Displaying Areas in the PROFILE Window

To view these areas	**C only:** **V**iew→**F**ilter→**C** level	**Disassembly only:** **V**iew→**F**ilter→**A**sm level	**C *and* disassembly:** **V**iew→**F**ilter→**B**oth levels
Lines	→**L**ine areas	→**L**ine areas	→**L**ine areas
❑ By line number	→**E**xplicitly	→**E**xplicitly	not applicable
❑ All lines in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All lines in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All lines everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Ranges	→**R**ange areas	→**R**ange areas	→**R**ange areas
❑ By line numbers	→**E**xplicitly	→**E**xplicitly	not applicable
❑ All ranges in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All ranges in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All ranges everywhere	→**G**lobally	→**G**lobally	→**G**lobally
Functions	→**F**unction areas	not applicable	→**F**unction areas
❑ By function name	→**E**xplicitly		not applicable
❑ All functions in a module	→in one **M**odule		→in one **M**odule
❑ All functions everywhere	→**G**lobally		→**G**lobally
All areas	→**R**ange areas	→**R**ange areas	→**R**ange areas
❑ All areas in a function	→in one **F**unction	→in one **F**unction	→in one **F**unction
❑ All areas in a module	→in one **M**odule	→in one **M**odule	→in one **M**odule
❑ All areas everywhere	→**G**lobally	→**G**lobally	→**G**lobally

Interpreting session data

General information about a profiling session is displayed in the COMMAND window during and after the session. This information identifies the starting and stopping points. It also lists statistics for three important areas:

- **Run cycles** shows the number of execution cycles consumed by the program from the starting point to the stopping point.
- **Profile cycles** equals the run cycles minus the cycles consumed by disabled areas.
- **Hits** shows the number of internal breakpoints encountered during the profiling session.

Viewing code associated with a profile area

You can view the code associated with a displayed profile area. The debugger will update the display so that the associated C or disassembly statements are shown in the FILE or DISASSEMBLY windows.

Use the mouse to select the profile area in the PROFILE window and display the associated code:

1) Point to the appropriate area name in the PROFILE window.

2) Click the right mouse button.

The area name and the associated C or disassembly statement will be highlighted. To view the code associated with another area, point and click again.

If you are attempting to show disassembly, you may have to make several attempts because program memory can be accessed only when the target is not running.

10.7 Saving Profile Data to a File

You may want to run several profiling sessions during a debugging session. Whenever you start a new profiling session,the results of the previous session are lost. However, you can save the results of the current profiling session to a system file. You can use two commands to do this:

vac To save the contents of the PROFILE window to a system file, use the VAC (view save current) command. The syntax for this command is:

vac *filename*

This saves only the current view; if, for example, you are viewing only the Count field, then only that information will be saved.

vaa To save all data for the currently displayed areas, use the VAA (view save all) command. The syntax for this command is:

vaa *filename*

This saves all views of the data—including the individual count, inclusive, etc.—with the percentage indications and histograms.

Both commands write profile data to *filename*. The filename can include path information. There is no default filename. If *filename* already exists, the command will overwrite the file with the new data.

Note that if the PROFILE window displays only a subset of the areas that are marked for profiling, data is saved *only for those areas that are displayed*. (For VAC, the currently displayed data will be saved for the displayed areas. For VAA, all data will be saved for the displayed areas.) If some areas are hidden and you want to save all the data, be sure to select View→Reset before saving the data to a file.

The file contents are in ASCII and are formatted in exactly the same manner as they are displayed (or would be displayed) in the PROFILE window. The general profiling-session information that is displayed in the COMMAND window is also written to the file.

Chapter 11

Summary of Commands and Special Keys

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

Topic **Page**

11.1 Functional Summary of Debugger Commands

This section summarizes the debugger commands according to these categories:

- **Changing modes.** These commands (listed on page 11-3) enable you to switch freely between the debugging modes (auto, mixed, minimal, and assembly).
- **Managing windows.** These commands (listed on page 11-3) enable you to select the active window and move or resize the active window.
- **Displaying and changing data.** These commands (listed on page 11-3) enable you to display and evaluate a variety of data items.
- **Performing system tasks.** These commands (listed on page 11-4) enable you to perform several DOS-like functions and provide you with some control over the target system.
- **Managing breakpoints.** These commands (listed on page 11-4) provide you with a command line method for controlling software breakpoints.
- **Displaying files and loading programs.** These commands (listed on page 11-5) enable 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 11-5) allow you to customize the debugger display, then save and later reuse the customized displays.
- **Memory mapping.** These commands (listed on page 11-5) enable you to define the areas of target memory that the debugger can access.
- **Running programs.** These commands (listed on page 11-6) provide you with a variety of methods for running your programs in the debugger environment.
- **Profiling commands.** These commands (listed on page 11-7) enable you to collect execution statistics for your code.

Changing modes

To put the debugger in	Use this command	See page
Assembly mode	asm	11-12
Auto mode for debugging C code	c	11-15
Minimal mode	minimal	11-27
Mixed mode	mix	11-27

Managing windows

To do this	Use this command	See page
Reposition the active window	move	11-28
Resize the active window	size	11-37
Select the active window	win	11-44
Make the active window as large as possible	zoom	11-45

Displaying and changing data

To do this	Use this command	See page
Evaluate and display the result of a C expression	?	11-10
Display the values in an array or structure or display the value that a pointer is pointing to	disp	11-18
Evaluate a C expression without displaying the results	eval	11-22
Display a different range of memory in the MEMORY window or display an additional MEMORY window	mem	11-26
Change the default format for displaying data values	setf	11-36
Continuously display the value of a variable, register, or memory location within the WATCH window	wa	11-42
Delete a data item from the WATCH window	wd	11-43
Show the type of a data item	whatis	11-43
Delete all data items from the WATCH window and close the WATCH window	wr	11-44

Performing system tasks

To do this	Use this command	See page
Define your own command string	alias	11-12
Clear all displayed information from the display area of the COMMAND window	cls	11-15
Record the information shown in the display area of the COMMAND window	dlog	11-20
Display a string to the COMMAND window while executing a batch file	echo	11-21
Conditionally execute debugger commands in a batch file	if/else/endif	11-23
Loop debugger commands in a batch file	loop/endloop	11-24
Exit the debugger	quit	11-32
Reset the target system	reset	11-32
Associate a beeping sound with the display of error messages	sound	11-38
Execute commands from a batch file	take	11-40
Delete an alias definition	unalias	11-40
Name additional directories that can be searched when you load source files	use	11-41

Managing breakpoints

To do this	Use this command	See page
Add a software breakpoint	ba	11-13
Delete a software breakpoint	bd	11-13
Display a list of all the software breakpoints that are set	bl	11-13
Reset (delete) all software breakpoints	br	11-14

Displaying files and loading programs

To do this	Use this command	See page
Display C and/or assembly language code at a specific point	addr	11-11
Reopen the CALLS window	calls	11-15
Display assembly language code at a specific address	dasm	11-18
Display a text file in the FILE window	file	11-22
Display a specific C function	func	11-23
Load an object file	load	11-24
Load only the object-code portion of an object file	reload	11-32
Load only the symbol-table portion of an object file	sload	11-38

Customizing the screen

To do this	Use this command	See page
Change the border style of any window	border	11-14
Change the screen colors, but don't update the screen immediately	color	11-16
Change the command-line prompt	prompt	11-31
Change the screen colors and update the screen immediately	scolor	11-34
Load and use a previously saved custom screen configuration	sconfig	11-35
Save a custom screen configuration	ssave	11-39

Memory mapping

To do this	Use this command	See page
Initialize a block of memory	fill	11-22
Add an address range to the memory map	ma	11-25
Enable or disable memory mapping	map	11-25
Delete an address range from the memory map	md	11-26
Display a list of the current memory map settings	ml	11-27
Reset the memory map (delete all ranges)	mr	11-29
Save a block of memory to a system file	ms	11-29

Running programs

To do this	Use this command	See page
Single-step through assembly language or C code, one C statement at a time; step over function calls	cnext	11-16
Single-step through assembly language or C code, one C statement at a time	cstep	11-17
Run a program up to a certain point	go	11-23
Single-step through assembly language or C code; step over function calls	next	11-29
Reset the target system	reset	11-32
Reset the program entry point	restart	11-32
Execute code in a function and return to the function's caller	return	11-33
Run a program	run	11-33
Single-step through assembly language or C code	step	11-39
Execute commands from a batch file	take	11-40

Profiling commands

All of the profiling commands can be entered from the pulldown menus. In many cases, using the pulldown menus is the easiest way to use some of these commands. For this reason and also because there are over 100 profiling commands, most of these commands are not described individually in this chapter (as the basic debugger commands are).

Listed below are some of the profiling commands that you might choose to enter from the command line instead of from a menu; these commands are also described in the alphabetical command summary. The remaining profiling commands are summarized in Section 11.4, *Summary of Profiling Commands*, on page 11-46.

To do this	Use this command	See page
Run a full profiling session	pf	11-30
Run a quick profiling session	pq	11-31
Resume a profiling session	pr	11-31
Add a stopping point	sa	11-33
Delete a stopping point	sd	11-35
List all the stopping points	sl	11-37
Delete all the stopping points	sr	11-38
Save all the profile data to a file	vaa	11-41
Save currently displayed profile data to a file	vac	11-41
Reset the display in the PROFILE window to show all areas and the default set of data	vr	11-42

11.2 How the Menu Selections Correspond to Commands

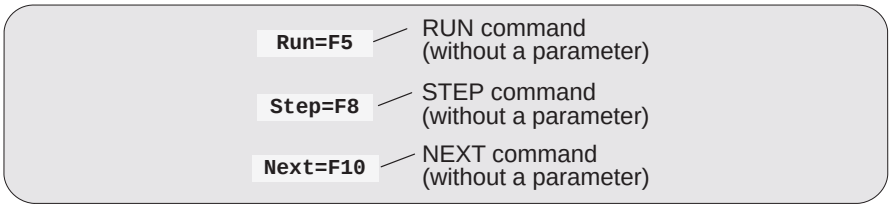
The following sample screens illustrate the relationship of the basic debugger commands to the menu bar and pulldown menus.

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

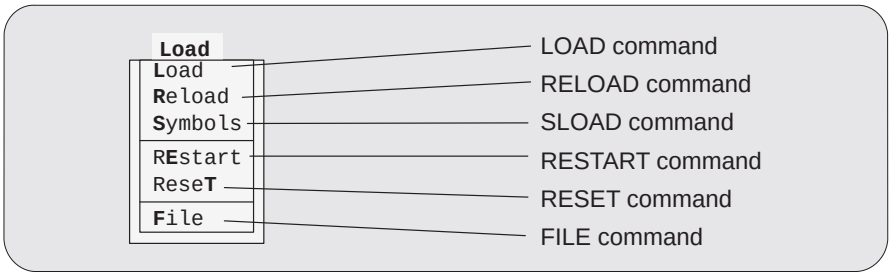
Note:

Because the profiling environment supports over 100 profile-specific commands, it's not practical to show the commands associated with the profile menu choices.

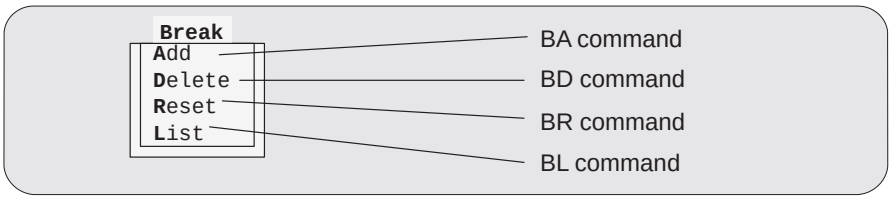
Program-execution commands



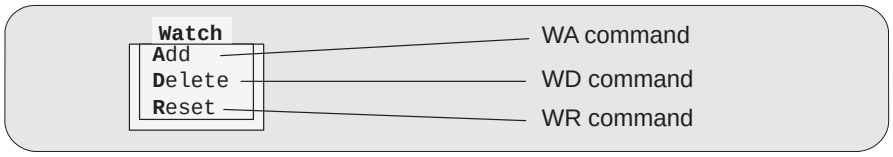
File/load commands



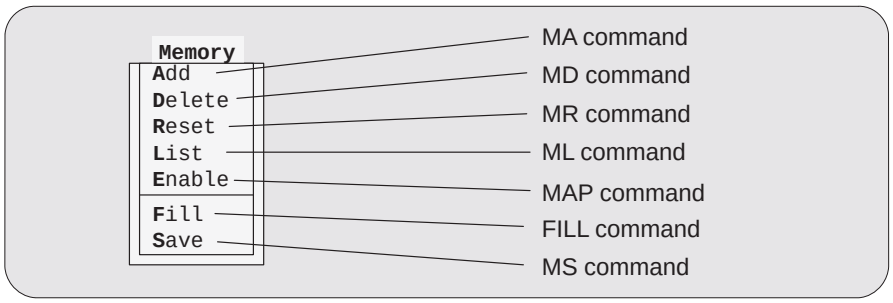
Breakpoint commands



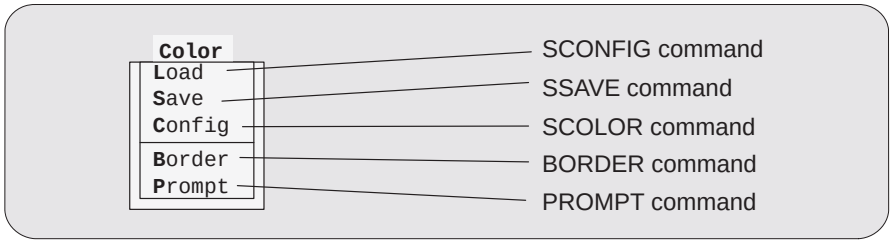
Watch commands



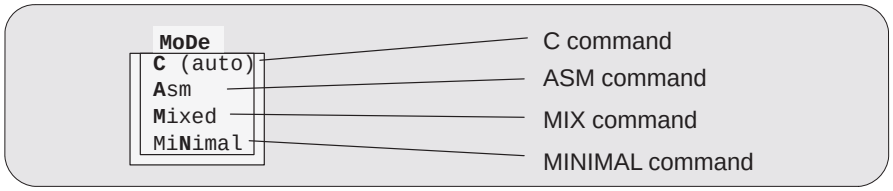
Memory commands



Screen-configuration commands



Mode commands



11.3 Alphabetical Summary of Debugger Commands

Commands are not case sensitive; to emphasize this, command names are shown in both uppercase and lowercase throughout this book.

? — *Evaluate Expression*

Syntax **?** *expression* [*, display format*]

Menu selection none

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

addr — *Display Code at Specified Address*

Syntax

addr *address*
addr *function name*

Menu selection

none

Environments

☑ basic debugger ☐ profiling

Description

Use the ADDR command to display C code or the disassembly at a specific point. ADDR's behavior changes, depending on the current debugging mode:

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

Note:

ADDR affects the FILE window only if the specified *address* is in a C function.

alias — *Define Custom Command String*

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

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description You can use the ALIAS command to associate one or more debugger commands with a single *alias name.*

You can include as many commands in the *command string* as you like, as long you separate them with semicolons and enclose the entire string of commands in quotation marks. You can also identify command parameters by a percent sign followed by a number (%1, %2, etc.). The total number of characters for an individual command (expanded to include parameter values) is limited to 132 (this restriction applies to the debugger version of the ALIAS command only).

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

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

asm — *Enter Assembly Mode*

Syntax **asm**

Menu selection Mo**D**e→**A**sm

Environments ☑ basic debugger ☐ profiling

Description The ASM command changes from the current debugging mode to assembly mode. If you're already in assembly mode, the ASM command has no effect.

ba — *Add Software Breakpoint*

Syntax **ba** *address*

Menu selection **B**reak→**A**dd

Environments ☑ basic debugger ☐ profiling

Description The BA command sets a software breakpoint at a specific *address*. This command is useful because it doesn't require you to search through code to find the desired line. The *address* can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label.

Breakpoints can be set in program memory (RAM) only; the *address* parameter is treated as a program-memory address.

bd — *Delete Software Breakpoint*

Syntax **bd** *address*

Menu selection **B**reak→ **D**elete

Environments ☑ basic debugger ☐ profiling

Description The BD command clears a software breakpoint at a specific *address*. The *address* can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label.

bl — *List Software Breakpoints*

Syntax **bl**

Menu selection **B**reak→**L**ist

Environments ☑ basic debugger ☐ profiling

Description The BL command provides an easy way to get a complete listing of all the software breakpoints that are currently set in your program. It displays a table of breakpoints in the display area of the COMMAND window. BL lists all the breakpoints that are set, in the order in which you set them.

border — *Change Style of Window Border*

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

Menu selection **C**olor→**B**order

Environments ☑ basic debugger ☐ profiling

Description The BORDER command changes the border style of the active window, the inactive windows, and the border style of any window that you're resizing. The debugger supports nine border styles. Each parameter for the BORDER command must be one of the numbers that identify these styles:

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

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

br — *Reset Software Breakpoint*

Syntax **br**

Menu selection **B**reak→**R**eset

Environments ☑ basic debugger ☐ profiling

Description The BR command clears all software breakpoints that are set.

c — *Enter Auto Mode*

Syntax **c**

Menu selection Mo**D**e→**C** (auto)

Environments ☑ basic debugger ☐ profiling

Description The C command changes from the current debugging mode to auto mode. If you're already in auto mode, the C command has no effect.

calls — *Open CALLS Window*

Syntax **calls**

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The CALLS command displays the CALLS window. The debugger displays this window automatically when you are in auto/C or mixed mode. However, you can close the CALLS window; the CALLS command opens the window again.

cls — *Clear Screen*

Syntax **cls**

Menu selection none

Environments ☑ basic debugger ☑ 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 Next=**F10** (in C code)

Environments ☑ basic debugger ☐ profiling

Description The CNEXT command is similar to the CSTEP command. It runs a program one C statement at a time, updating the display after executing each statement. If you're using CNEXT to step through assembly language code, the debugger won't update the display until it has executed all assembly language statements associated with a single C statement. Unlike CSTEP, CNEXT steps over function calls rather than stepping into them—you don't see the single-step execution of the function call.

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page 6-14, discusses this in detail).

color — *Change Screen Colors*

Syntax **color** *area name, attribute*$_1$ [,*attribute*$_2$ [,*attribute*$_3$ [,*attribute*$_4$]]]

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The COLOR command changes the color of specified areas of the debugger display. COLOR doesn't update the display; the changes take effect when another command, such as SCOLOR, updates the display. The *area name* parameter identifies the areas of the display that are affected. The *attributes* identify how the areas are affected. The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Valid values for the *attribute* parameters include:

black	blue	green	cyan
red	magenta	yellow	white
bright		blink	

Valid values for the *area name* parameters include:

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

You don't have to type an entire *attribute* or *area name*; you need to type only enough letters to uniquely identify the attribute. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they're listed above (left to right, top to bottom).

cstep — *Single-Step C*

Syntax **cstep** [*expression*]

Menu selection Step=**F8** (in C code)

Environments ☑ basic debugger ☐ profiling

Description The CSTEP single-steps through a program one C statement at a time, updating the display after executing each statement. If you're using CSTEP to step through assembly language code, the debugger won't update the display until it has executed all assembly language statements associated with a single C statement.

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

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page 6-14, discusses this in detail).

dasm — *Display Disassembly at Specific Address*

Syntax

dasm *address*
dasm *function name*

Menu selection

none

Environments

☑ basic debugger ☑ profiling

Description

The DASM command displays code beginning at a specific point within the DISASSEMBLY window.

disp — *Open DISP Window*

Syntax

disp *expression* [, *display format*]

Menu selection

none

Environments

☑ basic debugger ☐ profiling

Description

The DISP command opens a DISP window to display the contents of an array, structure, or pointer expressions to a scalar type (of the form **pointer*). If the *expression* is not one of these types, then DISP acts like a ? command.

Once you open a DISP window, you may find that a displayed member is itself an array, structure, or pointer:

A member that is an array looks like this	[. . .]
A member that is a structure looks like this	{. . .}
A member that is a pointer looks like an address	0x0000

You can display the additional data (the data pointed to or the members of the array or structure) in another DISP window by using the DISP command again, using the arrow keys to select the field and then pressing F9, or pointing the mouse cursor to the field and pressing the left mouse button. You can have up to 120 DISP windows open at the same time.

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

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

The *display format* parameter can be used only when you are displaying a scalar type, an array of scalar type, or an individual member of an aggregate type.

You can also use the DISP command with a typecast expression to display memory contents in any format. Here are some examples:

```
disp *0
disp *(float *)123
disp *(char *)0x111
```

This shows memory in the DISP window as an array of locations; the location that you specify with the *expression* parameter is member [0], and all other locations are offset from that location.

dlog — *Record Display Window*

Syntax

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

Menu selection

none

Environments

☑ basic debugger ☑ profiling

Description

The DLOG command allows you to record the information displayed in the COMMAND window into a log file.

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

dlog *filename*

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

❑ To end the recording session, enter:

dlog close ↵

If necessary, you can write over existing log files or append additional information to existing files. The optional parameters of the DLOG command control how existing log files are used:

❑ **Appending to an existing file.** Use the **a** parameter to open an existing file to which to append the information in the display area.

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

echo — *Echo String to Display Area*

Syntax **echo** *string*

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The ECHO command displays *string* in the display area of the COMMAND window. You can't 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 ECHO command only in a batch file.

else — *Execute Alternative Commands*

Description ELSE provides an alternative list of commands in the IF/ELSE/ENDIF command sequence. See page 11-23 for more information about these commands.

endif — *Terminate Conditional Sequence*

Description ENDIF identifies the end of a conditional-execution command sequence begun with an IF command. See page 11-23 for more information about these commands.

endloop — *Terminate Looping Sequence*

Description ENDLOOP identifies the end of the LOOP/ENDLOOP command sequence. See page 11-24 for more information about the LOOP/ENDLOOP commands.

eval — *Evaluate Expression*

Syntax

eval *expression*
e *expression*

Menu selection

none

Environments

☑ basic debugger ☑ profiling

Description

The EVAL command evaluates an expression like the ? command does *but does not show the result* in the display area of the COMMAND window. EVAL is useful for assigning values to registers or memory locations in a batch file (where it's not necessary to display the result).

file — *Display Text File*

Syntax

file *filename*

Menu selection

Load→**F**ile

Environments

☑ basic debugger ☑ profiling

Description

The FILE command displays the contents of any text file in the FILE window. The debugger continues to display this file until you run a program and halt in a C function. This command is intended primarily for displaying C source code. You can view only one text file at a time.

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

fill — *Fill Memory*

Syntax

fill *address, length, data*

Menu selection

Memory→**F**ill

Environments

☑ basic debugger ☐ profiling

Description

The FILL command fills a block of memory with a specified value.

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

func — *Display Function*

Syntax

func *function name*
func *address*

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description

The FUNC command displays a specified C function in the FILE window. You can identify the function by its name or its address. Note that FUNC works the same way FILE works, but with FUNC you don't need to identify the name of the file that contains the function.

go — *Run to Specified Address*

Syntax

go [*address*]

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description

The GO command executes code up to a specific point in your program. If you don't supply an *address*, then GO acts like a RUN command without an *expression* parameter.

if/else/endif — *Conditionally Execute Debugger Commands*

Syntax

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

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description

These commands allow you to execute debugger commands conditionally in a batch file. If the *expression* if nonzero, the debugger executes the commands between the IF and the ELSE or ENDIF. Note that the ELSE portion of the command sequence is optional.

The conditional commands work with the following provisions:

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

load — *Load Executable Object File*

Syntax **load** *object filename*

Menu selection **L**oad→ **L**oad

Environments ☑ basic debugger ☑ profiling

Description The LOAD command loads both an object file and its associated symbol table into memory. In effect, the LOAD command performs both a RELOAD and an SLOAD. If you don't supply an extension, the debugger looks for *filename*.out. Note that the LOAD command clears the old symbol table and closes the WATCH and DISP windows.

loop/endloop — *Loop Through Debugger Commands*

Syntax

loop *expression*
debugger commands
endloop

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The LOOP/ENDLOOP commands allow you to set up a looping situation in a batch file. These looping commands evaluate in the same method as in the run conditional command expression:

- If you use an *expression* that is not Boolean, the debugger evaluates the expression as a loop count.
- If you use a Boolean *expression*, the debugger executes the command repeatedly as long as the expression is true.

The LOOP/ENDLOOP commands work under the following conditions:

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

ma — *Add Block to Memory Map*

Syntax **ma** *address, length, type*

Menu selection **M**emory→**A**dd

Environments ☑ basic debugger ☑ profiling

Description The MA command identifies valid ranges of target memory. Note that a new memory map must not overlap an existing entry; if you define a range that overlaps an existing range, the debugger ignores the new range.

- The *address* parameter defines the starting address of a range in memory. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.
- The *length* parameter defines the length of the range. This parameter can be any C expression.
- The *type* parameter identifies the read/write characteristics of the memory range. The *type* must be one of these keywords:

To identify this kind of memory,	Use this keyword as the *type* parameter
Read-only memory	**R**, **ROM**, or **READONLY**
Write-only memory	**W**, **WOM**, or **WRITEONLY**
Read/write memory	**WR** or **RAM**
No-access memory	**PROTECT**
Input port	**INPORT** or **P|R**
Output port	**OUTPORT** or **P|W**
Input/output port	**IOPORT** or **P|R|W**

map — *Enable Memory Mapping*

Syntax **map** {**on** | **off**}

Menu selection **M**emory→**E**nable

Environments ☑ basic debugger ☑ profiling

Description The MAP command enables or disables memory mapping. In some instances, you may want to explicitly enable or disable memory. Note that disabling memory mapping can cause bus fault problems in the target because the debugger may attempt to access nonexistent memory.

md — *Delete Block From Memory Map*

Syntax **md** *address*

Menu selection **M**emory→**D**elete

Environments ☑ basic debugger ☑ profiling

Description The MD command deletes a range of memory from the debugger's memory map.

The *address* parameter identifies the starting address of the range of memory. 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
```

mem — *Modify MEMORY Window Display*

Syntax **mem** *expression* [, [*display format*] [, *window name*]]

Menu selection none

Environments ☑ basic debugger ☐ 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.

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

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

minimal — *Enter Minimal Mode*

Syntax **minimal**

Menu selection Mo**D**e→Mi**N**imal

Environments ☑ basic debugger ☐ profiling

Description The MINIMAL command changes from the current debugging mode to minimal mode. If you're already in minimal mode, the MINIMAL command has no effect.

mix — *Enter Mixed Mode*

Syntax **mix**

Menu selection Mo**D**e→**M**ixed

Environments ☑ basic debugger ☐ profiling

Description The MIX command changes from the current debugging mode to mixed mode. If you're already in mixed mode, the MIX command has no effect.

ml — *List Memory Map*

Syntax **ml**

Menu selection **M**emory→**L**ist

Environments ☑ basic debugger ☑ profiling

Description The ML command lists the memory ranges that are defined for the debugger's memory map. The ML command lists the starting address, ending address, and read/write characteristics of each defined memory range.

move — *Move Active Window*

Syntax **move** [*X position*, *Y position* [, *width*, *length*]]

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description

The MOVE command moves the active window to the specified XY position. If you choose, you can resize the window while you move it (see the SIZE command for valid *width* and *length* values). You can use the MOVE command in one of two ways:

- By supplying a specific *X position* and *Y position* or
- By omitting the *X position* and *Y position* parameters and using function keys to interactively move the window.

You can move a window by defining a new XY position for the window's upper left corner. Valid X and Y positions depend on the screen size and the window size. X positions are valid if the X position plus the window width in characters is less than or equal to the screen width in characters. Y positions are valid if the Y position plus the window height is less than or equal to the screen height in lines.

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

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

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

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

mr — *Reset Memory Map*

Syntax **mr**

Menu selection **M**emory→**R**eset

Environments ☑ basic debugger ☑ profiling

Description The MR command resets the debugger's memory map by deleting all defined memory ranges from the map.

ms — *Save Memory Block to File*

Syntax **ms** *address, length, filename*

Menu selection **M**emory→**S**ave

Environments ☑ basic debugger ☑ profiling

Description The MS command saves the values in a block of memory to a system file; files are saved in COFF format.

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

next — *Single-Step, Next Statement*

Syntax **next** [*expression*]

Menu selection Next=**F10** (in disassembly)

Environments ☑ basic debugger ☐ profiling

Description The NEXT command is similar to the STEP command. If you're in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time. Unlike STEP, NEXT never updates the display when executing called functions; NEXT always steps to the next consecutive statement. Unlike STEP, NEXT steps over function calls rather than stepping into them—you don't see the single-step execution of the function call.

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page 6-14, discusses this in detail).

pause — *Pause Execution*

Syntax **pause**

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The PAUSE command allows you to pause the debugger while running a batch file. Pausing is especially helpful in debugging the commands in a batch file.

When the debugger reads this command in a batch file, the debugger stops execution and displays the following message:

```
<< pause – type return >>
```

To continue processing, press ↵.

pf — *Profile, Full*

Syntax **pf** *starting point* [, *update rate*]

Menu selection **P**rofile→**F**ull

Environments ☐ basic debugger ☑ profiling

Description The PF command initiates a RUN and collects a full set of statistics on the defined areas between the *starting point* and the first-encountered stopping point. The *starting point* parameter can be a label, a function name, or a memory address.

The optional *update rate* parameter determines how often the PROFILE window will be updated. The *update rate* parameter can have one of these values:

Value	Description
0	This is the default. Statistics are not updated until the session is halted (although you can force an update by clicking the mouse in the window header). A "spinning wheel" character is shown to indicate that a profiling session is in progress.
≥1	Statistics are updated during the session. A value of **1** means that data is updated as often as possible.
<0	Statistics are not updated until the profiling session is halted, and the "spinning wheel" character is not displayed.

pq — *Profile, Quick*

Syntax **pq** *starting point* [, *update rate*]

Menu selection **P**rofile→**Q**uick

Environments ☐ basic debugger ☑ profiling

Description The PQ command initiates a RUN command and collects a subset of the available statistics on the defined areas between the *starting point* and the first-encountered stopping point. PQ is similar to PF, except that PQ doesn't collect exclusive or exclusive max data.

The *update rate* parameter is the same as for the PF command.

pr — *Resume Profiling Session*

Syntax **pr** [*clear data* [, *update rate*]]

Menu selection **P**rofile→**R**esume

Environments ☐ basic debugger ☑ profiling

Description The PR command resumes the last profiling session (initiated by PF or PQ), starting from the current program counter.

The optional *clear data* parameter tells the debugger whether or not it should clear out the previously collected data. The *clear data* parameter can have one of these values:

Value	Description
0	This is the default. The profiler will continue to collect data, adding it to the existing data for the profiled areas, and to use the previous internal profile stacks.
nonzero	All previously collected profile data and internal profile stacks are cleared.

The *update rate* parameter is the same as for the PF and PQ commands.

prompt — *Change Command-Line Prompt*

Syntax **prompt** *new prompt*

Menu selection **C**olor→**P**rompt

Environments ☑ basic debugger ☑ profiling

Description The PROMPT command changes the command-line prompt. The *new prompt* can be any string of characters (note that a semicolon or comma ends the string).

quit — *Exit Debugger*

Syntax **quit**

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The QUIT command exits the debugger and returns to the operating system.

reload — *Reload Object Code*

Syntax **reload** [*object filename*]

Menu selection **L**oad→**R**eload

Environments ☑ basic debugger ☑ profiling

Description The RELOAD command loads only an object file *without* loading its associated symbol table. This is useful for reloading a program when target memory has been corrupted. If you enter the RELOAD command without specifying a filename, the debugger reloads the file that you loaded last.

reset — *Reset Target System*

Syntax **reset**

Menu selection **L**oad→**R**eseT

Environments ☑ basic debugger ☑ profiling

Description The RESET command resets the simulator and reloads the monitor. Note that this is a *software* reset.

If you execute the RESET command, the simulator simulates the ’C6x processor and peripheral reset operation, putting the processor in a known state.

restart — *Reset PC to Program Entry Point*

Syntax **restart**
rest

Menu selection **L**oad→R**E**start

Environments ☑ basic debugger ☑ profiling

Description The RESTART or REST command resets the program to its entry point. (This assumes that you have already used one of the load commands to load a program into memory.)

return — *Return to Function's Caller*

Syntax **return**
ret

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The RETURN or RET command executes the code in the current C function and halts when execution reaches the caller. Breakpoints do not affect this command, but you can halt execution by pressing the left mouse button or pressing ESC.

run — *Run Code*

Syntax **run** [*expression*]

Menu selection Run=**F5**

Environments ☑ basic debugger ☐ profiling

Description The RUN command is the basic command for running an entire program. The command's behavior depends on the type of parameter you supply:

- If you don't supply an *expression*, the program executes until it encounters a breakpoint or until you press the left mouse button or press ESC.
- If you supply a logical or relational *expression*, the run becomes conditional (*Running code conditionally*, page 6-14, 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.

sa — *Add Stopping Point*

Syntax **sa** *address*

Menu selection **S**top-points→**A**dd

Environments ☐ basic debugger ☑ profiling

Description The SA command adds a stopping point at *address*. The *address* can be a label, a function name, or a memory address.

safehalt — *Toggle Safehalt Mode*

Syntax **safehalt** {**on** | **off**}

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The SAFEHALT command places the debugger in safehalt mode. When safehalt mode is off (the default), you can halt a running target device either by pressing ESC or by clicking a mouse button. When safehalt mode is on, you can halt a running target device only by pressing ESC; mouse clicks are ignored.

scolor — *Change Screen Colors*

Syntax **scolor** *area name*, *attribute*$_1$ [, *attribute*$_2$ [, *attribute*$_3$ [, *attribute*$_4$]]]

Menu selection **C**olor→**C**onfig

Environments ☑ basic debugger ☐ profiling

Description The SCOLOR command changes the color of specified areas of the debugger display and updates the display immediately. The *area name* parameter identifies the area of the display that is affected. The *attributes* identify how the area is affected. The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Valid values for the *attribute* parameters include:

black	blue	green	cyan
red	magenta	yellow	white
bright		blink	

Valid values for the *area name* parameters include:

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

You don't have to type an entire *attribute* or *area name*; you need to type only enough letters to uniquely identify the attribute. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they're listed above (left to right, top to bottom).

sconfig — *Load Screen Configuration*

Syntax **sconfig** [*filename*]

Menu selection **C**olor→**L**oad

Environments ☑ basic debugger ☑ profiling

Description The SCONFIG command restores the display to a specified configuration. This restores the screen colors, window positions, window sizes, and border styles that were saved with the SSAVE command into *filename*. If you don't supply a *filename*, the debugger looks for init.clr. The debugger searches for the specified file in the current directory and then in directories named with the D_DIR environment variable.

When you use SCONFIG to restore a configuration that includes multiple WATCH or MEMORY windows, the additional windows are not launched automatically. However, when you launch an additional window using the same name as before you saved the configuration, the window is placed in the correct location.

sd — *Delete Stopping Point*

Syntax **sd** *address*

Menu selection **S**top-points→**D**elete

Environments ☐ basic debugger ☑ profiling

Description The SD command deletes the stopping point at *address*.

setf — *Set Default Data-Display Format*

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

Menu selection none

Environments ☑ basic debugger ☐ profiling

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

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

char	short	uint	ulong	double
uchar	int	long	float	ptr

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

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

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

	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 Active Window*

Syntax **size** [*width*, *length*]

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The SIZE command changes the size of the active window. You can use the SIZE command in one of two ways:

- By supplying a specific *width* and *length* or
- By omitting the *width* and *length* parameters and using function keys to interactively resize the window.

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

If a window is in the middle of the display, you can't size it to the maximum height and width—you can size it only to the right and bottom screen borders. The easiest way to make a window as large as possible is to zoom it, see *Zooming a window* on page 3-26.

If you enter the SIZE command without *width* and *length* parameters, you can use arrow keys to size the window.

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

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

sl — *List Stopping Point*

Syntax **sl**

Menu selection **S**top-points→**L**ist

Environments ☐ basic debugger ☑ profiling

Description The SL command lists all of the currently set stopping points.

sload — *Load Symbol Table*

Syntax **sload** *object filename*

Menu selection **L**oad→**S**ymbols

Environments ☑ basic debugger ☑ profiling

Description The SLOAD command loads the symbol table of the specified object file. SLOAD is useful in a debugging environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). SLOAD clears the existing symbol table before loading the new one but does not modify memory or set the program entry point. Note that SLOAD closes the WATCH and DISP windows.

sound — *Enable Error Beeping*

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

Menu selection none

Environments ☑ basic debugger ☐ profiling

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

sr — *Reset Stopping Point*

Syntax **sr**

Menu selection **S**top-points→**R**eset

Environments ☐ basic debugger ☑ profiling

Description The SR command resets (deletes) *all* currently set stopping points.

ssave — *Save Screen Configuration*

Syntax **ssave** [*filename*]

Menu selection **C**olor→**S**ave

Environments ☑ basic debugger ☑ profiling

Description The SSAVE command saves the current screen configuration to a file. This saves the screen colors, window positions, window sizes, and border styles. SSAVE also saves the location of multiple WATCH and MEMORY windows. The *filename* parameter names the new screen configuration file. You can include path information (including relative pathnames); if you don't supply path information, the debugger places the file in the current directory. If you don't supply a *filename*, the debugger saves the current configuration into a file named init.clr and places the file in the current directory.

step — *Single-Step*

Syntax **step** [*expression*]

Menu selection Step=**F8** (in disassembly)

Environments ☑ basic debugger ☐ profiling

Description The STEP command single-steps through assembly language or C code. If you're in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time.

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

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page 6-14, discusses this in detail).

take — *Execute Batch File*

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

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The TAKE command tells the debugger to read and execute commands from a batch file. The *batch filename* parameter identifies the file that contains commands. If you don't supply a pathname as part of the filename, the debugger first looks in the current directory and then searches directories named with the D_DIR environment variable.

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. For the debugger, you can change this behavior:

- If you don't use the *suppress echo flag* parameter, or if you use it but supply a nonzero value, then the debugger behaves in the default manner.
- If you would like to suppress the echoing and updating, use the value 0 for the *suppress echo flag* parameter.

unalias — *Delete Alias Definition*

Syntax **unalias** *alias name*
unalias *****

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The UNALIAS command deletes defined aliases.

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

  `unalias NEWMAP` ↵

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

  `unalias *` ↵

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

use — *Use New Directory*

Syntax **use** [*directory name*]

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The USE command allows you to name an additional directory that the debugger can search when looking for source files. You can specify only one directory at a time.

If you enter the USE command without specifying a directory name, the debugger lists all of the current directories.

vaa — *Save All Profile Data to a File*

Syntax **vaa** *filename*

Menu selection **V**iew→S**a**ve→**A**ll views

Environments ☐ basic debugger ☑ profiling

Description The VAA command saves all statistics collected during the current profiling session. The data is stored in a system file.

vac — *Save Displayed Profile Data to a File*

Syntax **vac** *filename*

Menu selection **V**iew→S**a**ve→**C**urrent view

Environments ☐ basic debugger ☑ profiling

Description The VAC command saves all statistics currently displayed in the PROFILE window. (Statistics that aren't displayed aren't saved.) The data is stored in a system file.

version — *Display the Current Debugger Version*

Syntax **version**

Menu selection none

Environments ☑ basic debugger ☑ profiling

Description The VERSION command displays the debugger's copyright date and the current version number of the debugger, silicon, etc.

vr — *Reset PROFILE Window Display*

Syntax **vr**

Menu selection **V**iew→**R**eset

Environments ☐ basic debugger ☑ profiling

Description The VR command resets the display in the PROFILE window so that all marked areas are listed and statistics are displayed with default labels and in the default sort order.

wa — *Add Item to WATCH Window*

Syntax **wa** *expression* [,[*label*] [, [*display format*] [, *window name*]]]

Menu selection **W**atch→**A**dd

Environments ☑ basic debugger ☐ profiling

Description The WA command displays the value of *expression* in a WATCH window. If a WATCH window isn't open, executing WA opens a WATCH window. The *expression* parameter can be any C expression, including an expression that has side effects.

WA is most useful for watching an expression whose value changes over time; constant expressions serve no useful function in the watch window. The *label* parameter is optional. When used, it provides a label for the watched entry. If you don't use a *label*, the debugger displays the *expression* in the label field.

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

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

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

```
wa PC,,d ↵
```

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** *index number* [, *window name*]

Menu selection **W**atch→**D**elete

Environments ☑ basic debugger ☐ profiling

Description The WD command deletes a specific item from the WATCH window. The WD command's *index number* parameter must correspond to one of the watch indexes listed in the WATCH window. The optional *window name* parameter is used to specify a particular WATCH window.

whatis — *Find Data Type*

Syntax **whatis** *symbol*

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The WHATIS command shows the data type of *symbol* in the 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 — *Select Active Window*

Syntax **win** *WINDOW NAME*

Menu selection none

Environments ☑ basic debugger ☑ profiling

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

If several windows of the same type are visible on the screen, don't use the WIN command to select one of them. If you supply an ambiguous name (such as C, which could stand for CPU or CALLS), the debugger selects the first window it finds whose name matches the name you supplied. If the debugger doesn't find the window you asked for (because you closed the window or misspelled the name), then the WIN command has no effect.

wr — *Close WATCH Window*

Syntax **wr** [{***** | *window name}*]

Menu selection **W**atch→**R**eset

Environments ☑ basic debugger ☐ 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 *windowname*

- To close all WATCH windows, enter:

  wr * ↵

zoom — *Zoom Active Window*

Syntax **zoom**

Menu selection none

Environments ☑ basic debugger ☑ profiling

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

11.4 Summary of Profiling Commands

The following tables summarize the profiling commands that are used for marking, enabling, disabling, and unmarking areas and for changing the display in the PROFILE window. These commands are easiest to use from the pulldown menus, so they are not included in the alphabetical command summary. The syntaxes for these commands are provided here so that you can include them in batch files.

Table 11–1. Marking Areas

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

Table 11–2. Disabling Marked Areas

To disable this area	C only	Disassembly only	C *and* disassembly
Lines			
❑ By line number, address	**DCLE** *filename*, *line number*	**DALE** *address*	not applicable
❑ All lines in a function	**DCLF** *function*	**DALF** *function*	**DBLF** *function*
❑ All lines in a module	**DCLM** *filename*	**DALM** *filename*	**DBLM** *filename*
❑ All lines everywhere	**DCLG**	**DALG**	**DBLG**
Ranges			
❑ By line numbers, addresses	**DCRE** *filename*, *line number*	**DARE** *address*	not applicable
❑ All ranges in a function	**DCRF** *function*	**DARF** *function*	**DBRF** *function*
❑ All ranges in a module	**DCRM** *filename*	**DARM** *filename*	**DBRM** *filename*
❑ All ranges everywhere	**DCRG**	**DARG**	**DBRG**

Table 11–2. Disabling Marked Areas (Continued)

To disable this area	C only	Disassembly only	C *and* disassembly
Functions			
❑ By function name	**DCFE** *function*	not applicable	not applicable
❑ All functions in a module	**DCFM** *filename*		**DBFM** *filename*
❑ All functions everywhere	**DCFG**		**DBFG**
All areas			
❑ All areas in a function	**DCAF** *function*	**DAAF** *function*	**DBAF** *function*
❑ All areas in a module	**DCAM** *filename*	**DAAM** *filename*	**DBAM** *filename*
❑ All areas everywhere	**DCAG**	**DAAG**	**DBAG**

Table 11–3. Enabling Disabled Areas

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

Table 11–4. Unmarking Areas

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

Table 11–5. Changing the PROFILE Window Display

(a) Viewing specific areas

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

(b) Viewing different data

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

(c) Sorting the data

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

11.5 Summary of Special Keys

The debugger provides function key, cursor key, and command key sequences for performing a variety of actions:

- Editing text on the command line
- Using the command history
- Switching modes
- Halting or escaping from an action
- Displaying the pulldown menus
- Running code
- Selecting or closing a window
- Moving or sizing a window
- Scrolling through a window's contents
- Editing data or selecting the active field

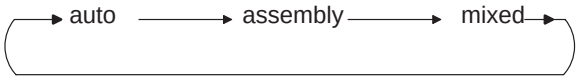
Editing text on the command line

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

Using the command history

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

Switching modes

To do this	Use these function keys
Switch debugging modes in this order:  	F3

Halting or escaping from an action

The escape key acts as an end or undo key in several situations.

To do this	Use this function key
Halt program execution	ESC
Close a pulldown menu	
Undo an edit of the active field in a data-display window (pressing this key leaves the field unchanged)	
Halt the display of a long list of data in the display area of the COMMAND window	

Displaying pulldown menus

To do this	Use these function keys
Display the Load menu	ALT L
Display the Break menu	ALT B
Display the Watch menu	ALT W
Display the Memory menu	ALT M
Display the Color menu	ALT C
Display the MoDe menu	ALT D
Display an adjacent menu	← or →
Execute any of the choices from a displayed pulldown menu	Press the highlighted letter corresponding to your choice

Running code

To do this	Use these function keys
Run code from the current PC (equivalent to the RUN command without an *expression* parameter)	F5
Single-step code from the current PC (equivalent to the STEP command without an *expression* parameter)	F8
Single-step code from the current PC; step over function calls (equivalent to the NEXT command without an *expression* parameter)	F10

Selecting or closing a window

To do this	Use these function keys
Select the active window (pressing this key makes each window active in turn; stop pressing the key when the desired window becomes active)	F6
Close the CALLS, WATCH, DISP, or additional MEMORY window (the window must be active before you can close it)	F4

Moving or sizing a window

You can use the arrow keys to interactively move a window after entering the MOVE or SIZE command without parameters.

To do this	Use these function keys
Move the window down one line Make the window one line longer	↓
Move the window up one line Make the window one line shorter	↑
Move the window left one character position Make the window one character narrower	←
Move the window right one character position Make the window one character wider	→

Scrolling a window's contents

These descriptions and instructions for scrolling apply to the active window. Some of these descriptions refer to specific windows; if no specific window is named, then the description/instructions refer to any window that is active.

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

Editing data or selecting the active field

The F9 function key makes the current field (the field that the cursor is pointing to) active. This has various effects, depending on the field.

To do this	Use this function key
FILE or DISASSEMBLY window: Set or clear a breakpoint	F9
CALLS window: Display the source to a listed function	
Any data-display window: Edit the contents of the current field	
DISP window: Open an additional DISP window to display a member that is an array, structure, or pointer	

Chapter 12

Basic Information About C Expressions

Many of the debugger commands take C expressions as parameters. This allows the debugger to have a relatively small, yet powerful, instruction set. Because C expressions can have side effects—that is, the evaluation of some types of expressions can affect existing values—you can use the same command to display or to change a value. This reduces the number of commands in the command set.

This chapter contains basic information that you'll need to know in order to use C expressions as debugger command parameters.

Topic **Page**

12.1 C Expressions for Assembly Language Programmers

It's not necessary for you to be an experienced C programmer in order to use the debugger. However, in order to use the debugger's full capabilities, you should be familiar with the rules governing C expressions. You should obtain a copy of *The C Programming Language* (first or second edition) by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey. This book is referred to in the C community, and in Texas Instruments documentation, as *K&R*.

Note:

A single value or symbol is a legal C expression.

K&R contains a complete description of C expressions; to get you started, here's a summary of the operators that you can use in expression parameters.

- **Reference operators**

–>	indirect structure reference	.	direct structure reference
[ ]	array reference	*	indirection (unary)
&	address (unary)		

- **Arithmetic operators**

+	addition (binary)	–	subtraction (binary)
*	multiplication	/	division
%	modulo	–	negation (unary)
(*type*)	typecast		

- **Relational and logical operators**

>	greater than	>=	greater than or equal to
<	less than	<=	less than or equal to
==	is equal to	!=	is not equal to
&&	logical AND	||	logical OR
!	logical NOT (unary)		

- **Increment and decrement operators**

++	increment	– –	decrement

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

- **Bitwise operators**

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

- **Assignment operators**

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

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

12.2 Using Expression Analysis in the Debugger

The debugger's expression analysis is based on C expression analysis. This includes all mathematical, relational, pointer, and assignment operators. However, a few limitations, as well as a few additional features, are not described in K&R C.

Restrictions

The following restrictions apply to the debugger's expression analysis features.

- The sizeof operator is not supported.
- The comma operator (,) is not supported (commas are used to separate parameter values for the debugger commands).
- Function calls and string constants are currently not supported in expressions.
- The debugger supports a limited capability of type casts; the following forms are allowed:

  (*basic type* **)**
  (*basic type* ***** *...***)**
  ([*structure/union/enum***]** *structure/union/enum tag* **)**
  ([*structure/union/enum***]** *structure/union/enum tag* ***** *...* **)**

  Note that you can use up to six *****s in a cast.

Additional features

- All floating-point operations are performed in double precision using standard widening. (This is transparent.) Floats are represented in IEEE floating-point format.
- All registers can be referenced by name. The TMS320C6x's auxiliary registers are treated as integers and/or pointers.
- Void expressions are legal (treated like integers).
- The specification of variables and functions can be qualified with context information. Local variables (including local statics) can be referenced with the expression form:

  *function name***.***local name*

This expression format is useful for examining the automatic variables of a function that is not currently being executed. Unless the variable is static, however, the function must be somewhere in the current call stack. Note that if you want to see local variables from the currently executing function, you need not use this form; you can simply specify the variable name (just as in your C source).

File-scoped variables (such as statics or functions) can be referenced with the following expression form:

*filename***.***function name*

or *filename***.***variable name*

This expression format is useful for accessing a file-scoped static variable (or function) that may share its name with variables in other files.

Note that in this expression, *filename* **does not include** the file extension; the debugger searches the object symbol table for any source filename that matches the input name, disregarding any extension. Thus, if the variable *ABC* is in file source.c, you can specify it as source.ABC.

Note that these expression forms can be combined into an expression of the form:

*filename***.***function name***.***variable name*

- Any integral or void expression can be treated as a pointer and used with the indirection operator (`*`). Here are several examples of valid use of a pointer in an expression:

```
*123
*AR5
*(AR2 + 123)
*(I*J)
```

  By default, the values are treated as integers (that is, these expressions point to integer values).

- Any expression can be typecast to a pointer to a specific type (overriding the default of pointing to an integer, as described above).

  Hint: You can use casting with the WA and DISP commands to display data in a desired format.

  For example, the expression:

```
*(float *)10
```

  treats 10 as a pointer to a floating-point value at location 10 in memory. In this case, the debugger fetches the contents of memory location 10 and treats the contents as a floating-point value. If you use this expression as a parameter for the DISP command, the debugger displays memory contents as an array of floating-point values within the DISP window, beginning with memory location 10 as array member [0].

Note how the first expression differs from the expression:

```
(float)*10
```

In this case, the debugger fetches an integer from address 10 and converts the integer to a floating-point value.

You can also typecast to user-defined types such as structures. For example, in the expression:

```
((struct STR *)10)–>field
```

the debugger treats memory location 10 as a pointer to a structure of type STR (assuming that a structure is at address 10) and accesses a field from that structure.

Appendix A

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. (For more information on the environment variables mentioned below, refer to your installation guide.)

The debugger:

1) Reads options from the command line.
2) Reads any information specified with the D_OPTIONS environment variable.
3) Reads information from the D_DIR and D_SRC environment variables.
4) Looks for the init.clr screen configuration file.

   (The debugger searches for the screen configuration file in directories named with D_DIR.)
5) Initializes the debugger screen and windows but initially displays only the COMMAND window.
6) Finds the batch file that defines your memory map by searching in directories named with D_DIR. The debugger expects this file to set up the memory map and follows these steps to look for the batch file:

   - ❑ When you invoke the debugger, it checks to see if you've used the –t debugger option. If it finds the –t option, the debugger reads and executes the specified file.
   - ❑ If you have not used the –t option, the debugger looks for the default initialization batch file.

     If the debugger finds the file, 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*.

7) Loads any object filenames specified with D_OPTIONS or specified on the command line during invocation.

8) Determines the initial mode (auto, assembly, mixed, or minimal) and displays the appropriate windows on the screen.

At this point, the debugger is ready to process any commands that you enter.

Appendix B

Debugger Messages

This appendix contains an alphabetical listing of the progress and error messages that the debugger might display in the display area of the COMMAND window. Each message contains both a description of the situation that causes the message and an action to take if the message indicates a problem or error.

Topic **Page**

B.1 Associating Sound With Error Messages

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

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

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

B.2 Alphabetical Summary of Debugger Messages

Symbol

‘]’ expected

Description This is an expression error—it means that the parameter contained an opening bracket symbol “**[**” but didn’t contain a closing bracket symbol “**]**”.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

‘)’ expected

Description This is an expression error—it means that the parameter contained an opening parenthesis symbol “**(**” but didn’t contain a closing parenthesis symbol “**)**”.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

A

Aborted by user

Description The debugger halted a long COMMAND display listing (from WHATIS, DIR, ML, or BL) because you pressed the ESC key.

Action None required; this is normal debugger behavior.

B

Breakpoint already exists at *address*

Description During single-step execution, the debugger attempted to set a breakpoint where one already existed. (This isn't necessarily a breakpoint that you set—it may have been an internal breakpoint that was used for single-stepping).

Action None should be required; you may want to reset the program entry point (RESTART) and reenter the single-step command.

Breakpoint table full

Description 200 breakpoints are already set, and there was an attempt to set another. The maximum limit of 200 breakpoints includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.

Action Enter a BL command to see where breakpoints are set in your program. Use the BR command to delete all software breakpoints, or use the BD command to delete individual software breakpoints.

C

Cannot allocate host memory

Description This is a fatal error—it means that the debugger is running out of memory.

Action You might try invoking the debugger with the –v option so that fewer symbols may be loaded. Or you might want to relink your program and link in fewer modules at a time.

Cannot allocate system memory

Description This is a fatal error—it means that the debugger is running out of memory.

Action You might try invoking the debugger with the –v option so that fewer symbols may be loaded. Or you might want to relink your program and link in fewer modules at a time.

Cannot edit field

Description Expressions that are displayed in the WATCH window cannot be edited.

Action If you attempted to edit an expression in the WATCH window, you may have actually wanted to change the value of a symbol or register used in the expression. Use the ? or EVAL command to edit the actual symbol or register. The expression value will automatically be updated.

Cannot find/open initialization file

Description The debugger can't find the init.cmd file.

Action Be sure that init.cmd is in the appropriate directory. If it isn't, copy it from the debugger product diskette. If the file is already in the correct directory, verify that the D_DIR environment variable is set up to identify the directory. See *Setting Up the Debugger Environment* in the appropriate installation guide.

Cannot halt the processor

Description This is a fatal error—for some reason, pressing ESC didn't halt program execution.

Action Exit the debugger. Invoke the autoexec or initdb.bat file; then invoke the debugger again.

Cannot map into reserved memory: ?

Description The debugger tried to access unconfigured/reserved/nonexistent memory.

Action Remap the reserved memory accesses.

Cannot open config file

Description The SCONFIG command can't find the screen-customization file that you specified.

Action Be sure that the filename was typed correctly. If it wasn't, reenter the command with the correct name. If it was, reenter the command and specify full path information with the filename.

Cannot open "*filename*"

Description The debugger attempted to show *filename* in the FILE window but could not find the file.

Action Be sure that the file exists as named. If it does, enter the USE command to identify the file's directory.

Cannot open object file: "*filename*"

Description The file specified with the LOAD, SLOAD, or RELOAD command is not an object file that the debugger can load.

Action Be sure that you're loading an actual object file. Be sure that the file was linked (you may want to run the cl6x shell program again to create an executable object file).

Cannot open new window

Description A maximum of 127 windows can be open at once. The last request to open a window would have made 128, which isn't possible.

Action Close any unnecessary windows. Windows that can be closed include WATCH, CALLS, DISP, and additional MEMORY windows. To close the WATCH window, enter WD. To close any of these windows, make the desired window active and press F4.

Cannot read processor status

Description This is a fatal error—for some reason, pressing ESC didn't halt program execution.

Action Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger again.

Cannot reset the processor

Description This is a fatal error—for some reason, pressing ESC didn't halt program execution.

Action Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger again.

Cannot restart processor

Description If a program doesn't have an entry point, then RESTART won't reset the PC to the program entry point.

Action Don't use RESTART if your program doesn't have an explicit entry point.

Cannot set/verify breakpoint at *address*

Description Either you attempted to set a breakpoint in read-only or protected memory, or there are hardware problems with the target system. This may also happen when you enable or disable on-chip memory while using breakpoints.

Action Check your memory map.

Cannot take address of register

Description This is an expression error. C does not allow you to take the address of a register.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

Command "*cmd*" not found

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

Action Reenter the correct command. Refer to Chapter 11, *Summary of Commands and Special Keys*.

Conflicting map range

Description A block of memory specified with the MA command overlaps an existing memory map entry. Blocks cannot overlap.

Action Use the ML command to list the existing memory map; this will help you find that existing block that the new block would overlap. If the existing block is not necessary, delete it with the MD command and reenter the MA command. If the existing block is necessary, reenter the MA command with parameters that will not overlap the existing block.

Corrupt call stack

Description The debugger tried to update the CALLS window and couldn't. This may be because a function was called that didn't return. Or it could be that the program stack was overwritten in target memory. Another reason you may have this message is that you are debugging code that has optimization enabled (for example, you did not use the –g compile switch); if this is the case, ignore this message—code execution is not affected.

Action If your program called a function that didn't return, then this is normal behavior (as long as you intended for the function not to return). Otherwise, you may be overwriting program memory.

E

Error in expression

Description This is an expression error.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

F

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.

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 was more than 65 518 bytes long.

Action Try loading the file without the symbol table (SLOAD), or use lnk6x 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 B.3, *Additional Instructions for Expression Errors*, page B-17.

Function required

Description The parameter for the FUNC command must be the name of a function in the program that is loaded.

Action Reenter the FUNC command with a valid function name.

I

Illegal addressing mode

Description An illegal 'C6x addressing mode was encountered.

Action Refer to the *TMS320C62xx CPU and Instruction Set Reference Guide* for valid addressing modes.

Illegal cast

Description This is an expression error—the expression parameter uses a cast that doesn't meet the C language rules for casts.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

Illegal control transfer instruction

Description The instruction following a delayed branch/call instruction was modifying the program counter.

Action Modify your source code.

Illegal left hand side of assignment

Description This is an expression error—the lefthand side of an assignment expression doesn't meet C language assignment rules.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

Illegal memory access

Description Your program tried to access unmapped memory.

Action Modify your source code.

Illegal opcode

Description An invalid 'C6x instruction was encountered.

Action Modify your source code.

Illegal operand of &

Description This is an expression error—the expression attempts to take the address of an item that doesn't have an address.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

Illegal pointer math

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

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

Illegal pointer subtraction

Description This is an expression error—the expression attempts to use pointers in a way that is not valid.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

Illegal structure reference

Description This is an expression error—either the item being referenced as a structure is not a structure, or you are attempting to reference a nonexistent portion of a structure.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

Illegal use of structures

Description This is an expression error—the expression parameter is not using structures according to the C language rules.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

Illegal use of void expression

Description This is an expression error—the expression parameter does not meet the C language rules.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

Integer not allowed

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

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

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 11, *Summary of Commands and Special Keys*.

Invalid attribute name

Description The COLOR and SCOLOR commands accept a specific set of area names for their first parameter. The parameter entered did not match one of the valid attributes.

Action Reenter the COLOR or SCOLOR command with a valid area name parameter. Valid area names are listed in Table 9–2 (page 9-3).

Invalid color name

Description The COLOR and SCOLOR commands accept a specific set of color attributes as parameters. The parameter entered did not match one of the valid attributes.

Action Reenter the COLOR or SCOLOR command with a valid color parameter. Valid color attributes are listed in Table 9–1 (page 9-2).

Invalid memory attribute

Description The third parameter of the MA command specifies the type, or attribute, of the block of memory that MA adds to the memory map. The parameter entered did not match one of the valid attributes.

Action 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)
PROTECT	(no-access memory)
OUTPORT, P|W	(output port)
INPORT, P|R	(input port)
IOPORT, P|R|W	(input/output port)

Invalid object file

Description Either the file specified with the LOAD, SLOAD, or RELOAD command is not an object file that the debugger can load, or it has been corrupted.

Action Be sure that you're loading an actual object file. Be sure that the file was linked (you may want to run cl6x 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; re-compile, assemble, and link with cl6x.

Invalid watch delete

Description The debugger can't delete the parameter supplied with the WD command. Usually, this is because the watch index doesn't exist or because a symbol name was typed instead of a watch index.

Action Reenter the WD command. Be sure to specify the watch index that matches the item you'd like to delete (this is the number in the left column of the WATCH window). Remember, you can't delete items symbolically—you must delete them by number.

Invalid window position

Description The debugger can't move the active window to the XY position specified with the MOVE command. Either the XY parameters are not within the screen limits, or the active window may be too large to move to the desired position.

Action You can use the mouse to move the window.

- ❑ If you don't have a mouse, enter the MOVE command without parameters; then use the arrow keys to move the window. When you're finished, you *must* press ESC or ↵.
- ❑ If you prefer to use the MOVE command with parameters, the minimum XY position is 0,1; the maximum position depends on which screen size you're using.

Invalid window size

Description The width and length specified with the SIZE or MOVE command may be too large or too small. If valid width and length were specified, then the active window is already at the far right or bottom of the screen and so cannot be made larger.

Action You can use the mouse to size the window.

- ❑ If you don't have a mouse, enter the SIZE command without parameters; then use the arrow keys to move the window. When you're finished, you *must* press ESC or ↵.
- ❑ If you prefer to use the SIZE command with parameters, the minimum size is 4 by 3; the maximum size depends on which screen size you're using.

L

Load aborted

Description This message always follows another message.

Action Refer to the message that preceded *Load aborted*.

Lval required

Description This is an expression error—an assignment expression was entered that requires a legal left-hand side.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

M

Memory map table full

Description Too many blocks have been added to the memory map. This will rarely happen unless blocks are added word by word (which is inadvisable).

Action Stop adding blocks to the memory map. Consolidate any adjacent blocks that have the same memory attributes.

N

Name "*name*" not found

Description The command cannot find the object named *name*.

Action If *name* is a symbol, be sure that it was typed correctly. If it wasn't, reenter the command with the correct name. If it was, then be sure that the associated object file is loaded.

Non-repeatable instruction

Description The instruction following the RPT instruction is not a repeatable instruction.

Action Modify your code.

P

Pointer not allowed

Description This is an expression error.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

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.

S

Specified map not found

Description The MD command was entered with an address or block that is not in the memory map.

Action Use the ML command to verify the current memory map. When using MD, you can specify only the first address of a defined block.

Structure member not found

Description This is an expression error—an expression references a non-existent structure member.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

Structure member name required

Description This is an expression error—a symbol name followed by a period but no member name.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

Structure not allowed

Description This is an expression error—the expression is attempting an operation that cannot be performed on a structure.

Action See Section B.3, *Additional Instructions for Expression Errors*, page B-17.

T

Take file stack too deep

Description Batch files can be nested up to 10 levels deep. Batch files can call other batch files, which can call other batch files, and so on. Apparently, the batch file that you are TAKEing calls batch files that are nested more than 10 levels deep.

Action Edit the batch file that caused the error. Instead of calling another batch file from within the offending file, you may want to copy the contents of the second file into the first. This will remove a level of nesting.

Too many breakpoints

Description 200 breakpoints are already set, and there was an attempt to set another. Note that the maximum limit of 200 breakpoints includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.

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

Too many paths

Description More than 20 paths have been specified cumulatively with the USE command, D_SRC environment variable, and –i debugger option.

Action Don't enter the USE command before entering another command that has a *filename* parameter. Instead, enter the second command and specify full path information for the *filename*.

U

Undeclared port address

Description You attempted to do a connect/disconnect on an address that isn't declared as a port.

Action Verify the address of the port to be connected or disconnected.

User halt

Description The debugger halted program execution because you pressed the ESC key.

Action None required; this is normal debugger behavior.

W

Window not found

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

Action Reenter the WIN command. Remember that window names must be typed in uppercase letters. Here are the valid window names; the bold letters show the smallest acceptable abbreviations:

CALLS	**CP**U	**DISP**
COMMAND	**DISA**SSEMBLY	**F**ILE
MEMORY	**P**ROFILE	**W**ATCH

Write not allowed for port

Description You attempted to connect a file for output operation to an address that is not configured for write.

Action Either change the 'C6x software to write a port that is configured for write, or change the attributes of the port.

B.3 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.

Appendix C

Glossary

A

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

aggregate type: A C data type, such as a structure or array, in which a variable is composed of multiple variables, called members.

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

ANSI C: A version of the C programming language that conforms to the C standards defined by the *American National Standards Institute.*

assembly mode: A debugging mode that shows assembly language code in the DISASSEMBLY window and doesn't show the FILE window, no matter what type of code is currently running.

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

auto mode: A context-sensitive debugging mode that automatically switches between showing assembly language code in the DISASSEMBLY window and C code in the FILE window, depending on what type of code is currently running.

B

batch file: One of two different types of files. One type contains DOS commands for the PC to execute. A second type of batch file contains debugger commands for the debugger to execute. The PC doesn't execute debugger batch files, and the debugger doesn't execute PC batch files.

benchmarking: A type of program execution that allows you to track the number of CPU cycles consumed by a specific section of code.

breakpoint: A point within your program where execution will halt because of a previous request from you.

C

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

CALLS window: A window that lists the functions called by your program.

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

children: Additional windows opened for aggregate types that are members of a parent aggregate type displayed in an existing DISP window.

cl6x: A shell utility that invokes the TMS320C6x 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, FILES, and CALLS windows.

COFF: *Common Object File Format*. An implementation of the object file format of the same name developed by AT&T. The TMS320 fixed-point DSP compiler, assembler, and linker use and generate COFF files.

command line: The portion of the COMMAND window where you can enter commands.

command-line cursor: A block-shaped cursor that identifies the current character position on the command line.

COMMAND window: A window that provides an area for you to enter commands and for the debugger to echo command entry, show command output, and list progress or error messages.

CPU window: A window that displays the contents of 'C6x on-chip registers, including the program counter, status register, A-file registers, and B-file registers.

current-field cursor: A screen icon that identifies the current field in the active window.

cursor: An icon on the screen (such as a rectangle or a horizontal line) that is used as a pointing device. The cursor is usually under mouse or keyboard control.

D

data-display windows: Windows for observing and modifying various types of data. This category includes the MEMORY, CPU, DISP, and WATCH windows.

D_DIR: An environment variable that identifies the directory containing the commands and files necessary for running the debugger.

debugger: A window-oriented software interface that helps you to debug 'C6x programs running on a 'C6x simulator.

disassembly: Assembly language code formed from the reverse-assembly of the contents of memory.

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

discontinuity: A state in which the addresses fetched by the debugger become nonsequential as a result of instructions that load the PC with new values, such as branches, calls, and returns.

DISP window: A window that displays the members of an aggregate data type.

display area: The portion of the COMMAND window where the debugger echoes command entry, shows command output, and lists progress or error messages.

D_OPTIONS: An environment variable that you can use for identifying often-used debugger options.

drag: To move the mouse while pressing one of the mouse buttons.

D_SRC: An environment variable that identifies directories containing program source files.

E

EGA: *Enhanced Graphics Adaptor.* An industry standard for video cards.

EISA: *Extended Industry Standard Architecture.* A standard for PC buses.

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.

I

init.cmd: A batch file that contains debugger-initialization commands. If this file isn't present when you first invoke the debugger, then all memory is invalid.

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

M

memory map: A map of memory space that tells the debugger which areas of memory can and can't be accessed.

MEMORY window: A window that displays the contents of memory.

menu bar: A row of pulldown menu selections found at the top of the debugger display.

minimal mode: A debugging mode that displays the COMMAND window, WATCH window, and DISP window only.

mixed mode: A debugging mode that simultaneously shows both assembly language code in the DISASSEMBLY window and C code in the FILE window.

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

P

PC: Personal computer or program counter, depending on the context and where it's used in this book: 1) In installation instructions or information relating to hardware and boards, *PC* means *personal computer* (as in IBM PC). 2) In general debugger and program-related information, *PC* means *program counter*, which is the register that identifies the current statement in your program.

point: To move the mouse cursor until it overlays the desired object on the screen.

pulldown menu: A command menu that is accessed by name or with the mouse from the menu bar at the top of the debugger display.

S

scalar type: A C type in which the variable is a single variable, not composed of other variables.

scrolling: A method of moving the contents of a window up, down, left, or right to view contents that weren't originally shown.

side effects: A feature of C expressions in which using an assignment operator in an expression affects the value of one of the components used in the expression.

simulator: A development tool that simulates the operation of the 'C6x and lets you execute and debug applications programs by using the 'C6x debugger.

single-step: A form of program execution that allows you to see the effects of each statement. The program is executed statement by statement; the debugger pauses after each statement to update the data-display windows.

symbol table: A file that contains the names of all variables and functions in your 'C6x program.

V

VGA: *Video Graphics Array.* An industry standard for video cards.

W

WATCH window: A window that displays the values of selected expressions, symbols, addresses, and registers.

window: A defined rectangular area of virtual space on the display.

Index

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

X

Z

IMPORTANT NOTICE

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

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.