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.

Copyright © 1997, Texas Instruments Incorporated
Preface

Read This First

About This Manual

This book tells you how to use the TMS320C6x C source debugger with the simulator. You can use the debugger to test and refine your 'C6x code.

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

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

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

Before you use this book, you should use your getting started guide to install the C source debugger.

Notational Conventions

This document uses the following conventions.

- The TMS320C6x family of devices is referred to as 'C6x.

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Result Displayed in the Command Window</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>whatis aai</code></td>
<td><code>int aai[10][5];</code></td>
</tr>
<tr>
<td><code>whatis xxx</code></td>
<td><code>struct xxx {</code></td>
</tr>
<tr>
<td></td>
<td><code>    int a;</code></td>
</tr>
<tr>
<td></td>
<td><code>    int b;</code></td>
</tr>
<tr>
<td></td>
<td><code>    int c;</code></td>
</tr>
<tr>
<td></td>
<td><code>    int f1 : 2;</code></td>
</tr>
<tr>
<td></td>
<td><code>    int f2 : 4;</code></td>
</tr>
<tr>
<td></td>
<td><code>    struct xxx *f3;</code></td>
</tr>
<tr>
<td></td>
<td><code>    int f4[10];</code></td>
</tr>
<tr>
<td></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

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

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

```
load object filename
```

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

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

```
run [expression]
```

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

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

```
sound {on | off}
```

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

Unless the list is enclosed in square brackets, you must choose one item from the list.
Related Documentation From Texas Instruments

The following books describe the 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:


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


Trademarks

320 Hotline On-line is a trademark of Texas Instruments Incorporated.
PC is a trademark of International Business Machines Corporation.
Solaris and SunOS are trademarks of Sun Microsystems, Inc.
SPARCstation is trademark of SPARC International, Inc., but licensed exclusively to Sun Microsystems, Inc.
Windows and Windows NT are registered trademarks of Microsoft Corporation.
X Window System is a trademark of the Massachusetts Institute of Technology.
If You Need Assistance . . .

### World-Wide Web Sites

<table>
<thead>
<tr>
<th>Web Site</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TI Online</td>
<td><a href="http://www.ti.com">http://www.ti.com</a></td>
</tr>
<tr>
<td>Semiconductor Product Information Center (PIC)</td>
<td><a href="http://www.ti.com/sc/docs/pic/home.htm">http://www.ti.com/sc/docs/pic/home.htm</a></td>
</tr>
<tr>
<td>DSP Solutions</td>
<td><a href="http://www.ti.com/dspss">http://www.ti.com/dspss</a></td>
</tr>
<tr>
<td>320 Hotline Online™</td>
<td><a href="http://www.ti.com/sc/docs/dspss/support.htm">http://www.ti.com/sc/docs/dspss/support.htm</a></td>
</tr>
</tbody>
</table>

### North America, South America, Central America

<table>
<thead>
<tr>
<th>Service</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Information Center (PIC)</td>
<td>(972) 644-5580</td>
</tr>
<tr>
<td>TI Literature Response Center U.S.A.</td>
<td>(800) 477-8924</td>
</tr>
<tr>
<td>Software Registration/Upgrades</td>
<td>(214) 638-0333 Fax: (214) 638-7742</td>
</tr>
<tr>
<td>U.S.A. Factory Repair/Hardware Upgrades</td>
<td>(281) 274-2285</td>
</tr>
<tr>
<td>U.S. Technical Training Organization</td>
<td>(972) 644-5580</td>
</tr>
<tr>
<td>DSP Hotline</td>
<td>(281) 274-2320 Fax: (281) 274-2324 Email: <a href="mailto:dsph@ti.com">dsph@ti.com</a></td>
</tr>
<tr>
<td>DSP Modern BBS</td>
<td>(281) 274-2323</td>
</tr>
<tr>
<td>DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/mirrors/tms320bbs</td>
<td></td>
</tr>
</tbody>
</table>

### Europe, Middle East, Africa

<table>
<thead>
<tr>
<th>Service</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>European Product Information Center (EPIC) Hotlines:</td>
<td></td>
</tr>
<tr>
<td>Multi-Language Support</td>
<td>+33 1 30 70 11 69 Fax: +33 1 30 70 10 32 Email: <a href="mailto:epic@ti.com">epic@ti.com</a></td>
</tr>
<tr>
<td>Deutsch</td>
<td>+49 8161 80 33 11 or +33 1 30 70 11 68</td>
</tr>
<tr>
<td>English</td>
<td>+33 1 30 70 11 65</td>
</tr>
<tr>
<td>Francais</td>
<td>+33 1 30 70 11 64</td>
</tr>
<tr>
<td>Italiano</td>
<td>+33 1 30 70 11 67</td>
</tr>
<tr>
<td>EPIC Modem BBS</td>
<td>+33 1 30 70 11 99</td>
</tr>
<tr>
<td>European Factory Repair</td>
<td>+33 4 93 22 25 40</td>
</tr>
<tr>
<td>Europe Customer Training Helpline</td>
<td>Fax: +49 81 61 80 40 10</td>
</tr>
</tbody>
</table>

### Asia-Pacific

<table>
<thead>
<tr>
<th>Service</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literature Response Center</td>
<td>+852 2 956 7288</td>
</tr>
<tr>
<td>Hong Kong DSP Hotline</td>
<td>+852 2 956 7268</td>
</tr>
<tr>
<td>Korea DSP Hotline</td>
<td>+82 2 551 2804</td>
</tr>
<tr>
<td>Korea DSP Modem BBS</td>
<td>+82 2 551 2914</td>
</tr>
<tr>
<td>Singapore DSP Hotline</td>
<td>+65 390 7179</td>
</tr>
<tr>
<td>Taiwan DSP Hotline</td>
<td>+886 2 377 1450</td>
</tr>
<tr>
<td>Taiwan DSP Modem BBS</td>
<td>+886 2 376 2592</td>
</tr>
<tr>
<td>Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/</td>
<td></td>
</tr>
</tbody>
</table>

### Japan

<table>
<thead>
<tr>
<th>Service</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Information Center</td>
<td>+0120-81-0028 (in Japan)</td>
</tr>
<tr>
<td>+03-3457-0972 or (INTL) 813-3457-0972</td>
<td></td>
</tr>
<tr>
<td>DSP Hotline</td>
<td>+03-3457-1259 or (INTL) 813-3457-1259</td>
</tr>
<tr>
<td>DSP BBS via Nifty-Serve</td>
<td>+03-3769-8735 or (INTL) 813-3769-8735 Type “Go TIASP”</td>
</tr>
</tbody>
</table>

### 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 Email: comments@books.sc.ti.com |
| Technical Documentation Services, MS 702  |
| P.O. Box 1443 Houston, Texas 77251-1443 |

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

1 Overview of a Code Development and Debugging System ........................................ 1-1  
*Provides an overview of the C source debugger, describes the code development environment, and tells you how to invoke the debugger and access the online help.*

1.1 Description of the C Source Debugger ................................................................. 1-2  
1.2 Developing Code for the TMS320C6x .................................................................... 1-4  
1.3 Preparing Your Program for Debugging ............................................................... 1-6  
1.3.1 Debugging optimized code .............................................................................. 1-6  
1.3.2 Profiling optimized code ................................................................................. 1-6  
1.4 Invoking the Debugger ............................................................................................ 1-7  
1.4.1 Clearing the .bss section (–c option) ............................................................... 1-8  
1.4.2 Displaying the debugger on a different machine (–d option) ......................... 1-8  
1.4.3 Identifying additional directories (–i option) ................................................. 1-8  
1.4.4 Selecting big-endian format (–me option) ..................................................... 1-8  
1.4.5 Entering the profiling environment (–profile option) .................................... 1-8  
1.4.6 Loading the symbol table only (–s option) .................................................... 1-9  
1.4.7 Identifying a new initialization file (–t option) ............................................. 1-9  
1.4.8 Loading without the symbol table (–v option) .............................................. 1-9  
1.4.9 Ignoring D_OPTIONS (–x option) ................................................................. 1-9  
1.5 Exiting the Debugger .............................................................................................. 1-10  
1.6 Debugging Your Programs ...................................................................................... 1-11  
1.7 Accessing Online Help ........................................................................................... 1-12  
1.7.1 Accessing a list of help topics ....................................................................... 1-12  
1.7.2 Accessing context-sensitive help ................................................................... 1-12  
1.7.3 Accessing help for debugger commands ..................................................... 1-13
Contents

2 The Debugger Display ......................................................... 2-1
   Describes the default displays, tells you how to switch between assembly language and C debugging, describes the various types of windows on the display, and tells you how to identify the active window.
   2.1 Debugging Modes and Default Displays ........................................ 2-2
       Auto mode ........................................................................ 2-2
       Assembly mode .................................................................. 2-2
       Mixed mode ....................................................................... 2-4
       Restrictions associated with debugging modes ......................... 2-5
   2.2 Descriptions of the Different Kinds of Windows and Their Contents .... 2-6
       File window ...................................................................... 2-8
       Disassembly window .......................................................... 2-10
       Calls window .................................................................... 2-12
       Profile window ................................................................... 2-13
       Memory window ................................................................... 2-14
       CPU window ...................................................................... 2-17
       Watch window .................................................................... 2-18
       Command window .............................................................. 2-20
   2.3 The Active Window ............................................................. 2-21

3 Customizing the Debugger Display ............................................. 3-1
   Describes the methods that you can use for customizing the display.
   3.1 Managing Debugger Windows ................................................ 3-2
       Floating windows outside the application window ....................... 3-2
       Docking debugger windows ..................................................... 3-4
       Closing debugger windows ...................................................... 3-6
   3.2 Displaying the Status Bar ........................................................ 3-7
   3.3 Displaying and Moving the Toolbar ............................................. 3-8
       Moving the toolbar ............................................................... 3-8
       Undocking the toolbar ......................................................... 3-10
       Hiding the toolbar ................................................................ 3-11
   3.4 Changing the Prompt ............................................................. 3-12
   3.5 Saving and Using Custom Displays .......................................... 3-13
       Saving a custom display ....................................................... 3-13
       Loading a custom display ....................................................... 3-14
       Invoking the debugger with a custom display .......................... 3-15
4 Entering and Using Commands ........................................ 4-1
Tells you how to use the menu bar and the toolbar for entering commands, describes the rules for entering commands on the command line, and tells you how to define your own command strings.

4.1 Using the Menu Bar and the Toolbar ............................ 4-2
Overview of the debugger menus ..................................... 4-2
Selecting menu options .................................................. 4-5
Understanding the menu conventions ............................... 4-5
Overview of the toolbar .................................................. 4-5

4.2 Entering Commands on the Command Line ................... 4-6
Typing in and entering commands .................................... 4-7
Using the command history ............................................. 4-9
Clearing the display area ................................................. 4-9

4.3 Defining Your Own Command Strings ......................... 4-10
Defining an alias ......................................................... 4-11
Defining an alias with parameters ................................. 4-11
Editing or redefining an alias ....................................... 4-12
Deleting an alias ......................................................... 4-12
Considerations for using alias definitions .......................... 4-12

4.4 Entering Operating-System Commands ....................... 4-13
Entering a single command from the debugger command line .. 4-13
Entering several commands from a system shell .............. 4-14

5 Defining a Memory Map ............................................. 5-1
Contains instructions for setting up a memory map that enables the debugger to correctly access target memory and includes hints about using batch files.

5.1 The Memory Map: What It Is and Why You Must Define It .... 5-2
Potential memory map problems ..................................... 5-3

5.2 A Sample Memory Map ............................................ 5-4

5.3 Creating or Modifying the Memory Map ..................... 5-5
Adding a range of memory ............................................. 5-5
Deleting a range of memory .......................................... 5-6
Modifying a defined range of memory ............................... 5-7

5.4 Enabling Memory Mapping ....................................... 5-8

5.5 Defining a Memory Map in a Batch File .................... 5-10

5.6 Returning to the Original Memory Map ..................... 5-11

5.7 Simulating I/O Space ............................................. 5-12
Connecting an I/O port ................................................. 5-12
Disconnecting an I/O port ............................................. 5-13
6  Loading and Displaying Code  ......................................................... 6-1
Tells you how to use the debugger modes to view the source files and how to load source files
and object files.

6.1 Viewing Assembly Language Code, C Code, or Both .......................... 6-2
Selecting a debugging mode ......................................................... 6-3

6.2 Loading and Displaying Assembly Language Code ............................ 6-4
Loading an object file and its symbol table ...................................... 6-4
Loading an object file without its symbol table .................................. 6-5
Loading a symbol table only ......................................................... 6-5
Loading code while invoking the debugger ........................................ 6-5
Displaying portions of disassembly ................................................ 6-6
Displaying assembly source code .................................................. 6-7

6.3 Displaying C Code ................................................................. 6-8
Displaying the contents of a text file .............................................. 6-8
Displaying a specific C function ................................................... 6-9
Displaying code beginning at a specific point .................................... 6-10

7  Running Code ............................................................................ 7-1
Describes the basic run commands and single-step commands, tells you how to halt program
execution, and discusses using software breakpoints.

7.1 Defining the Starting Point for Program Execution ............................ 7-2
7.2 Using the Basic Run Commands ................................................... 7-4
Running an entire program ........................................................... 7-4
Running code up to a specific point in a program ............................... 7-5
Running the code in the current C function ...................................... 7-6
Running code through breakpoints ............................................... 7-6
Resetting the simulator ................................................................. 7-6

7.3 Single-Stepping Through Code ................................................... 7-7
Single-stepping through assembly language or C code ......................... 7-7
Single-stepping through C code .................................................... 7-8
Single-stepping through code and stepping over breakpoints ............... 7-9
Single-stepping through code and stepping over C functions ............. 7-9

7.4 Running Code Conditionally ..................................................... 7-10
7.5 Benchmarking ................................................................. 7-11
7.6 Halting Program Execution ...................................................... 7-12
7.7 Using Software Breakpoints ...................................................... 7-13
Setting a software breakpoint ....................................................... 7-14
Clearing a software breakpoint .................................................... 7-16
Clearing all software breakpoints ............................................... 7-16
Saving breakpoint settings .......................................................... 7-17
Loading saved breakpoint settings ............................................... 7-18
8 Managing Data

Describes the data-display windows and tells you how to edit data (memory contents, register contents, and individual variables).

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Where Data Is Displayed</td>
<td>8-2</td>
</tr>
<tr>
<td>8.2 Basic Commands for Managing Data</td>
<td>8-2</td>
</tr>
<tr>
<td>Determining the type of a variable</td>
<td>8-2</td>
</tr>
<tr>
<td>Evaluating an expression</td>
<td>8-3</td>
</tr>
<tr>
<td>8.3 Basic Methods for Changing Data Values</td>
<td>8-4</td>
</tr>
<tr>
<td>Editing data displayed in a window</td>
<td>8-4</td>
</tr>
<tr>
<td>Editing data using expressions that have side effects</td>
<td>8-4</td>
</tr>
<tr>
<td>8.4 Managing Data in Memory</td>
<td>8-6</td>
</tr>
<tr>
<td>Changing the memory range displayed in a Memory window</td>
<td>8-6</td>
</tr>
<tr>
<td>Opening an additional Memory window</td>
<td>8-7</td>
</tr>
<tr>
<td>Displaying memory contents while you are debugging C</td>
<td>8-8</td>
</tr>
<tr>
<td>Saving memory values to a file</td>
<td>8-9</td>
</tr>
<tr>
<td>Filling a block of memory</td>
<td>8-10</td>
</tr>
<tr>
<td>8.5 Managing Register Data</td>
<td>8-12</td>
</tr>
<tr>
<td>Displaying register contents</td>
<td>8-12</td>
</tr>
<tr>
<td>Accessing single-precision floating-point registers</td>
<td>8-15</td>
</tr>
<tr>
<td>Accessing double-precision floating-point registers</td>
<td>8-16</td>
</tr>
<tr>
<td>8.6 Managing Data in a Watch Window</td>
<td>8-17</td>
</tr>
<tr>
<td>Displaying data in a Watch window</td>
<td>8-17</td>
</tr>
<tr>
<td>Displaying additional data</td>
<td>8-19</td>
</tr>
<tr>
<td>Deleting watched values</td>
<td>8-19</td>
</tr>
<tr>
<td>8.7 Displaying Data in Alternative Formats</td>
<td>8-20</td>
</tr>
<tr>
<td>Changing the default format for specific data types</td>
<td>8-20</td>
</tr>
<tr>
<td>Changing the default format with data-management commands</td>
<td>8-22</td>
</tr>
</tbody>
</table>

9 Profiling Code Execution

Describes the profiling environment and tells you how to collect statistics about code execution.

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Overview of the Profiling Environment</td>
<td>9-2</td>
</tr>
<tr>
<td>9.2 An Overview of the Profiling Process</td>
<td>9-3</td>
</tr>
<tr>
<td>A profiling strategy</td>
<td>9-3</td>
</tr>
<tr>
<td>9.3 Entering the Profiling Environment</td>
<td>9-4</td>
</tr>
<tr>
<td>9.4 Defining Areas for Profiling</td>
<td>9-5</td>
</tr>
<tr>
<td>Marking an area with a mouse</td>
<td>9-5</td>
</tr>
<tr>
<td>Marking an area with a dialog box</td>
<td>9-8</td>
</tr>
<tr>
<td>Disabling an area</td>
<td>9-10</td>
</tr>
<tr>
<td>Reenabling a disabled area</td>
<td>9-11</td>
</tr>
<tr>
<td>Unmarking an area</td>
<td>9-12</td>
</tr>
<tr>
<td>Restrictions on profiling areas</td>
<td>9-12</td>
</tr>
<tr>
<td>9.5 Defining a Stopping Point</td>
<td>9-15</td>
</tr>
<tr>
<td>Setting a software breakpoint</td>
<td>9-15</td>
</tr>
<tr>
<td>Clearing a software breakpoint</td>
<td>9-16</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>9.6</td>
<td>Running a Profiling Session</td>
</tr>
<tr>
<td></td>
<td>Running a full or quick profiling session</td>
</tr>
<tr>
<td></td>
<td>Resuming a profiling session that has halted</td>
</tr>
<tr>
<td>9.7</td>
<td>Viewing Profile Data</td>
</tr>
<tr>
<td></td>
<td>Viewing different profile data</td>
</tr>
<tr>
<td></td>
<td>Sorting profile data</td>
</tr>
<tr>
<td></td>
<td>Viewing different profile areas</td>
</tr>
<tr>
<td></td>
<td>Interpreting session data</td>
</tr>
<tr>
<td></td>
<td>Viewing code associated with a profile area</td>
</tr>
<tr>
<td>9.8</td>
<td>Saving Profile Data to a File</td>
</tr>
<tr>
<td></td>
<td>Saving the contents of the Profile window</td>
</tr>
<tr>
<td></td>
<td>Saving all data for currently displayed areas</td>
</tr>
<tr>
<td>10</td>
<td>Summary of Commands</td>
</tr>
<tr>
<td></td>
<td>Provides functional and alphabetical summaries of the basic debugger commands and the profiling commands.</td>
</tr>
<tr>
<td>10.1</td>
<td>Functional Summary of Debugger Commands</td>
</tr>
<tr>
<td></td>
<td>Changing modes</td>
</tr>
<tr>
<td></td>
<td>Managing windows</td>
</tr>
<tr>
<td></td>
<td>Displaying and changing data</td>
</tr>
<tr>
<td></td>
<td>Performing system tasks</td>
</tr>
<tr>
<td></td>
<td>Managing breakpoints</td>
</tr>
<tr>
<td></td>
<td>Displaying files and loading programs</td>
</tr>
<tr>
<td></td>
<td>Customizing the screen</td>
</tr>
<tr>
<td></td>
<td>Memory mapping</td>
</tr>
<tr>
<td></td>
<td>Running programs</td>
</tr>
<tr>
<td>10.2</td>
<td>Alphabetical Summary of Debugger Commands</td>
</tr>
<tr>
<td></td>
<td>Profiling commands</td>
</tr>
<tr>
<td>10.3</td>
<td>Summary of Profiling Commands</td>
</tr>
<tr>
<td>11</td>
<td>Using Batch Files to Enter Commands</td>
</tr>
<tr>
<td></td>
<td>Tells you how to create and execute batch files for repetitive tasks.</td>
</tr>
<tr>
<td>11.1</td>
<td>Creating a Batch File</td>
</tr>
<tr>
<td></td>
<td>Echoing strings in a batch file</td>
</tr>
<tr>
<td></td>
<td>Controlling command execution in a batch file</td>
</tr>
<tr>
<td></td>
<td>Pausing the execution of a batch file</td>
</tr>
<tr>
<td>11.2</td>
<td>Executing a Batch File</td>
</tr>
<tr>
<td>11.3</td>
<td>Recording Information From the Display Area</td>
</tr>
<tr>
<td>12</td>
<td>Basic Information About C Expressions</td>
</tr>
<tr>
<td></td>
<td>Many of the debugger commands accept C expressions as parameters. This chapter provides general information about the rules governing C expressions and describes specific implementation features related to using C expressions as command parameters.</td>
</tr>
<tr>
<td>12.1</td>
<td>C Expressions for Assembly Language Programmers</td>
</tr>
<tr>
<td>12.2</td>
<td>Using Expression Analysis in the Debugger</td>
</tr>
<tr>
<td></td>
<td>Restrictions</td>
</tr>
<tr>
<td></td>
<td>Additional features</td>
</tr>
</tbody>
</table>
**A  Alternate Methods for Performing Tasks**  
*Describes the function key methods for entering commands and other alternatives that you can use instead of the toolbar and menu options.*

- A.1  Scrolling a Window's Contents  
- A.2  Editing Data or Selecting the Active Field  
- A.3  Escaping From an Action  
- A.4  Where the Debugger Looks for Files

**B  What the Debugger Does During Invocation**  
*In some circumstances, you may find it helpful to know the steps that the debugger goes through during the invocation process; this appendix lists these steps.*

**C  Debugger Messages**  
*Describes progress and error messages that the debugger may display.*

- C.1  Associating Sound With Error Messages  
- C.2  Alphabetical Summary of Debugger Messages  
- C.3  Additional Instructions for Expression Errors

**D  Glossary**  
*Defines acronyms and key terms used in this book.*
Figures

1–1 The Basic Debugger Display ................................................................. 1-2
1–2 TMS320C6x Software Development Flow ......................................... 1-4
2–1 Typical Assembly Display (for Auto Mode and Assembly Mode) .......... 2-3
2–2 Typical C Display (for Auto Mode Only) ............................................. 2-4
2–3 Typical Mixed Display (for Mixed Mode Only) ................................. 2-5
2–4 Assembly Language Statements Associated With a C Statement ........ 2-11
2–5 Watch add Dialog Box ................................................................. 2-19
2–6 Default Appearances of an Active and an Inactive Window .......... 2-21
4–1 The Menu Bar and the Toolbar .............................................................. 4-2
4–2 File Menu ..................................................................................... 4-2
4–3 View Menu ..................................................................................... 4-3
4–4 Setup Menu ..................................................................................... 4-3
4–5 Memory Menu .................................................................................. 4-3
4–6 Target Menu ..................................................................................... 4-3
4–7 Mode Menu ..................................................................................... 4-4
4–8 Profile Menu ..................................................................................... 4-4
4–9 Window Menu ................................................................................... 4-4
4–10 Help Menu ....................................................................................... 4-4
4–11 The Command Window ...................................................................... 4-6
5–1 Sample Memory Map for Use With a TMS320C6x Simulator ............ 5-4
8–1 The Default Memory Window .............................................................. 8-6
8–2 Reordering Registers in the CPU Window Using the Drag-and-Drop Method ......................................................... 8-13
9–1 An Example of the Profile Window ...................................................... 9-20
9–2 Cycling Through the Profile Window Fields ......................................... 9-22
1–1 Summary of Debugger Options .......................................................... 1-7
4–1 Menu Conventions ........................................................................... 4-5
6–1 Debugger Modes and Displays ......................................................... 6-2
8–1 Pseudoregister Names for Single-Precision Floating-Point Registers ................................................. 8-15
8–2 Pseudoregister Names for Double-Precision Floating-Point Registers ................................................. 8-16
8–3 Display Formats for Debugger Data .................................................. 8-20
8–4 Data Types for Displaying Debugger Data ......................................... 8-21
9–1 Debugger Commands That Can/Cannot Be Used in the Profiling Environment ......................................... 9-4
9–2 Using the Profile Marking Dialog Box to Mark Areas ................................................. 9-9
9–3 Disabling, Enabling, Unmarking, or Viewing Areas ................................................. 9-13
9–4 Types of Data Shown in the Profile Window ......................................... 9-21
10–1 Marking Areas .............................................................................. 10-49
10–2 Disabling Marked Areas ................................................................ 10-49
10–3 Enabling Disabled Areas ................................................................ 10-50
10–4 Unmarking Areas ......................................................................... 10-51
10–5 Changing the Profile Window Display .............................................. 10-52
11–1 Predefined Constants for Use With Conditional Commands .......................... 11-3
Overview of a Code Development and Debugging System

The 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 C source debugger, describes the code development environment, and provides instructions and options for invoking the debugger.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Description of the C Source Debugger</td>
<td>1-2</td>
</tr>
<tr>
<td>1.2 Developing Code for the TMS320C6x</td>
<td>1-4</td>
</tr>
<tr>
<td>1.3 Preparing Your Program for Debugging</td>
<td>1-6</td>
</tr>
<tr>
<td>1.4 Invoking the Debugger</td>
<td>1-7</td>
</tr>
<tr>
<td>1.5 Exiting the Debugger</td>
<td>1-10</td>
</tr>
<tr>
<td>1.6 Debugging Your Programs</td>
<td>1-11</td>
</tr>
<tr>
<td>1.7 Accessing Online Help</td>
<td>1-12</td>
</tr>
</tbody>
</table>
1.1 Description of the C Source Debugger

The 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.

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

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

Figure 1–1. The Basic Debugger Display

- Pulldown menus are available for most tasks
- Toolbar icons let you select commands quickly
- C source display
- Disassembly display
- Function call traceback
- Interactive command entry with command history
- Scrolling data displays with on-screen editing
- Natural-format data displays
- Context-sensitive status bar
- Toolbars for file, view, setup, memory, target, mode, profile, window, and help
Key features of the debugger

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

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

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

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

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

- **Flexible command entry.** There are various ways to enter commands. You can use the menu options and dialog boxes, click toolbar icons or use function keys as shortcuts to often-used commands, or use a command line.

- **Dynamic profiling.** 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 and helps you identify bottlenecks within the code.

- **All the standard features you expect in a world-class debugger.** The debugger provides you with complete control over program execution with features like conditional execution and single-stepping (including single-stepping into or over function calls). You can set or clear a breakpoint with a click of the mouse. You can define a memory map that identifies the portions of target memory that the debugger can access. The debugger can execute commands from a batch file, providing you with an easy method for entering often-used command sequences.
1.2 Developing Code for the TMS320C6x

The 'C6x is well supported by a complete set of hardware and software development tools, including a C compiler, an assembly optimizer, an assembler, and a linker. Figure 1–2 illustrates the basic 'C6x code development flow.

Figure 1–2. 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 1–2.

- **The assembly optimizer** allows you to write linear assembly code without being concerned with the TMS320C6x pipeline structure or with assigning registers. It assigns registers and uses loop optimization to turn linear assembly into highly parallel assembly that takes advantage of software pipelining.

  See the *TMS320C6x Optimizing C Compiler User’s Guide* for more information.

- **The C compiler** accepts C source code and produces TMS320C6x assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are included in the compiler package:
  - The shell program enables you to compile, assemble, and link source modules in one step.
  - The optimizer modifies code to improve the efficiency of C programs.
  - The interlist utility interlists C source statements with assembly language output to correlate code produced by the compiler with your source code.

  See the *TMS320C6x Optimizing C Compiler User’s Guide* for more information.

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

  See the *TMS320C6x Assembly Language Tools User’s Guide* for more information.

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

  See the *TMS320C6x Assembly Language Tools User’s Guide* for more information.

- The main product of this development process is a module that can be executed in a **TMS320C6x target system**. You can use an instruction-accurate and clock-accurate software simulator to refine and correct your code.
1.3 Preparing Your Program for Debugging

Before you use the debugger, you must create an executable object file. To do so, start with C source, assembly optimizer source, and/or assembly language code. You can use the cl6x shell program to compile, assemble, and link your source code, creating an executable object file. To be able to debug the object file, you must use the –g shell option. The –g option generates symbolic debugging directives that are used by the debugger.

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

For more information about the cl6x shell program and its options and about creating an executable object file for use with the debugger, see the TMS320C6x Optimizing C Compiler User’s Guide.

Debugging optimized code

If you intend to debug optimized code, use the –g shell option with the –o shell option. The –g option generates symbolic debugging directives that are used by the debugger for C source debugging, but it disables many code generator optimizations. When you use the –o option (which invokes the optimizer) with the –g option, you turn on the maximum amount of optimization that is compatible with debugging.

If you have trouble debugging loops in your code, you can use the –mu shell option to turn off software pipelining. Software-pipelined loops are sometimes difficult to debug because the code is not presented serially.

Profiling optimized code

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

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

- Double-click the icon for the debugger.
- Open the directory that contains the debugger executable file. Double-click the executable file.

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

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

```
sim6x [filename] options
```

Where

- `sim6x` invokes the debugger.
- `filename` is an optional parameter that names an object file that the debugger loads into memory during invocation. The debugger looks for the file in the current directory; if the file is not in the current directory, you must supply the entire path-name. If you do not supply an extension for the filename, the debugger assumes that the extension is `.out`.
- `options` supply the debugger with additional information.

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 the information about setting up the environment variables in your getting started guide).

**Table 1–1. Summary of Debugger Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-c</code></td>
<td>Clear the <code>.bss</code> section</td>
</tr>
<tr>
<td><code>-d</code></td>
<td>Display the debugger on different machine (X Window System™ only)</td>
</tr>
<tr>
<td><code>-i</code></td>
<td>Identify additional directories that contain source files</td>
</tr>
<tr>
<td><code>-me</code></td>
<td>Select big-endian format</td>
</tr>
<tr>
<td><code>-profile</code></td>
<td>Enter the profiling environment</td>
</tr>
<tr>
<td><code>-s</code></td>
<td>Load the symbol table only</td>
</tr>
<tr>
<td><code>-t</code></td>
<td>Identify a new initialization file</td>
</tr>
<tr>
<td><code>-v</code></td>
<td>Load without the symbol table</td>
</tr>
<tr>
<td><code>-x</code></td>
<td>Ignore <code>D_OPTIONS</code></td>
</tr>
</tbody>
</table>
Invoking the Debugger

Clearing the .bss section (–c option)

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

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 your getting started guide for more information). If you use both the DISPLAY environment variable and the –d option, –d 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 as many pathnames as necessary; use the –i option with each pathname. For example:

`sim6x –i pathname1 –i pathname2 –i pathname3 …`

Using –i is similar to using the D_SRC environment variable (see the information about setting up the environment variables in your getting started 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 big-endian format (–me option)

The –me option tells the debugger that the object file to be loaded is in big-endian format. The default is little-endian format.

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. 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 most useful in an emulation environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). In such an environment, loading the symbol table allows you to perform symbolic debugging and examine the values of C variables.

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

Identifying a new initialization file (–t option)

The –t option allows you to specify your own, customized initialization command file to use instead of siminit.cmd 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 File→Load Program menu option or 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 your getting started guide).
1.5 Exiting the Debugger

To exit the debugger, use one of these methods:

☐ From File menu at the top of the debugger display, select Exit.

☐ Close the application window for the debugger.

☐ From the command line, enter:

```
quit
```
1.6 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.

**Step 1**
Prepare a C program or assembly language program for debugging.

**Step 2**
Ensure that the debugger has a valid memory map.
See Chapter 5, *Defining a Memory Map*.

**Step 3**
Load the program’s object file.

**Step 4**
Run the loaded file. You can run the entire program, run parts of the program, or single-step through the program.

**Step 5**
Stop the program at critical points and examine important information.

**Step 6**
Once you have decided what changes must be made to your program, exit the debugger, edit your source file, and return to Step 1.
1.7 Accessing Online Help

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

Accessing a list of help topics

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

1) Open the list of help topics by using one of these methods:
   - Click the Help Topics icon on the toolbar:
   - From the Help menu, select Help Topics.
   - From the command line, enter:
     `help`

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

Accessing context-sensitive help

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

To find out about an item in the debugger display, follow these steps:

1) Click the Help icon on the toolbar:

   This changes the pointer to a question mark.

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

To find out about a dialog box or window, follow these steps:

1) Make the window or dialog box active.

2) Press `F1`.

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

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

```
help  debugger command
```

The HELP command opens a help topic that describes the `debugger command`. 
The C source debugger has a window-oriented display. This chapter shows what the windows look like and describes the basic types of windows that you can use.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Debugging Modes and Default Displays</td>
<td>2-2</td>
</tr>
<tr>
<td>2.2 Descriptions of the Different Kinds of Windows and Their Contents</td>
<td>2-6</td>
</tr>
<tr>
<td>2.3 The Active Window</td>
<td>2-21</td>
</tr>
</tbody>
</table>
2.1 Debugging Modes and Default Displays

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

**Auto mode**

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

- When the debugger is running assembly language code, you see an assembly display similar to the one in Figure 2–1. The Disassembly window displays the reverse assembly of memory contents.
  
  When you first invoke the debugger, you see a display similar to this.

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

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

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

**Assembly mode**

*Assembly mode* is for viewing assembly language programs only. In this mode, you see a display similar to the one shown in Figure 2–1. When you are in assembly mode, you always see the assembly display, regardless of whether C or assembly language is currently running.

In assembly mode, the debugger automatically displays the Memory window, the Disassembly window, the CPU register window, and the Command window. In addition to these windows, you can open Watch windows and additional Memory windows.
Figure 2–1. Typical Assembly Display (for Auto Mode and Assembly Mode)
Debugging Modes and Default Displays

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

**Mixed mode**

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

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

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

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

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

- The CALLS, DISP, FUNC, and FILE commands are valid only in auto and mixed modes.
- The MEM command is valid only in assembly and mixed modes.
2.2 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. For the File window, the debugger displays the name of the file shown in the window instead of the word File. Each type of window serves a specific purpose and has unique characteristics. There are eight different windows, divided into these general categories:

- Code-display windows display assembly language or C code. There are three code-display windows:
  - A File window displays any text file that you want to display; its main purpose, however, is to display C source code. You can display multiple File windows at one time.
  - The Disassembly window displays the disassembly (assembly language version) of memory contents.
  - The Calls window 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 three 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 Watch window displays selected data such as variables, specific registers, or memory locations. You can display multiple Watch windows at one time.

- The Command window provides an area for typing in commands and re-entering commands and an area for displaying various types of information, such as progress messages, error messages, or command output.
Descriptions of the Different Kinds of Windows and Their Contents

All of the windows have context menus that you can use to perform tasks such as displaying or hiding information in a window and controlling how a window is displayed. To display a context menu, follow these steps:

1) Move your pointer over a debugger window.

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

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

The remainder of this section describes the individual windows.
**File window**

**Purpose**
Shows any text file you want to display. The text file name appears as the label for the window.

**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→Open menu option
- With the FILE command
- Automatically when you are in auto or mixed mode and your program begins executing C code or begins executing assembly code assembled with the –g assembler option

**Affected by**
- File→Open menu option
- FILE,FUNC, and ADDR commands
- Breakpoint and run commands

You can use the File→Open menu option to display the contents of any file within a 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. When your program enters a new C module, the debugger opens a new File window to show the new C file.

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 have set a software breakpoint
If you assemble your assembly code with the –g assembler option, the File window displays the contents of your assembly source file. This allows you to view all assembly source comments and true assembly statements:

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

The context menu for the File window allows you to:

- Display or hide the line numbers in the File window.
- Set or clear a breakpoint on the current line. (The current line is shown with a heavy band across the line.)
- Run the processor to the current line.
- Set the PC to the current line.

For more information about using a context menu, see page 2-7.
**Descriptions of the Different Kinds of Windows and Their Contents**

**Disassembly window**

<table>
<thead>
<tr>
<th>Address:</th>
<th>00000000 074a82a 08bb00</th>
<th>MVK S2</th>
<th>0-afff3-f50.B15</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000004</td>
<td>074a82a</td>
<td>MVK S2</td>
<td>0-afff3-f50.B15</td>
</tr>
<tr>
<td>00000006</td>
<td>074a82a</td>
<td>MVK S2</td>
<td>0-afff3-f50.B15</td>
</tr>
<tr>
<td>00000008</td>
<td>074a82a</td>
<td>MVK S2</td>
<td>0-afff3-f50.B15</td>
</tr>
<tr>
<td>00000010</td>
<td>074a82a</td>
<td>MVK S2</td>
<td>0-afff3-f50.B15</td>
</tr>
<tr>
<td>00000012</td>
<td>074a82a</td>
<td>MVK S2</td>
<td>0-afff3-f50.B15</td>
</tr>
<tr>
<td>00000014</td>
<td>074a82a</td>
<td>MVK S2</td>
<td>0-afff3-f50.B15</td>
</tr>
<tr>
<td>00000016</td>
<td>074a82a</td>
<td>MVK S2</td>
<td>0-afff3-f50.B15</td>
</tr>
<tr>
<td>00000018</td>
<td>074a82a</td>
<td>MVK S2</td>
<td>0-afff3-f50.B15</td>
</tr>
<tr>
<td>00000020</td>
<td>074a82a</td>
<td>MVK S2</td>
<td>0-afff3-f50.B15</td>
</tr>
<tr>
<td>00000022</td>
<td>074a82a</td>
<td>MVK S2</td>
<td>0-afff3-f50.B15</td>
</tr>
<tr>
<td>00000024</td>
<td>074a82a</td>
<td>MVK S2</td>
<td>0-afff3-f50.B15</td>
</tr>
</tbody>
</table>

**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**

- Target → Restart menu option
- RESTART, 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
- All statements associated with the current C statement, as shown in Figure 2–4
Figure 2–4. Assembly Language Statements Associated With a C Statement

The context menu for the Disassembly window allows you to:

- Display or hide the addresses and opcodes in the Disassembly window.
- Set or clear a breakpoint on the current line. (The current line is shown with as a heavy band across the line.)
- Run the processor to the current line.
- Set the PC to the current line.
- Display or hide the Address combo box.
- Control how the Disassembly window is displayed

For more information about using a context menu, see page 2-7.
Calls window

Purpose
Lists the function you are in, its caller, and the caller’s caller, etc., as long as each function is a C function.

Editable?
No; pressing the edit key (F9) or double-clicking a function name changes the File and Disassembly windows to show the source associated with the called function.

Modes
Auto (C display only) and mixed.

Created
- Automatically when you are displaying C code
- With the CALLS command if you previously 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 by double-clicking the function name.
Profile window

Descriptive table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Area Name</th>
<th>Count</th>
<th>Inclusive</th>
<th>Incl-Max</th>
<th>Exclusive</th>
<th>Excl-Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Function f1()</td>
<td>4</td>
<td>9630</td>
<td>9628</td>
<td>105</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>C Function f2()</td>
<td>4</td>
<td>13980</td>
<td>8384</td>
<td>116</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>C Function f3()</td>
<td>4</td>
<td>13980</td>
<td>8384</td>
<td>116</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>C Function main()</td>
<td>1</td>
<td>26547</td>
<td>26547</td>
<td>36</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

**Purpose**: Displays statistics collected during a profiling session

**Editable?**: No

**Modes**: Mixed

**Created by**: By entering the profiling environment: Profile \(\rightarrow\) Profile Mode

**Affected by**:
- Profile \(\rightarrow\) Change View, Profile \(\rightarrow\) Select Areas, and Profile \(\rightarrow\) Run menu options
- PF and PQ commands
- Clicking in the header area of the window to change the type of data displayed in 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.

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), see Chapter 9, *Profiling Code Execution*. 
Descriptions of the Different Kinds of Windows and Their Contents

Memory window

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 for the default Memory window only
With the MEM command and a unique window name for additional Memory windows
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 associated with 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 4 (0x4). Although the window shows three columns of data, there is still only one column of addresses; the first value is at address 0x0000 7C68, the second is at address 0x0000 7C6C, etc.; the fourth value (first value in the second row) is at address 0x0000 7C74, the fifth is at address 0x0000 7C78, 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.

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 and named RANGE1, enter:

```
mem 0x8000,,RANGE1
```

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

By default, memory contents are shown in hexadecimal format. You can use the optional `display format` parameter to display the data in another format, as shown in Table 8–3 on page 8-20.

Changing the memory range shown in a Memory window

To change the memory range shown in a Memory window, enter a new starting address in the Address field of the Memory window:

You can enter an absolute address, a symbolic address, or any C expression. 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. If you enter a value that the debugger does not recognize (for example, you enter ffff with no 0x prefix), the debugger displays an error in display area of the Command window.
The context menu for the Memory window allows you to:

- Display or hide the Address combo box.
- Control how the Memory window is displayed

For more information about using a context menu, see page 2-7.
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.

You can reorder the registers in the CPU window and display the ones that you are most interested in at the top of the CPU window. To do so, use the drag-and-drop method:
**Watch window**

A Watch window helps you track the values of expressions, variables, arrays, structures, 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 a Watch window to track the values of the specific registers you are interested in.

To display a value in a Watch window, follow these steps:

1) From the Setup menu, select Watch Variable. This displays the Watch add dialog box shown in Figure 2–5.

2) In the Expression field, enter item that you want to watch. The expression can be any C expression, including an expression that has side effects.

3) If you want to assign a label for the watched item, use the Label field. If you leave the Label field blank, the debugger displays the expression as the label.

4) If you want to change the data format for the watched item, use the Format field. The data formats that you can use are shown in Table 8–3 on page 8-20. The Format field is optional.

5) If you want to open a new Watch window, enter a name for the new Watch window in the Window name field. This field is optional.

6) Click OK.
The Debugger Display

Figure 2–5. Watch add Dialog Box

The Watch window uses a box icon preceding a watched item to indicate that the item is a structure, pointer, or array. You can display the additional data (the data pointed to or the members of the array or structure) by clicking the box icon:

You can also display additional data by selecting an item and pressing SPACE.

To delete an entry from a Watch window, follow these steps:
1) Select the item in the Watch window that you want to delete.
2) Press DELETE.
Descriptions of the Different Kinds of Windows and Their Contents

Command window

The Command window has two parts:

- **Command line.** This is where you enter commands. The debugger keeps a list of the last 100 commands that you entered. You can use the command-history combo box to select and reenter commands from a list without retyping them:

- **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, see Section 4.2, *Entering Commands From the Command Line*, on page 4-6.
2.3 The Active Window

The debugger allows you to designate a single debugger window as the *active window*. The active window allows you to tell the debugger which window you want to affect.

The debugger highlights the active window by changing the appearance of the window title bar, as shown in Figure 2–6.

*Figure 2–6. Default Appearances of an Active and an Inactive Window*

(a) An active window
This window title bar is highlighted to show that it is active

(b) An inactive window
This window title bar is not highlighted and is not active

To make a window active, choose one of these methods:

- Click the window.
- From the View menu, select the window that you want to make active.
Use one of the following key combinations:
- **ALT F1** makes the Command window active
- **ALT F2** makes the Disassembly window active
- **ALT F3** makes the Memory window active
- **ALT F4** makes the CPU window active
- **ALT F5** makes the Calls window active
- **ALT F6** makes the Profile window active

Press **CONTROL F6** to cycle through the windows in the debugger display, making each window active in turn and making the previously active window inactive.
Customizing the Debugger Display

The debugger’s graphical user interface follows the windowing conventions of your windowing system. You can size and move the debugger windows in the same way that you size and move windows in your other windowing applications.

You can also customize your debugger display by changing the way windows are displayed, hiding specific windows, and moving the toolbar. Once you have customized the display to your liking, you can save the custom configuration for use in future debugging sessions.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Managing Debugger Windows</td>
<td>3-2</td>
</tr>
<tr>
<td>3.2 Displaying the Status Bar</td>
<td>3-7</td>
</tr>
<tr>
<td>3.3 Displaying and Moving the Toolbar</td>
<td>3-8</td>
</tr>
<tr>
<td>3.4 Changing the Prompt</td>
<td>3-12</td>
</tr>
<tr>
<td>3.5 Saving and Using Custom Displays</td>
<td>3-13</td>
</tr>
</tbody>
</table>
3.1 Managing Debugger Windows

When you first invoke the debugger, all of the debugger windows are in one main debugger application window. If you move one of the debugger windows to an outer edge of the main application window, a scroll bar appears on the application window, allowing you to scroll the main application window to see the partially hidden window.

Floating windows outside the application window

You can make a debugger window float outside the debugger application window. This means that the floating window always sits on top of the debugger application window, and you can move the floating window anywhere on the debugger display.

For example, if you move a nonfloating window to the top of the debugger application window, the nonfloating window never covers the menu bar;
instead, the window appears to slide under the menu bar. Unlike a nonfloating window, you can move a floating window anywhere on the screen and even cover the toolbar and menu bar.

A floating window always appears active—its title bar is always highlighted.

You can make any or all of the debugger windows floating windows except the File windows. To make a window a floating window, be sure that the Float In Main Window context-menu option is not selected:

A floating window can overlap other debugger windows or even the debugger menu bar. When a floating window moves off of the application window, you do not see a scroll bar in the debugger application window:

Even though the Calls window has moved off of the screen, debugger application window does not have a scroll bar.
Docking debugger windows

You can make any floating window a docked window. When a window is docked, you do not see the title bar for the window. This frees up space in your debugger application window for additional windows or for larger windows. A docked window looks like this:

You can dock any or all of the debugger windows except the File windows. To dock a debugger window, follow these steps:

1) Make the window a floating window; be sure that the context menu for the window has Float In Main Window disabled.

2) In the context menu for the window, be sure that Allow Docking is enabled. (Be sure there is a check mark next to Allow Docking.)

3) Drag the window so that it aligns with one of the borders of the application window or aligns against another debugger window. The window’s border changes when the window is in a dockable location.

4) Release the window.
Managing Debugger Windows

Here is an example of multiple docked windows:

- **Command window**
- **Memory window**
- **Disassembly window**
Closing debugger windows

Sometimes you might want to close a window to free up space on your screen.

To close any debugger window (except the File window), select the Hide context-menu option.

Alternatively, you can close any window (including the File window) by making that window active and pressing `CONTROL`+`F4`.

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.

To reopen a closed window (except the File window), select the window name from the View menu.

To reopen a File window, click the Open icon on the toolbar:

![Open icon]

You can also use the File menu to reopen a file. The File menu lists the most recent files that you have opened.
3.2 Displaying the Status Bar

By default, the debugger application window displays a status bar. The status bar tells you whether the processor is running and provides context-sensitive help.

When your cursor is not over a particular debugger window, the status bar tells you how to access online help.

The status bar tells you when the processor is running.

When you move your cursor over a debugger window, the status bar displays information about that window. For example:

When your cursor is over the Memory window, the status bar displays this.

You can turn off the status bar by using the View menu. Be sure that the Status Bar option is not selected. (There is no check mark next to the option.)
Displaying and Moving the Toolbar

3.3 Displaying and Moving the Toolbar

By default, the debugger application window displays a toolbar under the menu bar. The toolbar provides quick access to commonly used commands.

Click icons on the toolbar to select commands quickly

You can move the toolbar from under the menu bar to another part of the debugger application window or make the toolbar a window of its own (undock the toolbar). You can also hide the toolbar.

Moving the toolbar

You can move the toolbar to any of the outer borders of the debugger application window. To move the toolbar, follow these steps:

1) Move your pointer between any two icons on the toolbar.
2) Drag the toolbar to any border of the debugger application window:
3) Release the toolbar.

The toolbar moves to the new location:
**Undocking the toolbar**

You can make the toolbar a window of its own. This undocks the toolbar and allows you to move it around freely in your debugger application window.

To undock the toolbar, follow these steps:

1) Move your pointer between any two icons on the toolbar.
2) Drag the toolbar anywhere in the application window:

3) Release the toolbar.
Displaying and Moving the Toolbar

The toolbar is now its own window. You can move it around like any other floating window.

Hiding the toolbar

If you choose not to use the toolbar and want to save space in your debugger display, you can hide the toolbar. To display or hide the toolbar, use the View menu:

- When there is a check mark next to the Toolbar option, the toolbar is displayed.
- When there is no check mark next to the Toolbar option, the toolbar is hidden.
3.4 Changing the Prompt

The debugger allows you to change the command-line prompt with 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. The new prompt cannot be longer than 132 characters.

When you save your screen configuration (as described in Section 3.5), the debugger does not save the command-line prompt as part of a custom configuration. If you always want to use a customized prompt, you can add a PROMPT command to the init.cmd batch file that the debugger executes at invocation.
3.5 Saving and Using Custom Displays

The debugger allows you to save a custom screen configuration into a file and restore a screen configuration from a file. The screen-configuration files are binary files, not text files, so you cannot edit the files with a text editor.

**Saving a custom display**

Once you have customized the debugger display to your liking, you can save the current screen configuration to a file. To do so, follow these steps:

1) From the File menu, select Save As Config. This displays the Save Configuration File dialog box:

![Save Configuration File dialog box](image)

   - Enter a name for the configuration file. Use a .clr extension.
   - Select the directory where you want the file to be saved.
   - In the File name field, enter a name for the configuration file. You can use a .clr extension to identify the file as a screen-configuration file.

2) Click Save.

   This saves the window locations and window sizes for all debugging modes, including the size and location for multiple File, Watch, and Memory windows. However, the debugger does not save docking information about docked windows. If you have one or more docked windows, you save the screen configuration, and you load that configuration, the debugger does not display any windows as docked. If you want the windows docked, you must follow the docking procedure again.
Loading a custom display

You can restore a particular configuration to the display. To do so, follow these steps:

1) From the File menu, select Load Config. This displays the Open Configuration File dialog box:

   ![Open Configuration File dialog box]

   - You can change the directory that you want to search.
   - Select from a list of files.

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

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

3) Click Open.

This restores the window locations and sizes saved in the selected file.

When you use File→Load Config to restore a configuration that includes multiple File, Watch, or Memory windows, the additional windows are not opened automatically. However, when you open an additional window using the same name as in this saved configuration, the window is placed in the saved location.
Saving and Using Custom Displays

**Note:**
The file created by the File→Save As Config or SSAVE command in this version of the debugger saves screen-configuration 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 screen-configuration 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 can 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:

```
scconfig filename
```

The debugger searches for the file in the current directory and then in directories named with the D_DIR environment variable.

If you have changed the configuration and would like to revert to the default configuration, enter the SCONFIG command without a filename.
The debugger provides you with several methods for entering commands:

- From the toolbar
- From the menu bar
- With function keys
- From the command line
- From a batch file

This chapter contains instructions for using the menu bar, the toolbar, and the command line to enter commands. This chapter also describes how you can create aliases for commands and commands sequences that you enter frequently. For information about using a batch file for entering commands, see Chapter 11, *Using Batch Files to Enter Commands*.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Using the Menu Bar and the Toolbar</td>
<td>4-2</td>
</tr>
<tr>
<td>4.2 Entering Commands on the Command Line</td>
<td>4-6</td>
</tr>
<tr>
<td>4.3 Defining Your Own Command Strings</td>
<td>4-10</td>
</tr>
<tr>
<td>4.4 Entering Operating-System Commands</td>
<td>4-13</td>
</tr>
</tbody>
</table>
4.1 Using the Menu Bar and the Toolbar

At the top of the default debugger window are the menu bar and the toolbar. The menu selections and the toolbar offer you several methods for entering debugger commands. Figure 4–1 illustrates the basic menu bar and the toolbar.

Figure 4–1. The Menu Bar and the Toolbar

Click icons on the toolbar to select commands quickly
Use the mouse to select menu options

Overview of the debugger menus

Figure 4–2 through Figure 4–10 show each of the debugger menus and briefly describe the tasks you can perform with each menu. The individual menu options are discussed in the various sections throughout this book as they apply to the topic that is being discussed.

Figure 4–2. File Menu

The File menu allows you to:

- Open files (such as C files) for viewing
- Load and reload object files and symbol tables
- Create and execute batch files
- Save and load the screen configuration
- Exit the debugger
Using the Menu Bar and the Toolbar

**Figure 4–3. View Menu**

![View Menu](image)

The View menu allows you to:
- Hide or show the toolbar and the status bar
- Make an individual window active

**Figure 4–4. Setup Menu**

![Setup Menu](image)

The Setup menu allows you to:
- Set up breakpoints
- Open a Watch window to display data values
- Customize debugger commands (create aliases)

**Figure 4–5. Memory Menu**

![Memory Menu](image)

The Memory menu allows you to:
- Create or modify your memory map
- Save memory values
- Fill a block of memory

**Figure 4–6. Target Menu**

![Target Menu](image)

The Target menu allows you to:
- Single-step through C code, assembly code, or both
- Execute your entire program
- Restart execution at the original program entry point
- Reset the simulator or the processor
Using the Menu Bar and the Toolbar

Figure 4–7. Mode Menu

The Mode menu allows you to select the debugging mode: C (auto) mode, assembly mode, or mixed mode.

Figure 4–8. Profile Menu

The Profile menu allows you to:
- Enter the profiling environment
- Select the data that you want to view in the Profile window
- Define areas for profiling
- Run a profiling session
- Save profile data to a file

Figure 4–9. Window Menu

The Window menu allows you to:
- Arrange the debugger windows
- Make individual windows active (except the File window)

Figure 4–10. Help Menu

The Help menu allows you to:
- Access the debugger help topics
- Access copyright and version information about the debugger
Using the Menu Bar and the Toolbar

Selecting menu options

To open a menu and choose a menu option, follow these steps:

1) Click the name of the menu that you want to open. This opens the menu.
2) Click the menu option that you want to execute.

You can also open a menu by using the \texttt{ALT} key. Press \texttt{ALT} then type the letter that is underlined in the menu name. To select a menu option from an open menu, type the letter that is underlined in the option name.

Understanding the menu conventions

The debugger uses the menu conventions listed in Table 4–1.

Table 4–1. Menu Conventions

<table>
<thead>
<tr>
<th>Menu Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A menu or menu option is dimmed or not visible</td>
<td>You cannot access this menu or menu option currently. For example, the Halt! option on the Target menu is dimmed when the processor is not running. When the processor is running, you can access the Halt! option, so it is not dimmed then.</td>
</tr>
<tr>
<td>An ellipsis (\ldots) follows a menu option</td>
<td>When you choose this menu option, a dialog box is displayed in which you must supply or confirm some information.</td>
</tr>
<tr>
<td>A check mark is next to a menu option</td>
<td>The menu option is in effect. For example, when the toolbar is displayed, you see a check mark next to the Toolbar option on the View menu.</td>
</tr>
<tr>
<td>A key combination is next to a menu option</td>
<td>You can use the key combination to select the menu option (without having to open the menu).</td>
</tr>
</tbody>
</table>

Overview of the toolbar

The toolbar provides you with quick access to commonly used commands. Here are the commands that you can execute from the toolbar:

To execute one of these commands, click the toolbar icon for the command.
4.2 Entering Commands on the Command Line

In addition to the menu bar and toolbar, the debugger supports a set of commands that help you to control and monitor program execution and perform other tasks. Chapter 10, *Summary of Commands*, summarizes all of the debugger commands with an alphabetical reference.

You can enter a command by using the Command window. Figure 4–11 shows the Command window.

*Figure 4–11. The Command Window*

The Command window serves two purposes:

- The command line portion of the window provides you with an area for typing commands. For example, the command line in Figure 4–11 shows that a RUN command was typed in (but not yet entered). You can view and select from a list of commands that you previously entered by clicking the arrow at the end of the command line or by pressing ↓. For more information about command history, see page 4-9.

- The display area provides the debugger with a space for echoing commands, displaying command output, or displaying errors and messages for you to read.

If you enter a command through an alternate method (using a menu, toolbar icon, or function keys), the Command window does not echo the entered command.
Typing in and entering commands

To enter a command from the Command window, you must be sure that the cursor is in the command-line area of the Command window:

![Image showing Command window with commands]

Commands are not case sensitive.

To execute a command that you have typed, press Enter. The debugger then:

1) Echoes the command to the display area
2) Executes the command and displays any resulting output

The debugger does not clear the command line when command execution completes. This allows you to reenter the same command easily by pressing Enter.

You can edit the text on the command line with these keystrokes:

- Moving through the command line

<table>
<thead>
<tr>
<th>To...</th>
<th>Press...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move cursor to the left without deleting characters</td>
<td>←</td>
</tr>
<tr>
<td>Move cursor to the right without deleting characters</td>
<td>→</td>
</tr>
<tr>
<td>Move to the beginning of the previous word without deleting characters</td>
<td>CONTROL ←</td>
</tr>
<tr>
<td>Move to the beginning of the next word without deleting characters</td>
<td>CONTROL →</td>
</tr>
<tr>
<td>Move to the beginning of the line without deleting characters</td>
<td>HOME</td>
</tr>
<tr>
<td>Move to the end of the line without deleting characters</td>
<td>END</td>
</tr>
<tr>
<td>Move the characters after the cursor to the right</td>
<td>SPACE</td>
</tr>
</tbody>
</table>
Modifying text on the command line

<table>
<thead>
<tr>
<th>To...</th>
<th>Press...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert characters into the text on the command line</td>
<td>INSERT</td>
</tr>
<tr>
<td>Copy highlighted text</td>
<td>CONTROL C</td>
</tr>
<tr>
<td>Paste text from the windowing system clipboard</td>
<td>CONTROL V</td>
</tr>
</tbody>
</table>

Selecting text on the command line

<table>
<thead>
<tr>
<th>To...</th>
<th>Press...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select one character to the left of the cursor</td>
<td>SHIFT ←</td>
</tr>
<tr>
<td>Select one character to the right of the cursor</td>
<td>SHIFT →</td>
</tr>
<tr>
<td>Select all characters to the left of the cursor</td>
<td>SHIFT HOME</td>
</tr>
<tr>
<td>Select all characters to the right of the cursor</td>
<td>SHIFT END</td>
</tr>
</tbody>
</table>

Deleting text from the command line

<table>
<thead>
<tr>
<th>To...</th>
<th>Press...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete the character to the left of the cursor</td>
<td>CONTROL H or BACK SPACE</td>
</tr>
<tr>
<td>Delete the character to the right of the cursor</td>
<td>DEL</td>
</tr>
<tr>
<td>Delete highlighted text</td>
<td>DEL or CONTROL X</td>
</tr>
</tbody>
</table>
Using the command history

The debugger keeps an internal list, or command history, of the commands that you enter. It tracks the last 100 commands that you entered. If you want to reenter a command, you can move through this list, select a command that you have already executed, and reexecute it.

To use the command history, click the arrow at the end of the command line. This displays the list of commands that you have entered from the command line:

To reenter command, select the command from the list and press F2.

To repeat the last command that you entered, press F2.

Clearing the display area

Occasionally, you may want to completely clear all information shown in the display area of the Command window. To do so, use the CLS command. The format for this command is:

cls
4.3 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 allows you to define an alias name for the command(s) and then enter the alias name as if it were a debugger command.

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

- **List of defined aliases**
  - INIT: load test.out; file source.c; go main
  - RESTRUN: restart; run

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

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

To define an alias, follow these steps:

1) From the Setup menu, select Alias Commands. This displays the Alias Control dialog box.
2) In the Name field, enter a name for the alias.
3) In the Command string field, enter the command string that you want to associate with the alias name. If you want to associate multiple commands with the alias, separate the commands with a semicolon.
4) Click Apply.
5) Click OK to close the Alias Control dialog box.

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

Defining an alias with parameters

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

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

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

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

**Editing or redefining an alias**

To edit or redefine an alias, follow these steps:

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

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

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

4) Click Apply.

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

**Deleting an alias**

To delete an alias, follow these steps:

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

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

3) Click Delete.

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

**Considerations for using alias definitions**

Alias definitions are lost when you exit the debugger. If you want to reuse aliases, define them in a batch file. Use the ALIAS command, as described on page 10-11.

Individual commands within a command string are limited to an expanded length of 132 characters. The expanded length of the command includes the length of any substituted parameter values.
4.4 Entering Operating-System Commands

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

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

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

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

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

**Entering a single command from the debugger command line**

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

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

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

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

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

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

The message would be displayed until you pressed Enter. If you wanted the debugger to display the message and then return immediately to the debugger environment, you could enter the command in this way:

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

```
Entering Operating-System Commands

Entering several commands from a system shell

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

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

exit
Defining a Memory Map

Before you begin a debugging session, you must supply the debugger with a memory map. The memory map tells the debugger which areas of memory it can and cannot access.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 The Memory Map: What It Is and Why You Must Define It</td>
<td>5-2</td>
</tr>
<tr>
<td>5.2 A Sample Memory Map</td>
<td>5-4</td>
</tr>
<tr>
<td>5.3 Creating or Modifying the Memory Map</td>
<td>5-5</td>
</tr>
<tr>
<td>5.4 Enabling Memory Mapping</td>
<td>5-8</td>
</tr>
<tr>
<td>5.5 Defining a Memory Map in a Batch File</td>
<td>5-10</td>
</tr>
<tr>
<td>5.6 Returning to the Original Memory Map</td>
<td>5-11</td>
</tr>
<tr>
<td>5.7 Simulating I/O Space</td>
<td>5-12</td>
</tr>
</tbody>
</table>
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 cannot 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 cannot 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 enables you to modify the default memory map or define a new memory map interactively (as described in Section 5.3 on page 5-5) or by 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, which executes when you invoke the debugger.

- You can define the memory map in a separate batch file of your own that you can execute when you invoke the debugger (using the –t debugger option) or from the debugger command line (using the File→Take menu option).

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

1) It checks to see whether you have used the –t debugger option. The –t option allows you to specify a batch file other than the initialization batch file shipped with the debugger. If it finds the –t option, the debugger reads and executes the specified file. See Section 5.5 on page 5-10 for more information about defining the memory map in a batch file.

2) If you do not use the –t option, the debugger looks for the default initialization batch file. The batch filename for the simulator is called siminit.cmd. 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. If the debugger finds the file, it reads and executes the file.
Potential memory map problems

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

- **Accessing invalid memory addresses.** If you do not supply a batch file containing memory-map commands, the debugger is initially unable to access any target memory locations. Invalid memory addresses and their contents are highlighted in the data-display windows. (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 defined in a batch file or with the Memory Map Control dialog box (described in Section 5.3 on page 5-5). Alternatively, you can turn memory mapping off during a load by disabling memory mapping (described in Section 5.4 on page 5-8). When you turn memory mapping off, you can still access memory locations. However, the debugger does not prevent you from accessing memory locations that you have not defined as valid in the memory map.
5.2 A Sample Memory Map

Because you must define a memory map before you can run any programs, it is convenient to define the memory map in the initialization batch files. Figure 5–1 shows the memory map that is defined in the initialization batch file that accompanies the 'C6x simulator. 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. (For more information about the MA command, see Section 5.5 on page 5-10.) 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

(b) Memory map for TMS320C6x local memory

| ma 0x0, 0x400, RAM          | 0x0000 0000 to 0x0000 3FF Read/write memory |
| ma 0x400, 0xFC00, RAM       | 0x0000 0400 to 0x0000 FFFF Read/write memory |
| ma 0x80000000, 0x10000, RAM | 0x0001 0000 to 0x7FFF FFFF Read/write memory |
|                            | 0x8000 0000 to 0x8000 FFFF Reserved          |
|                            | 0x8001 0000 to 0xFFFF FFFF Reserved           |
5.3 Creating or Modifying the Memory Map

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

**Adding a range of memory**

To add a range of memory, follow these steps:

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

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

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

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

   - Read/write
   - Read-only
   - Write-only
   - No access
   - Input port
   - Output port
   - Input/output port

5) Click Apply.

6) Click OK.

The following restrictions apply to identifying usable memory ranges:

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

- Be sure that the map ranges that you specify in a COFF file match those that you define with the Memory Map Control dialog box. Moreover, the origin and length values for a range that you define with the MEMORY directive in your linker command file must match the Start and Length values for the same range in the Memory Map Control dialog box.

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

Deleting a range of memory

To delete a range of memory, follow these steps:

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

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

3) Click Delete.

4) Click OK.
Modifying a defined range of memory

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

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

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

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

4) Click Apply.

5) Click OK.
5.4 Enabling Memory Mapping

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

Click here to enable/disable memory mapping

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

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

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

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

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

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

Error in expression
5.5 Defining a Memory Map in a Batch File

You can create a batch file that contains memory map commands. This provides you with a convenient way to define a memory for each debugging session.

To define a memory map in a batch file, use the MA command. The syntax for the MA command is:

\texttt{ma address, length, type}

- The \textit{address} parameter defines the starting address of a range. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. If you want to specify a hex address, be sure to prefix the address number with \texttt{0x}; otherwise, the debugger treats the number as a decimal address.

- The \textit{length} parameter defines the length of the range. This parameter can be any C expression.

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

<table>
<thead>
<tr>
<th>To identify this kind of memory . . .</th>
<th>Use this keyword as the \textit{type} parameter . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-only memory</td>
<td>R or ROM</td>
</tr>
<tr>
<td>Write-only memory</td>
<td>W or WOM</td>
</tr>
<tr>
<td>Read/write memory</td>
<td>R</td>
</tr>
<tr>
<td>No-access memory</td>
<td>PROTECT</td>
</tr>
<tr>
<td>Input port</td>
<td>INPORT or P</td>
</tr>
<tr>
<td>Output port</td>
<td>OUTPORT or P</td>
</tr>
<tr>
<td>Input/output port</td>
<td>IOPORT or P</td>
</tr>
</tbody>
</table>

The memory ranges that you define have the same restrictions as those defined for the Memory→Mapping menu option described in Section 5.3 on page 5-5.

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

- Use the –t debugger option to specify the batch file when you invoke the debugger. For more information, see page 1-9.

- Use the TAKE command. For more information, see Section 11.2, \textit{Executing a Batch File}, on page 11-6.
5.6 Returning to the Original Memory Map

If you modify the memory map during a debugging session, you may want to go back to the original memory map without quitting and reinvoking the debugger. You can do this by resetting the memory map and then using the 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. (You could put the MR command in the batch file, preceding the commands that define the memory map.) The TAKE command tells the debugger to execute commands from the specified batch file.
5.7 Simulating I/O Space

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

Connecting an I/O port

The MC (memory connect) command connects INPORT, OUTPORT, or IOPORT memory to an input or output file. The syntax for this command is:

\[
\text{mc} \quad \text{port address, length, filename, \{READ | WRITE\}}
\]

- The *port address* parameter defines the address of the I/O port. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. If you want to specify a hex address, be sure to prefix the address number with \text{0x}; otherwise, the debugger treats the number as a decimal address.

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

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

- The final parameter is specified as \text{READ} or \text{WRITE} and defines how the file will be used (for input or output, respectively).

The file is accessed during an IN or OUT instruction to the associated port address. Any port in I/O space can be connected to a file. A maximum of one input and one output file can be connected to a single port; multiple ports can be connected to a single file.

Example 5–1 shows how an input port can be connected to an input file named in.dat.
Example 5–1. Connecting an Input Port to an Input File

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

0A00
1000
2000
.
.
.

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

\begin{verbatim}
MA 0x50,2,IOPORT
MC 0x50,2,in.dat,READ
\end{verbatim}

Disconnecting an I/O port

Before you can use the MD command to delete a port from the memory map, you must use the MI command to disconnect the port. The syntax for this command is:

\begin{verbatim}
mi port address, {READ | WRITE}
\end{verbatim}

The port address identifies the port that is to be closed. The read/write characteristics must match the parameter used when the port was connected.
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.
6.1 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 ways. The debugger has three 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

Table 6–1 summarizes the modes and displays; for a complete description of the three debugging modes, see Section 2.1, Debugging Modes and Default Displays, on page 2-2.

**Table 6–1. Debugger Modes and Displays**

<table>
<thead>
<tr>
<th>Use this mode...</th>
<th>To view this type of code...</th>
<th>The debugger uses these code-display windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>assembly</td>
<td>assembly language code only (even if your program is executing C code)</td>
<td>Disassembly</td>
</tr>
<tr>
<td>auto</td>
<td>assembly language code (when that is what your program is running)</td>
<td>Disassembly</td>
</tr>
<tr>
<td>auto</td>
<td>C code only (when that is what your program is running)</td>
<td>File Calls</td>
</tr>
<tr>
<td>auto</td>
<td>an assembly source code file (when you assemble the file with the –g assembler option)</td>
<td>File</td>
</tr>
<tr>
<td>mixed</td>
<td>both assembly language and C code</td>
<td>Disassembly File Calls</td>
</tr>
</tbody>
</table>

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

When you first invoke the debugger, it automatically comes up in auto mode. You can then choose assembly or mixed mode. To change the debugging mode, use the Mode menu:

You can also use F3 to change the debugging mode. Pressing this key causes the debugger to switch modes in this order:
6.2 Loading and Displaying Assembly Language Code

To debug a program, you must load the program’s object code into memory. You create an object file by compiling, assembling, and linking your source files; see Section 1.3, Preparing Your Program for Debugging, on page 1-6.

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

**Loading an object file and its symbol table**

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

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

   ![Load Program File dialog box](image)

   - You can change the directory that you want to search
   - Select from a list of files

   1. File name: [Input field]
   2. Files of type: [Program Files (*.out)]

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

3) Click Open.
Loading an object file without its symbol table

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

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

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

```
reload object filename
```

Loading a symbol table only

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

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

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

Loading code while invoking the debugger

You can load an object file when you invoke the debugger. (This has the same effect as using the File→Load Program menu option described on page 6-4.) To do this, enter the appropriate debugger-invocation command along with the name of the object file.

If you want to load only a file’s symbol table when you invoke the debugger, use the –s option. (This has the same effect as using the File→Load Symbols menu option.) To do this, enter the appropriate debugger-invocation command along with the name of the object file and specify –s (see page 1-9 for more information).
Displaying portions of disassembly

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

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

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

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.
Loading and Displaying Assembly Language Code

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

Displaying assembly source code

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

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

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

Displaying the contents of a text file

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

1) Use one of these methods to open the Open File dialog box:
   - Click the Open icon on the toolbar:
   - From the File menu, select Open.

This displays the Open File dialog box:

![Open File dialog box]

- Select from a list of files
- You can change the directory that you want to search
- Select the type of file you want to open
- File name: [Enter file name]
- Files of type: [Select file type]

Open File dialog box:

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

Open dialog box:

- [Open] button
- [Cancel] button
2) Select the file that you want to open. To do so, you might need to do one or more of the following:
   - Change the working directory.
   - Select the type of file that you want to open (for example, .c, .h).

3) Click Open.

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

Displaying a C file does not load that file’s object code. If you want to be able to run the program, you must load the file’s associated object code as described in Section 6.2, Loading and Displaying Assembly Language Code, on page 6-4.

**Displaying a specific C function**

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

```
func { function name | address }
```

FUNC modifies the display so that the code associated with the function or address that you specify is displayed within 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.

FUNC works similarly to the Open File dialog box, but you do not need to identify the name of the file that contains the function.

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

- Double-click the name of the C function.
- Select the name of the C function and press \( F9 \).
Displaying C Code

Displaying code beginning at a specific point

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

```plaintext
addr { address | function name }
```

In a C display, ADDR works like the FUNC command, positioning the code starting at `address` or at `function name` as the first line of code in the File window. In mixed mode, ADDR affects both the File and Disassembly windows.
To debug your programs, you must execute them on a 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.

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

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

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Defining the Starting Point for Program Execution</td>
<td>7-2</td>
</tr>
<tr>
<td>7.2 Using the Basic Run Commands</td>
<td>7-4</td>
</tr>
<tr>
<td>7.3 Single-Stepping Through Code</td>
<td>7-7</td>
</tr>
<tr>
<td>7.4 Running Code Conditionally</td>
<td>7-10</td>
</tr>
<tr>
<td>7.5 Benchmarking</td>
<td>7-11</td>
</tr>
<tr>
<td>7.6 Halting Program Execution</td>
<td>7-12</td>
</tr>
<tr>
<td>7.7 Using Software Breakpoints</td>
<td>7-13</td>
</tr>
</tbody>
</table>
7.1 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 line in the File or Disassembly window that has a yellow arrow next to it. To do this, execute one of these commands:
  
  ```
  dasm PC
  or
  addr PC
  ```

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

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

  ![Restart Icon](image)

  Alternatively, you can select Restart from the Target menu.

- Set the PC to the current line in the File or Disassembly window. The current line is identified with a heavy band across the line:

  ![Current Line](image)

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

  1) Open the context menu for the window. (For more information, see page 2-7.)

  2) Select Set PC to Cursor from the context menu.
Modify the PC’s contents with one of these commands:

\[ ?\text{PC} = \text{new value} \]

or

\[ \text{eval } \text{pc} = \text{new value} \]

Modify the value of the PC in the CPU register. (For more information about changing values the displayed in the CPU window, see Section 8.3, *Basic Methods for Changing Data Values*, on page 8-4.)

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

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

- Run an entire program
- Run code up to a specific point in a program
- Run code in the current C function
- Run code through breakpoints

You can also use the debugger to reset the simulator.

Running an entire program

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

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

  ```plaintext
  run [expression]
  ```

  If you supply a logical or relational expression, the RUN command becomes a conditional run (see Section 7.4 on page 7-10).

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

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

- The debugger encounters a breakpoint. (For more information about how breakpoints affect a conditional run, see Section 7.4 on page 7-10.)
- You click the Halt icon on the toolbar:
- You select Halt! from the Target menu.
- You press $ESC$.  

Running code up to a specific point in a program

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

```
go [address]
```

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

- The debugger encounters a breakpoint.
- You click the Halt icon on the toolbar:

- You select Halt! from the Target menu.
- You press `ESC`.

You can also execute code from the current PC to the current line in the File or Disassembly window. The current line is identified with a heavy band across the line:

![Current PC and Current line](image)

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

- Open the context menu for the window. (For more information, see page 2-7.)
- Select Run to Cursor from the context menu.
Running the code in the current C function

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

- Click the Return icon on the toolbar:

- From the Target menu, select Return.

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

- Click the Halt icon on the toolbar:

- From the Target menu, select Halt!

- Press \texttt{ESC}.

Running code through breakpoints

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

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

To halt a continuous run, use one of the methods described in Section 7.6 on page 7-12.

Resetting the simulator

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

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

When you execute a software reset, the simulator simulates the ‘C6x processor and peripheral reset operation, putting the processor in a known state.
7.3 Single-Stepping Through Code

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

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

Note:
The debugger ignores interrupts when you single-step through assembly language code.

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

- If you do not supply an expression, the program executes a single statement, then halts.
- If you supply a logical or relational expression, this becomes a conditional single-step execution (see Section 7.4 on page 7-10).
- If you supply any other type of expression, the debugger treats the expression as a count parameter. The debugger single-steps count assembly language statements unless you are currently in C code. If you are currently in C code, the debugger single-steps count C statements.

**Single-stepping through assembly language or C code**

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

If you are in mixed mode, the debugger executes one assembly language statement at a time.
To use the basic single-step command, choose one of these methods:

- Click the Step icon on the toolbar:

- From the Target menu, select Step.

- Press \( F8 \).

- From the command line, enter the `STEP` command. The format for this command is:

```plaintext
step [expression]
```

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

For more information about the compiler’s \(-g\) option, see the *TMS320C6x Optimizing C Compiler User’s Guide*.

**Single-stepping through C code**

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

- Click the C Step icon on the toolbar:

- From the Target menu, select Step C.

- Press \( \text{CONTROL}F8 \).

- From the command line, enter the `CSTEP` command. The format for this command is:

```plaintext
cstep [expression]
```
**Single-stepping through code and stepping over breakpoints**

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

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

To halt a continuous step, use one of the methods described in Section 7.6 on page 7-12.

**Single-stepping through code and stepping over C functions**

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

- To use the next single-step command and single-step in terms of assembly language or C statements (similar to the basic single-step command), choose one of these methods:
  - Click the Next icon on the toolbar:
  - From the Target menu, select Next.
  - Press `F10`.
  - From the command line, enter the `NEXT` command. The format for this command is:
    ```
    next [expression]
    ```

- To use the next single-step command and single-step in terms of C statements (similar to the C single-step command), choose one of these methods:
  - Click the Next C icon on the toolbar:
  - From the Target menu, select Next C.
  - Press `CONTROL` + `F10`.
  - From the command line, enter the `CNEXT` command. The format for this command is:
    ```
    cnext [expression]
    ```
7.4 Running Code Conditionally

The RUN, STEP, CSTEP, NEXT, and CNEXT commands all have an optional expression parameter that can be a relational or logical expression. This type of expression uses one of the following operators as the highest precedence operator in the expression:

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

When you use this type of expression with these commands, the command becomes a conditional run. The debugger executes the command repeatedly for as long as the expression evaluates to true.

You must use software breakpoints with conditional runs; the expression is evaluated each time the debugger encounters a breakpoint. (Breakpoints are described in Section 7.7 on page 7-13.) For single-step commands, the expression is evaluated at each statement. Each time the debugger evaluates the conditional expression, it updates the screen.

Generally, you should set the breakpoints on statements that are related in some way to the expression. For example, if you are observing a particular variable in a Watch window, you may want to set breakpoints on statements that affect that variable and to use that variable in the expression.
7.5 Benchmarking

The debugger allows you to keep track of the number of CPU clock cycles consumed by a particular section of code. The debugger maintains the count in a pseudoregister named CLK. This process is referred to as benchmarking.

Benchmarking code is a multiple-step process:

**Step 1:** Set a software breakpoint at the statement that marks the beginning of the section of code that you want to benchmark. (For more information about setting software breakpoints, see Section 7.7 on page 7-13.)

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

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

**Step 4:** Enter the RUNB command:

```plaintext
runb
```

When the processor halts at the second breakpoint, the value of CLK is valid. To display it, use the `?` command or enter it into the Watch window with the Setup → Watch Variable menu option. This value is valid until you enter another run command.

**Notes:**

1) The RUNB command counts CPU clock cycles from the current PC to the breakpoint. This count is not cumulative. You cannot add the number of clock cycles between point A and point B to the number of cycles between point B and point C to learn the number of cycles between point A and point C. This situation occurs because of pipeline filling and flushing.

2) The value in CLK is valid only after using a RUNB command that is terminated by a software breakpoint.

3) When programming in C, avoid using a variable named CLK.
7.6 Halting Program Execution

Whenever you are running or single-stepping code, program execution halts automatically if the debugger encounters a breakpoint or if it reaches a particular point where you told it to stop (by supplying a count or an address with the RUN, GO, or any of the single-step commands). If you want to halt program execution explicitly, you can use one of these methods:

- Click the Halt icon on the toolbar:

- From the Target menu, select Halt!

- Press **ESC**.

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

During the debugging process, you may want to halt execution temporarily so that you can examine the contents of selected variables, registers, and memory locations before continuing with program execution. You can do this by setting software breakpoints at critical points in your code. You can set software breakpoints in assembly language code and in C code. A software breakpoint halts any program execution, whether you are running or single-stepping through code.

Software breakpoints are especially useful in combination with conditional execution (described in Section 7.4 on page 7-10).

When you set a software breakpoint, the debugger highlights the breakpointed line with this prefix: ●.

If you set a breakpoint in the disassembly, the debugger also highlights the associated C statement if the debugger has access to the C source. If you set a breakpoint in the C source, the debugger also highlights the associated statement in the disassembly. (If more than one assembly language statement is associated with a C statement, the debugger highlights the first of the associated assembly language statements.)

Notes:

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

2) You can set up to 200 breakpoints.

3) You cannot set multiple breakpoints at the same statement.
Setting a software breakpoint

To set a breakpoint, click next to the statement in the Disassembly or File window where you want the breakpoint to occur:

Click next to the statement where you want the breakpoint to occur

When you click next to a statement in the Disassembly or File window, a breakpoint symbol is shown:

A breakpoint is set on this statement

Another way to set a breakpoint is to use the context menu for the File or Disassembly window. You can set a breakpoint on the current line in the File or Disassembly window. The current line is identified with a heavy band across the line:

Current PC

Current line
To set a breakpoint on the current line in the File or Disassembly window, follow these steps:

- Open the context menu for the window. (For more information, see page 2-7.)

- Select Toggle Breakpoint from the context menu.

You can also set a breakpoint by using the Breakpoint Control dialog box. To open the Breakpoint Control dialog box, use one of these methods:

- Click the Breakpoints icon on the toolbar:

- From the Setup menu, select Breakpoints.

This displays the Breakpoint Control dialog box:

```
List of set breakpoints

To set a breakpoint, enter an absolute address, any C expression, the name of a C function, or the name of an assembly language label and click Add.

To set a breakpoint, follow these steps:

1) In the Address field of the Breakpoint Control dialog box, enter an absolute address, any C expression, the name of a C function, or the name of an assembly language label. If you want to specify a hex address, be sure to prefix the address number with 0x; otherwise, the debugger treats the number as a decimal address.

2) Click Add. The new breakpoint appears in the breakpoint list.

3) Click Close to close the Breakpoint Control dialog box.
```
Clearing a software breakpoint

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

To clear a breakpoint, click the breakpoint symbol (●) in the File or Disassembly window.

Another way to clear a breakpoint is to use the context menu for the File or Disassembly window:

1) Select the line in the File or Disassembly window where you want to remove the breakpoint.
2) From the context menu for the window, select Toggle Breakpoint.

You can also clear a breakpoint by using the Breakpoint Control dialog box (see the illustration on page 7-15):

1) Open the Breakpoint Control dialog box by using one of these methods:
   - Click the Breakpoints icon on the toolbar:
   - From the Setup menu, select Breakpoints.
2) Select the address of the breakpoint that you want to clear.
3) Click Delete. The breakpoint is removed from the breakpoint list.
4) Click Close to close the Breakpoint Control dialog box.

Clearing all software breakpoints

To clear all software breakpoints, follow these steps:

1) Open the Breakpoint Control dialog box by using one of these methods:
   - Click the Breakpoints icon on the toolbar:
   - From the Setup menu, select Breakpoints.
2) Click Delete All.
3) Click Close to close the Breakpoint Control dialog box.
**Saving breakpoint settings**

Software breakpoint settings are lost when you exit the debugger. However, you can save the list of breakpoints that you have set by following these steps:

1) Open the Breakpoint Control dialog box by using one of these methods:
   - Click the Breakpoints icon on the toolbar:
   - From the Setup menu, select Breakpoints.

2) Click Save List. This displays the Save Breakpoint File dialog box:

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

4) In the File name field, enter a name for the breakpoint file. You can use a .bpt extension to identify the file as a breakpoint file. Click Save.

5) In the Breakpoint Control dialog box, click Close.

**Notes:**

1) The breakpoint file is editable.
2) You can execute the breakpoint file with the TAKE command to automatically set up the breakpoints that defined in the file.
3) You can include the breakpoint file in your initialization batch file.
Using Software Breakpoints

**Loading saved breakpoint settings**

To load a list of saved breakpoints, follow these steps:

1) Open the Breakpoint Control dialog box by using one of these methods:
   - Click the Breakpoints icon on the toolbar:
   - From the Setup menu, select Breakpoints.

2) Click Load List. This displays the Load Breakpoint File dialog box:

   ![Load Breakpoint File dialog box]

   Select from a list of files

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

4) Click Open.

5) In the Breakpoint Control dialog box, click Close.

**Note:**

When you load a breakpoint file, breakpoints that you have defined previously in your debugging session are not cleared but remain in effect.
The debugger allows you to examine and modify many types of data related to the 'C6x and to your program. You can display and modify these values:

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

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Where Data Is Displayed</td>
<td>8-2</td>
</tr>
<tr>
<td>8.2 Basic Commands for Managing Data</td>
<td>8-2</td>
</tr>
<tr>
<td>8.3 Basic Methods for Changing Data Values</td>
<td>8-4</td>
</tr>
<tr>
<td>8.4 Managing Data in Memory</td>
<td>8-6</td>
</tr>
<tr>
<td>8.5 Managing Register Data</td>
<td>8-12</td>
</tr>
<tr>
<td>8.6 Managing Data in a Watch Window</td>
<td>8-17</td>
</tr>
<tr>
<td>8.7 Displaying Data in Alternative Formats</td>
<td>8-20</td>
</tr>
</tbody>
</table>
8.1 Where Data Is Displayed

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

<table>
<thead>
<tr>
<th>Type of Data</th>
<th>Window Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory locations</td>
<td>Memory window</td>
<td>Displays the contents of a range of memory</td>
</tr>
<tr>
<td>Register values</td>
<td>CPU window</td>
<td>Displays the contents of `C6x registers</td>
</tr>
<tr>
<td>Pointer data, variables, aggregate</td>
<td>Watch window</td>
<td>Displays selected data</td>
</tr>
<tr>
<td>types, and specific memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>locations or registers</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

8.2 Basic Commands for Managing Data

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

**Determining the type of a variable**

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, a structure tag, a typedef name, or an enumeration constant.

<table>
<thead>
<tr>
<th>Command</th>
<th>Result Displayed in the Command Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>whatis aai</td>
<td>int aai[10][5];</td>
</tr>
<tr>
<td>whatis xxx</td>
<td>struct xxx {</td>
</tr>
<tr>
<td></td>
<td>int a;</td>
</tr>
<tr>
<td></td>
<td>int b;</td>
</tr>
<tr>
<td></td>
<td>int c;</td>
</tr>
<tr>
<td></td>
<td>int f1 : 2;</td>
</tr>
<tr>
<td></td>
<td>int f2 : 4;</td>
</tr>
<tr>
<td></td>
<td>struct xxx *f3;</td>
</tr>
<tr>
<td></td>
<td>int f4[10];</td>
</tr>
</tbody>
</table>
Evaluating an expression

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

```
? expression
```

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

If the result of `expression` is scalar, the debugger displays the result as a decimal value in the Command window. If `expression` is a structure or array, the debugger displays the entire contents of the structure or array; you can halt long listings by pressing `ESC`.

Here are some examples that use the `?` command.

<table>
<thead>
<tr>
<th>Command</th>
<th>Result Displayed in the Command Window</th>
</tr>
</thead>
<tbody>
<tr>
<td>? aai</td>
<td><code>aai[0][0]</code> 1</td>
</tr>
<tr>
<td></td>
<td><code>aai[0][1]</code> 23</td>
</tr>
<tr>
<td></td>
<td><code>aai[0][2]</code> 45</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td>? j</td>
<td>4194425</td>
</tr>
<tr>
<td>? j=0x5a</td>
<td>90</td>
</tr>
</tbody>
</table>

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

```
eval expression
```

or

```
e expression
```

EVAL is useful for assigning values to registers or memory locations in a batch file, where it is not necessary to display the result.
8.3 Basic Methods for Changing Data Values

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

**Editing data displayed in a window**

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

- Registers displayed in the CPU window
- Memory contents displayed in a Memory window
- Values or elements displayed in a Watch window

To modify data in a data-display window, follow these steps:

1) Select the data item that you want to modify. Choose one of these methods:
   - Double-click the data item that you want to modify.
   - Select the data item that you want to modify and press F9.

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

3) When you finish typing the new information, press Enter or click on another data value. This replaces the original value with the new value.

**Editing data using expressions that have side effects**

Using the overwrite editing feature to modify data is straightforward. However, data-management methods take advantage of the fact that C expressions are accepted as parameters by most debugger commands and that C expressions can have side effects. When an expression has a side effect, the value of some variable in the expression changes as the result of evaluating the expression.

Side effects allow you to coerce many commands into changing values for you. Specifically, it is most helpful to use ? and EVAL to change data as well as display it.

For example, if you want to see what is in register 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
```
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:

\[
\begin{align*}
= & \quad += & \quad -= & \quad *= & \quad /= \\
\%=& & \&=& & ^=& \\
\gg=& & ++ & \quad & --
\end{align*}
\]
8.4 Managing Data in Memory

The main way to observe memory contents is to view the display in a Memory window. In mixed and assembly modes, the debugger displays the default Memory window automatically (labeled Memory). You can open any number of additional Memory windows to display different memory ranges. Figure 8–1 shows the default Memory window.

Figure 8–1. The Default Memory Window

The amount of memory that you can display in a Memory window is limited by the size of the window (which is limited only by your monitor’s screen size).

The debugger allows you to change the memory range displayed in the Memory window and to open additional Memory windows. The debugger also allows you to change the values at individual locations; for more information, see Section 8.3, Basic Methods for Changing Data Values, on page 8-4.

Changing the memory range displayed in a Memory window

To change the memory range displayed in a Memory window, enter a new starting address in the Address field of the Memory window, as shown in Figure 8–1. If you want to specify a hex address, be sure to prefix the address number with 0x; otherwise, the debugger treats the number as a decimal address.

You can also change the display of any data-display window—including the Memory window—by scrolling through the window’s contents. In the Memory window, the scroll bar handle is displayed in the middle of the scroll bar (see Figure 8–1). The middle of memory contents is defined as the most recent starting address that you entered in the Address field of the Memory window or with the MEM command (described on page 10-27). You can scroll up or down to see 1K bytes of memory on either side of the current starting address.
Opening an additional Memory window

To open an additional Memory window, use the MEM command. The syntax for this command is:

\[
\text{mem} \ \text{expression} [, [\text{display format}] [, \text{window name}]]
\]

- The \textit{expression} represents the address of the first entry in the Memory window. The end of the range is defined by the size of the window: to show more memory locations, make the window larger; to show fewer locations, make the window smaller.

  The \textit{expression} can be an absolute address, a symbolic address, or any C expression. Here are some examples:

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

\[
\text{mem} \ 0x0
\]

  Memory window addresses are shown in hexadecimal format. If you want to specify a hex address, be sure to prefix the address number with \textit{0x}; otherwise, the debugger treats the number as a decimal address.

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

\[
\text{mem} \ \&\text{SYM}
\]

  Prefix the symbol with the \& operator to use the address of the symbol.

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

\[
\text{mem} \ \text{SP} - \ A0 + \text{label}
\]

- The \textit{display format} parameter is optional. When used, the data is displayed in the selected format, as shown in Table 8-3 on page 8-20.

- Use the \textit{window name} parameter to name the additional Memory window. The debugger appends the \textit{window name} to the Memory window label. If you do not supply a name, the debugger does not open a new window; it simply updates the default Memory window to reflect the changes.
Displaying memory contents while you are debugging C

If you are debugging C code in auto mode, you do not see a Memory window—the debugger does not show the Memory window in the C-only display. However, there are several ways to display memory in this situation.

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

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

  ? *0x26

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

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

  wa *0x26

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

  disp *(float *)0x26
Saving memory values to a file

Sometimes it is useful to save a block of memory values to a file. You can use the Memory→Save menu option to do this; the files are saved in COFF format.

1) From the Memory menu, select Save. This displays the Save Memory to COFF File dialog box:

2) In the Address field, enter the first address in the block that you want to save. To specify a hex address, prefix the address number with 0x; otherwise, the debugger treats the number as a decimal address.

3) In the Length field, enter a length, in words, of the block. This parameter can be any C expression.

4) In the Filename field, enter a name for the saved block of memory. If you do not supply an extension, the debugger adds a .obj extension.

5) Click OK.
For example, to save the values in data memory locations 0x0000–0x0010 to a file named memsave.obj, you could enter:

![Save Memory to COFF File dialog box](image)

To reload memory values that were saved in a file, use the File→Load Program menu option.

**Filling a block of memory**

Sometimes it is useful to fill an entire block of memory at once with a particular value. You can fill a block of memory word by word by using the Memory→Fill Word command.

1) From the Memory menu, select Fill Word. This displays the Fill Memory dialog box:

![Fill memory dialog box](image)

2) In the Address field, enter the first address in the block that you want to fill. To specify a hex address, prefix the address number with 0x; otherwise, the debugger treats the number as a decimal address.

3) In the Length field, enter a length, in words, of the block.
4) In the Data field, enter a value that you want placed in each word in the block.

5) Click OK.

For example, to fill memory locations 0x10FF–0x1139 with the value 0xABCD, you could enter:

If you want to check whether memory has been filled correctly, you can change the Memory window display to show the block of memory beginning at memory address 0x10FF:

You can also use the debugger to fill a block of memory byte by byte by using the Memory→Fill Byte command.

1) From the Memory menu, select Fill Byte. This displays the Fill Memory—Byte dialog box.

2) In the Address field, enter the first address in the block that you want to fill. To specify a hex address, prefix the address number with 0x; otherwise, the debugger treats the number as a decimal address.

3) In the Length field, enter a length, in bytes, of the block.

4) In the Data field, enter a value that you want placed in each byte in the block.

5) Click OK.
8.5 Managing Register Data

In mixed and assembly modes, the debugger maintains a CPU window that displays the contents of individual registers.

![CPU Window Screenshot]

The debugger provides commands that allow you to display and modify the contents of specific registers. You can use the data-management commands or the debugger’s overwrite editing capability to modify the contents of any register displayed in the CPU or Watch window. For more information, see Section 8.3, Basic Methods for Changing Data Values, on page 8-4.

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 are 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 window.

You can also reorder the registers in the CPU window and display the ones that you are most interested in at the top of the CPU window. To do so, use the drag-and-drop method, as shown in Figure 8–2.
Figure 8–2. Reordering Registers in the CPU Window Using the Drag-and-Drop Method

In addition to the CPU window, you can observe the contents of selected registers by using the ? (evaluate expression) command or Setup → Watch Variable menu option:

- If you have only a temporary interest in the contents of a register, you can use the ? command to display the register’s contents. For example, if you want to know the contents of 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 Setup→Watch Variable to display the register in a Watch window. For example, if you want to observe the control status register (CSR), you could enter:

![Watch add window](image)

This adds the CSR to the Watch window in hexadecimal format and labels it as *Control Status Register*. The register's contents are continuously updated, just as if you were observing the register in the CPU window.

When you are debugging C in auto mode, the ? command and Setup→Watch Variable menu option are useful because the debugger does not show the CPU window in the C-only display.
Accessing single-precision floating-point registers

The debugger allows you to display the registers in the two general-purpose register files (A and B) in single-precision floating-point format. Table 8–1 lists the pseudoregister name for each general-purpose register.

Table 8–1. Pseudoregister Names for Single-Precision Floating-Point Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Pseudoregister Name</th>
<th>Register Name</th>
<th>Pseudoregister Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>FA0</td>
<td>B0</td>
<td>FB0</td>
</tr>
<tr>
<td>A1</td>
<td>FA1</td>
<td>B1</td>
<td>FB1</td>
</tr>
<tr>
<td>A2</td>
<td>FA2</td>
<td>B2</td>
<td>FB2</td>
</tr>
<tr>
<td>A3</td>
<td>FA3</td>
<td>B3</td>
<td>FB3</td>
</tr>
<tr>
<td>A4</td>
<td>FA4</td>
<td>B4</td>
<td>FB4</td>
</tr>
<tr>
<td>A5</td>
<td>FA5</td>
<td>B5</td>
<td>FB5</td>
</tr>
<tr>
<td>A6</td>
<td>FA6</td>
<td>B6</td>
<td>FB6</td>
</tr>
<tr>
<td>A7</td>
<td>FA7</td>
<td>B7</td>
<td>FB7</td>
</tr>
<tr>
<td>A8</td>
<td>FA8</td>
<td>B8</td>
<td>FB8</td>
</tr>
<tr>
<td>A9</td>
<td>FA9</td>
<td>B9</td>
<td>FB9</td>
</tr>
<tr>
<td>A10</td>
<td>FA10</td>
<td>B10</td>
<td>FB10</td>
</tr>
<tr>
<td>A11</td>
<td>FA11</td>
<td>B11</td>
<td>FB11</td>
</tr>
<tr>
<td>A12</td>
<td>FA12</td>
<td>B12</td>
<td>FB12</td>
</tr>
<tr>
<td>A13</td>
<td>FA13</td>
<td>B13</td>
<td>FB13</td>
</tr>
<tr>
<td>A14</td>
<td>FA14</td>
<td>B14</td>
<td>FB14</td>
</tr>
<tr>
<td>A15</td>
<td>FA15</td>
<td>B15</td>
<td>FB15</td>
</tr>
</tbody>
</table>

You can display the contents of these registers by using the ? (evaluate expression) command or Setup→Watch Variable menu selection.

For example, assume B15 = 0xE908. If you want to evaluate the FB15 pseudoregister, enter:

```
?fb15
```

The debugger shows the result in the display area of the Command window:

```
?fb15 8.2595861e–041
```

To modify the FA15 pseudoregister and set it equal to 15.75, enter:

```
?fa15 = 15.75
```

The debugger displays the following in the display area of the Command window:

```
?fa15 = 15.75 1.575000e+001
```
Managing Register Data

**Accessing double-precision floating-point registers**

The debugger allows you to access double-precision floating-point values in even/odd register pairs. There are 16 sets of general-purpose register pairs. Table 8–2 lists the pseudoregister name for each general-purpose register pair.

**Table 8–2. Pseudoregister Names for Double-Precision Floating-Point Registers**

<table>
<thead>
<tr>
<th>Register Pair</th>
<th>Pseudoregister Name</th>
<th>Register Pair</th>
<th>Pseudoregister Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1:A0</td>
<td>DA0</td>
<td>B1:B0</td>
<td>DB0</td>
</tr>
<tr>
<td>A3:A2</td>
<td>DA2</td>
<td>B3:B2</td>
<td>DB2</td>
</tr>
<tr>
<td>A5:A4</td>
<td>DA4</td>
<td>B5:B4</td>
<td>DB4</td>
</tr>
<tr>
<td>A7:A6</td>
<td>DA6</td>
<td>B7:B6</td>
<td>DB6</td>
</tr>
<tr>
<td>A9:A8</td>
<td>DA8</td>
<td>B9:B8</td>
<td>DB8</td>
</tr>
<tr>
<td>A11:A10</td>
<td>DA10</td>
<td>B11:B10</td>
<td>DB10</td>
</tr>
<tr>
<td>A13:A12</td>
<td>DA12</td>
<td>B13:B12</td>
<td>DB12</td>
</tr>
<tr>
<td>A15:A14</td>
<td>DA14</td>
<td>B15:B14</td>
<td>DB14</td>
</tr>
</tbody>
</table>

You can display the contents of these registers by using the ? (evaluate expression) command or Setup→Watch Variable menu option.

- For example, if you want to evaluate the A11:A10 register pair, enter the ? command with the DA10 pseudoregister name:
  
  ```
  ?da10
  ```

  The debugger shows the result in the display area of the Command window:

  ```
  ?da10 1.0933371e-309
  ```

- To modify the A11:A10 register pair (DA10) and set it equal to 25.75, enter:
  
  ```
  ?da10 = 25.75
  ```

  The debugger displays the following in the display area of the Command window:

  ```
  ?da10 = 25.75 2.575000e+001
  ```
8.6 Managing Data in a Watch Window

The debugger does not 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 would not be useful. Instead, the debugger allows you to open a Watch window that shows you how program execution affects specific expressions, variables, registers, or memory locations. You can choose which ones you want to observe. You can also use the Watch window to display members of complex, aggregate data types, such as arrays and structures.

The debugger displays a Watch window only when you specifically request a Watch window (described below).

Remember, you can use the data-management commands or the debugger’s overwrite editing capability to modify the contents of any value displayed in the Watch window. For more information, see Section 8.3, Basic Methods for Changing Data Values, on page 8-4.

Displaying data in a Watch window

To display a value in the Watch window, follow these steps:

1) From the Setup menu, select Watch Variable. This displays the Watch add dialog box:
2) In the Expression field, enter item that you want to watch. The expression can be any C expression, including an expression that has side effects. If you want to use the contents of an address as a parameter, be sure to prefix the address with the C indirection operator (*). For example, you could enter this value in the Expression field:

\[*0x26\]

3) If you want to assign a label for the watched item, use the Label field. If you leave the Label field blank, the debugger displays the expression as the label.

4) If you want to change the data format for the watched item, use the Format field. The data formats that you can use are shown in Table 8–3 on page 8-20. The format field is optional.

5) If you want to open a new Watch window, enter a name for the new Watch window in the Window name field. This field is optional. When you enter a window name, the debugger appends the window name to the Watch window label. If you do not supply a name, the debugger adds the item to the default Watch window.

6) Click OK.

After you open a Watch window, executing Setup→Watch Variable and using the same window name adds additional values to the Watch window. You can open as many Watch windows as you need by using unique window names.
Displaying additional data

When you use the Watch window to view structures, pointers, or arrays, you can display the additional data (the data pointed to or the members of the array or structure) by clicking the box icon next to watched item:

![Watch Window Example](image)

You can also display additional data by selecting an item and pressing `SPACE`.

Deleting watched values

To delete an entry from a Watch window, follow these steps:

1) Select the item in the Watch window that you want to delete.
2) Press `DELETE`.

If you want to close a Watch window and delete all of the items in that window in a single step, use the WR (watch reset) command. The syntax is:

```
wr [ (* | window name ) ]
```

The optional `window name` parameter deletes a particular Watch window; `*` deletes all Watch windows.

Note:

The debugger automatically closes any Watch windows when you execute File→Load Program, File→Load Symbols, the LOAD command, or the SLOAD command.
8.7 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, or Watch window can be displayed in a variety of formats.

Changing the default format for specific data types

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

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

The display format parameter identifies the new display format for any data of type data type. Table 8–3 lists the available formats and the corresponding characters that can be used as the display format parameter.

Table 8–3. Display Formats for Debugger Data

<table>
<thead>
<tr>
<th>Display Format</th>
<th>Parameter</th>
<th>Display Format</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default for the data type</td>
<td>*</td>
<td>Octal</td>
<td>o</td>
</tr>
<tr>
<td>ASCII character (bytes)</td>
<td>c</td>
<td>Valid address</td>
<td>p</td>
</tr>
<tr>
<td>Decimal</td>
<td>d</td>
<td>ASCII string</td>
<td>s</td>
</tr>
<tr>
<td>Exponential floating point</td>
<td>e</td>
<td>Unsigned decimal</td>
<td>u</td>
</tr>
<tr>
<td>Decimal floating point</td>
<td>f</td>
<td>Hexadecimal</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 8–4 lists the C data types that can be used for the data type parameter. Only a subset of the display formats applies to each data type, so Table 8–4 also shows valid combinations of data types and display formats.
Table 8–4. Data Types for Displaying Debugger Data

<table>
<thead>
<tr>
<th>Data Type</th>
<th>c</th>
<th>d</th>
<th>o</th>
<th>x</th>
<th>e</th>
<th>f</th>
<th>p</th>
<th>s</th>
<th>u</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>uchar</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>short</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>int</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>uint</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>long</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>ulong</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>ptr</td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>

Default Display Format: ASCII (c)

Decimal (d)

Exponential floating point (e)

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 should see a display that looks something like this:

  ![Display Formats](image)

- To reset all data types back to their default display formats, enter:
  ```
  setf *
  ```
Displaying Data in Alternative Formats

**Changing the default format with data-management commands**

You can also use the Setup→Watch Variable menu option and the ?, MEM, WA, and DISP commands to show data in alternative display formats. (The ? and DISP commands use alternative formats only for scalar types, arrays of scalar types, and individual members of aggregate types.)

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

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

Here are some examples:

- To watch the PC in octal, enter:

  ![Watch add dialog box](image)

  - Expression: pc
  - Format: o

- To display memory contents in octal, enter:

  ```
  mem 0x0, o
  ```

The valid combinations of data types and display formats listed for SETF also apply to the data displayed with ?, MEM, Setup→Watch Variable, WA, and DISP. For example, if you want to use display format e or f, the data that you are displaying must be of type float or type double. Additionally, you cannot use the s display format parameter with the MEM command.
The profiling environment is a special debugger environment that provides a method for collecting execution statistics about specific areas in your code. These statistics give you immediate feedback on your application’s performance.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Overview of the Profiling Environment</td>
<td>9-2</td>
</tr>
<tr>
<td>9.2 Overview of the Profiling Process</td>
<td>9-3</td>
</tr>
<tr>
<td>9.3 Entering the Profiling Environment</td>
<td>9-4</td>
</tr>
<tr>
<td>9.4 Defining Areas for Profiling</td>
<td>9-5</td>
</tr>
<tr>
<td>9.5 Defining the Stopping Point</td>
<td>9-15</td>
</tr>
<tr>
<td>9.6 Running a Profiling Session</td>
<td>9-17</td>
</tr>
<tr>
<td>9.7 Viewing Profile Data</td>
<td>9-20</td>
</tr>
<tr>
<td>9.8 Saving Profile Data to a File</td>
<td>9-27</td>
</tr>
</tbody>
</table>
# 9.1 Overview of the Profiling Environment

The profiling environment builds on the same intuitive interface available in the basic debugging environment and has these additional features:

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

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

- **Comprehensive display of statistics.** The profiler provides all the information you need for identifying bottlenecks in your code:
  - The number of times each area was entered during the profiling session.
  - The total execution time of an area, including or excluding the execution time of any subroutines called from within the area.
  - The maximum time for one iteration of an area, including or excluding the execution time of any subroutines called from within the area.

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

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

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

- **Disabled areas.** In addition to identifying areas that you can collect statistics on, you can also identify areas that you do not want to affect the statistics. This removes the timing impact from code such as a standard library function or a fully optimized portion of code.
9.2 An Overview of the Profiling Process

Profiling consists of five simple steps:

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Enter the profiling environment.</th>
<th>See Entering the Profiling Environment, page 9-4.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 2</td>
<td>Identify the areas of code where you want to collect statistics.</td>
<td>See Defining Areas for Profiling, page 9-5.</td>
</tr>
<tr>
<td>Step 3</td>
<td>Identify the profiling session stopping points.</td>
<td>See Defining a Stopping Point, page 9-15.</td>
</tr>
<tr>
<td>Step 5</td>
<td>View the profile data.</td>
<td>See Viewing Profile Data, page 9-20.</td>
</tr>
</tbody>
</table>

Note:

When you compile a program that will be profiled, you must use the –g and the –as shell 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

Here is a suggestion for a basic approach to profiling the performance of your program.

1) Mark all the functions in your program as profile areas.
2) Run a profiling session; find the busiest functions.
3) Unmark all the functions.
4) Mark the individual lines in the busy functions and run another profiling session.
9.3 Entering the Profiling Environment

To enter the profiling environment, select Profile Mode from the Profile menu.

Some restrictions apply to the profiling environment:

- The debugger will always be in mixed mode.
- Command, Disassembly, File, and Profile are the only windows available; additional windows, such as a Watch window, cannot be opened.
- The profiling environment supports only a subset of the debugger commands. Table 9–1 lists the debugger commands that can and cannot be used in the profiling environment.

<table>
<thead>
<tr>
<th>Can be used</th>
<th>Cannot be used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data-evaluation commands (such as ? and EVAL)</td>
<td>Most run commands (except the basic RUN command)</td>
</tr>
<tr>
<td>Breakpoint commands</td>
<td>Debugging mode commands (such as ASM, C, and MIX)</td>
</tr>
<tr>
<td>Memory-mapping commands</td>
<td>Commands related to the Watch, Memory, or Calls window</td>
</tr>
<tr>
<td>System commands (such as SYSTEM, TAKE, and ALIAS)</td>
<td></td>
</tr>
<tr>
<td>Windowing commands (such as SIZE, MOVE, and ZOOM)</td>
<td></td>
</tr>
</tbody>
</table>

Chapter 10, Summary of Commands, summarizes all of the debugger commands and tells you whether a command is valid in the profiling environment.
9.4 Defining Areas for Profiling

Within the profiling environment, you can collect statistics on three types of areas:

- Individual lines in C or disassembly
- Ranges in C or disassembly
- Functions in C only

To identify any of these areas for profiling, mark the line, range, or function. You can disable areas so that they do not affect the profile data, and you can reenable areas that have been disabled. You can also unmark areas that you are no longer interested in.

Using the mouse is the simplest way to mark, disable, enable, and unmark areas. A dialog box also supports these and more complex tasks.

The following subsections explain how to mark, disable, reenable, and unmark profile areas by using the mouse or the dialog box. For restrictions on profiling areas, see page 9-12.

Marking an area with a mouse

Marking an area qualifies it for profiling so that the debugger can collect timing statistics about the area.

Remember, to display C code, use the File→Open menu option or the FUNC command; to display disassembly, use the DASM command.

Notes:

1) Marking an area in C does not mark the associated code in disassembly.
2) Areas can be nested; for example, you can mark a line within a marked range. The debugger will report statistics for both the line and the function.
3) Ranges cannot overlap, and they cannot span function boundaries.
Defining Areas for Profiling

To mark an area with the mouse, follow these steps:

1) In the File or Disassembly window, click next to the line that you want to mark or next to the first line of the range that you want to mark:

When you click once next a line, a mouse icon appears, telling you that you need to click one more time:

2) Choose to mark a single line or a range:

☐ To mark a single line, click the mouse icon. This turns the mouse icon into a green right arrow:

☐ To mark a range, select the first line and then click the mouse icon on the last line.
To mark a range, click the last line of the range that you want to mark. This marks the range with two green right arrows that are connected:

You can also use the mouse to mark a function in C code. To do so, follow these steps:

1) In the File window, click next to the statement that declares the function that you want to mark.

2) When you see the mouse icon, click again to mark and enable the C function.

**Note:**

If you try to mark a line or function by double-clicking next to the statement that you want to mark, the debugger sets a software breakpoint instead of marking the line or function. To mark a line or function, click once. When you see the mouse icon, click again.
Defining Areas for Profiling

**Marking an area with a dialog box**

You can use a dialog box to mark areas for profiling. To do so, follow these steps:

1) Open the Profile Marking dialog box by using one of these methods:
   - From the Profile menu, select Select Areas.
   - From the context menu for the Profile window, select Select Areas.
   This displays the Profile Marking dialog box:

   - If you select Lines, enter an absolute address, C expression, assembly label, or line number.
   - If you select Ranges, enter a start and end value as absolute addresses, C expressions, assembly labels, or line numbers.

2) In the Level box, select C or Assembly.

3) In the Area box, select Lines, Ranges, or Functions. See Table 9–2 for a list of valid combinations.
4) Depending on what you select in step 3, do one or more of the following:
   - Next to Lines, enter an absolute address, C expression, assembly label, or line number.
   - Next to Ranges, enter a Start and an End value as absolute addresses, C expressions, assembly labels, or line numbers.
   - From the Module combo box, select a specific filename.
   - From the Function combo box, select a specific function name.

See Table 9–2 for a list of valid combinations.

5) Click Mark.

6) Click Close to close the dialog box.

Table 9–2. Using the Profile Marking Dialog Box to Mark Areas

(a) Marking lines

<table>
<thead>
<tr>
<th>To mark this area...</th>
<th>If C level is selected</th>
<th>If Assembly level is selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>By line number, address</td>
<td>Select a module name.</td>
<td>In the Area box, select Lines.</td>
</tr>
<tr>
<td></td>
<td>In the Area box, select Lines.</td>
<td>Next to Lines, specify an absolute address, a C expression, or an assembly label</td>
</tr>
<tr>
<td></td>
<td>Next to Lines, specify a line number.</td>
<td></td>
</tr>
<tr>
<td>All lines in a function</td>
<td>Select a function name.</td>
<td>In the Area box, select Lines.</td>
</tr>
<tr>
<td></td>
<td>In the Area box, select Lines.</td>
<td>Select a function name.</td>
</tr>
</tbody>
</table>

(b) Marking ranges

<table>
<thead>
<tr>
<th>To mark this area...</th>
<th>If C level is selected</th>
<th>If Assembly level is selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>By line numbers, addresses</td>
<td>Select a module name.</td>
<td>In the Area box, select Ranges.</td>
</tr>
<tr>
<td></td>
<td>In the Area box, select Ranges.</td>
<td>Next to Ranges, specify a Start and an End value. Use an absolute address, a C expression, or an assembly label for each.</td>
</tr>
<tr>
<td></td>
<td>Next to Ranges, specify a Start line number and an End line number.</td>
<td></td>
</tr>
</tbody>
</table>

(c) Marking functions

<table>
<thead>
<tr>
<th>To mark this area...</th>
<th>If C level is selected</th>
<th>If Assembly level is selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>By function name</td>
<td>Select a function name.</td>
<td>Not applicable</td>
</tr>
<tr>
<td></td>
<td>In the Area box, select Functions.</td>
<td></td>
</tr>
<tr>
<td>All functions in a module</td>
<td>Select a module name.</td>
<td>Not applicable</td>
</tr>
<tr>
<td></td>
<td>In the Area box, select Functions.</td>
<td></td>
</tr>
<tr>
<td>All functions everywhere</td>
<td>In the Area box, select Functions.</td>
<td>Not applicable</td>
</tr>
<tr>
<td></td>
<td>Be sure that Function and Module are set to N/A.</td>
<td></td>
</tr>
</tbody>
</table>
Disabling an area

At times, it is useful to identify areas that you do not want affecting profile statistics. To do this, disable the appropriate area. Disabling effectively subtracts the timing information of the disabled area from all profile areas that include or call the disabled area. Areas must be marked before they can be disabled.

For example, if you have marked a function that calls a standard C function such as malloc(), you may not want malloc() to affect the statistics for the calling function. You could mark the line that calls malloc(), and then disable the line. This way, the profile statistics for the function would not include the statistics for malloc().

Note:
If you disable an area after you have already collected statistics on it, that information will be lost.

The easiest way to disable an area is to click the green arrow(s) next to marked line, range, or function. When you do so, the arrow(s) becomes white:

You can also disable an area by using the Profile Marking dialog box:

1) Open the Profile Marking dialog box by using one of these methods:
   - From the Profile menu, select Select Areas.
   - From the context menu for the Profile window, select Select Areas.
   This displays the Profile Marking dialog box.

2) In the Level box, select C, Assembly, or Both.

3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 9–3 on page 9-13 for a list of valid combinations.
Defining Areas for Profiling

4) Depending on what you select in step 3, do one or more of the following:
   - Next to Lines, enter an absolute address, C expression, assembly label, or line number.
   - Next to Ranges, enter a Start and an End value as absolute addresses, C expressions, assembly labels, or line numbers.
   - From the Module combo box, select a specific filename.
   - From the Function combo box, select a specific function name.
   See Table 9–3 for a list of valid combinations.

5) Click Disable.

6) Click Close to close the dialog box.

Reenabling a disabled area

When an area has been disabled and you would like to profile it once again, you must enable the area. To reenable an area, click the white arrow(s) next to marked line, range, or function; the area will once again be highlighted with a green arrow.

You can also reenable an area by using the Profile Marking dialog box:

1) Open the Profile Marking dialog box by using one of these methods:
   - From the Profile menu, select Select Areas.
   - From the context menu for the Profile window, select Select Areas.
   This displays the Profile Marking dialog box.

2) In the Level box, select C, Assembly, or Both.

3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 9–3 for a list of valid combinations.

4) Depending on what you select in step 3, do one or more of the following:
   - Next to Lines, enter an absolute address, C expression, assembly label, or line number.
   - Next to Ranges, enter a Start and an End value as an absolute address, C expression, assembly label, or line number.
   - From the Module combo box, select a specific filename.
   - From the Function combo box, select a specific function name.
   See Table 9–3 for a list of valid combinations.

5) Click Enable.

6) Click Close to close the dialog box.
Defining Areas for Profiling

Unmarking an area

If you want to stop collecting information about a specific area, unmark it.

The easiest way to unmark an area is to double-click the green or white arrow(s) next to marked line, range, or function. This unmarks the line, range, or function.

You can also unmark an area by using the Profile Marking dialog box:

1) Open the Profile Marking dialog box by using one of these methods:
   - From the Profile menu, select Select Areas.
   - From the context menu for the Profile window, select Select Areas.

2) In the Level box, select C, Assembly, or Both.

3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 9–3 for a list of valid combinations.

4) Depending on what you select in step 3, do one or more of the following:
   - Next to Lines, enter an absolute address, C expression, assembly label, or line number.
   - Next to Ranges, enter a Start and an End value as absolute addresses, C expressions, assembly labels, or line numbers.
   - From the Module combo box, select a specific filename.
   - From the Function combo box, select a specific function name.
   See Table 9–3 for a list of valid combinations.

5) Click Unmark.

6) Click Close to close the dialog box.

Restrictions on profiling areas

The following restrictions apply to profiling areas:

- An area cannot begin in the delay slot of a branch instruction.
- An area can end in the last delay slot of a branch instruction but cannot end in any other delay slot of a branch instruction.
### Table 9–3. Disabling, Enabling, Unmarking, or Viewing Areas

(a) Disabling, enabling, unmarking, or viewing lines

<table>
<thead>
<tr>
<th>To identify this area...</th>
<th>If the C level is selected</th>
<th>If the Assembly level is selected</th>
<th>If the Both level is selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>By line number, address†</td>
<td>□ Select a module name.</td>
<td>□ In the Area box, select Lines.</td>
<td>□ Next to Lines, specify an absolute address, a C expression, or an assembly label.</td>
</tr>
<tr>
<td></td>
<td>□ In the Area box, select Lines.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>□ Next to Lines, specify a line number.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All lines in a function</td>
<td>□ Select a function name.</td>
<td>□ In the Area box, select Lines.</td>
<td>□ Select a function name.</td>
</tr>
<tr>
<td></td>
<td>□ In the Area box, select Lines.</td>
<td></td>
<td>□ In the Area box, select Lines.</td>
</tr>
<tr>
<td>All lines in a module</td>
<td>□ Select a module name.</td>
<td>□ In the Area box, select Lines.</td>
<td>□ Select a module name.</td>
</tr>
<tr>
<td></td>
<td>□ In the Area box, select Lines.</td>
<td></td>
<td>□ In the Area box, select Lines.</td>
</tr>
<tr>
<td>All lines everywhere</td>
<td>□ In the Area box, select Lines.</td>
<td>□ Be sure that Function and Module are set to N/A.</td>
<td>□ In the Area box, select Lines.</td>
</tr>
<tr>
<td></td>
<td>□ Be sure that Function and Module are set to N/A.</td>
<td></td>
<td>□ Be sure that Function and Module are set to N/A.</td>
</tr>
</tbody>
</table>

† You cannot specify line numbers or addresses when using the Profile View dialog box.

(b) Disabling, enabling, unmarking, or viewing ranges

<table>
<thead>
<tr>
<th>To identify this area...</th>
<th>If the C level is selected</th>
<th>If the Assembly level is selected</th>
<th>If the Both level is selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>By line numbers, addresses†</td>
<td>□ Select a module name.</td>
<td>□ In the Area box, select Ranges.</td>
<td>□ Next to Ranges, specify a Start and an End value as absolute addresses, C expressions, or assembly labels.</td>
</tr>
<tr>
<td></td>
<td>□ In the Area box, select Ranges.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>□ Next to Ranges, specify a Start line number and an End line number.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All ranges in a function</td>
<td>□ Select a function name.</td>
<td>□ In the Area box, select Ranges.</td>
<td>□ Select a function name.</td>
</tr>
<tr>
<td></td>
<td>□ In the Area box, select Ranges.</td>
<td></td>
<td>□ In the Area box, select Ranges.</td>
</tr>
<tr>
<td>All ranges in a module</td>
<td>□ Select a module name.</td>
<td>□ In the Area box, select Ranges.</td>
<td>□ Select a module name.</td>
</tr>
<tr>
<td></td>
<td>□ In the Area box, select Ranges.</td>
<td></td>
<td>□ In the Area box, select Ranges.</td>
</tr>
<tr>
<td>All ranges everywhere</td>
<td>□ In the Area box, select Ranges.</td>
<td>□ Be sure that Function and Module are set to N/A.</td>
<td>□ In the Area box, select Ranges.</td>
</tr>
<tr>
<td></td>
<td>□ Be sure that Function and Module are set to N/A.</td>
<td></td>
<td>□ Be sure that Function and Module are set to N/A.</td>
</tr>
</tbody>
</table>

† You cannot specify line numbers or addresses when using the Profile View dialog box.
Table 9–3. Disabling, Enabling, Unmarking, or Viewing Areas (Continued)

(c) Disabling, enabling, unmarking, or viewing functions

<table>
<thead>
<tr>
<th>To identify this area...</th>
<th>If the C level is selected</th>
<th>If the Assembly level is selected</th>
<th>If the Both level is selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>By function name</td>
<td>□ Select a function name.</td>
<td>□ Select a function name.</td>
<td>□ Select a function name.</td>
</tr>
<tr>
<td></td>
<td>□ In the Area box, select Functions.</td>
<td>□ In the Area box, select Functions.</td>
<td>□ In the Area box, select Functions.</td>
</tr>
<tr>
<td>All functions in a module</td>
<td>□ Select a module name.</td>
<td>□ Select a module name.</td>
<td>□ Select a module name.</td>
</tr>
<tr>
<td></td>
<td>□ In the Area box, select Functions.</td>
<td>□ In the Area box, select Functions.</td>
<td>□ In the Area box, select Functions.</td>
</tr>
<tr>
<td>All functions everywhere</td>
<td>□ In the Area box, select Functions.</td>
<td>□ In the Area box, select Functions.</td>
<td>□ In the Area box, select Functions.</td>
</tr>
<tr>
<td></td>
<td>□ Be sure that Function and Module are set to N/A.</td>
<td>□ Be sure that Function and Module are set to N/A.</td>
<td>□ Be sure that Function and Module are set to N/A.</td>
</tr>
</tbody>
</table>

(d) Disabling, enabling, unmarking, or viewing all areas

<table>
<thead>
<tr>
<th>To identify this area...</th>
<th>If the C level is selected</th>
<th>If the Assembly level is selected</th>
<th>If the Both level is selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>All areas in a function</td>
<td>□ Select a function name.</td>
<td>□ Select a function name.</td>
<td>□ Select a function name.</td>
</tr>
<tr>
<td></td>
<td>□ In the Area box, select All areas.</td>
<td>□ In the Area box, select All areas.</td>
<td>□ In the Area box, select All areas.</td>
</tr>
<tr>
<td>All areas in a module</td>
<td>□ Select a module name.</td>
<td>□ Select a module name.</td>
<td>□ Select a module name.</td>
</tr>
<tr>
<td></td>
<td>□ In the Area box, select All areas.</td>
<td>□ In the Area box, select All areas.</td>
<td>□ In the Area box, select All areas.</td>
</tr>
<tr>
<td>All areas everywhere</td>
<td>□ In the Area box, select All areas.</td>
<td>□ In the Area box, select All areas.</td>
<td>□ In the Area box, select All areas.</td>
</tr>
<tr>
<td></td>
<td>□ Be sure that Function and Module are set to N/A.</td>
<td>□ Be sure that Function and Module are set to N/A.</td>
<td>□ Be sure that Function and Module are set to N/A.</td>
</tr>
</tbody>
</table>
9.5 Defining a Stopping Point

Before you run a profiling session, you must identify the point where the debugger should stop collecting statistics. By default, C programs contain an exit label, and this is defined as the default stopping point when you load your program. (You can delete exit as a stopping point, if you choose.) If your program does not contain an exit label, or if you prefer to stop at a different point, you can use a software breakpoint to define another stopping point. You can set multiple breakpoints; the debugger stops at the first one it finds.

Even though no statistics can be gathered for areas following a breakpoint, the areas will be listed in the Profile window.

---

**Note:**

You cannot set a software breakpoint on a statement that has already been defined as a part of a profile area.

Setting and clearing a software breakpoint in the profiling environment is similar to setting and clearing a software breakpoint in the basic debugging environment. For more information about setting and clearing software breakpoints, see Section 7.7 on page 7-13.

### Setting a software breakpoint

To set a breakpoint, **double-click** next to the statement in the Disassembly or File window where you want the breakpoint to occur.

You can also set a breakpoint using the Breakpoint Control dialog box:

1) Open the Breakpoint Control dialog box by using one of these methods:
   - Click the Breakpoints icon on the toolbar:
     ![Breakpoints icon](image)
   - From the Setup menu, select Breakpoints.

2) In the Address field of the Breakpoint Control dialog box, enter an absolute address, any C expression, the name of a C function, or an assembly language label.

3) Click Add. The new breakpoint appears in the breakpoint list.

4) Click Close to close the Breakpoint Control dialog box.
Clearing a software breakpoint

To clear a breakpoint, *double-click* the breakpoint symbol (●) in the File or Disassembly window.

You can also clear a breakpoint by using the Breakpoint Control dialog box:

1) Open the Breakpoint Control dialog box by using one of these methods:
   - Click the Breakpoints icon on the toolbar:
   - From the Setup menu, select Breakpoints.

2) Select the address of the breakpoint that you want to clear.

3) Click Delete. The breakpoint is removed from the breakpoint list.

4) Click Close to close the Breakpoint Control dialog box.
9.6 Running a Profiling Session

Once you have defined profile areas and a stopping point, you can run a profiling session. You can run two types of profiling sessions:

- A full profile collects a full set of statistics for the defined profile areas.
- A quick profile collects a subset of the available statistics (it does not collect exclusive or exclusive max data, which are described in Section 9.7 on page 9-20). This reduces overhead because the debugger does not have to track entering/exiting subroutines within an area.

**Running a full or quick profiling session**

To run a profiling session, follow these steps:

1) Open the Profile Run dialog box by using one of these methods:
   - Click the Run icon on the toolbar:
   - From the Profile menu, select Run.
   - Press F5.

This displays the Profile Run dialog box:

```
Profile Run

Run Method:
- Full, all fields
- Quick, no exclusive fields
- Resume, Clear data

Display Ratio:
<table>
<thead>
<tr>
<th>Often</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Start Point:

OK    Cancel    Help
```

2) In the Run Method box, select the type of profiling session that you want to perform: Full or Quick.
Running a Profiling Session

3) Slide the Display Rate scale to specify how often the display is updated. You can choose a Display Rate from Often to Never. A Display Rate of Never causes the profiler to display profiling information only when the profiling session is complete.

4) In the Start Point field, enter the starting point for the profiling session. The starting point can be a label, a function name, or a memory address. You can choose from a list of starting points by clicking on the arrow at the end of the Start Point field.

5) Click OK.

After you click OK, your program restarts and runs to the defined starting point. You can tell that the debugger is profiling because the status bar changes to Target: Profiling:

For Help, press F1

Profiling begins when the starting point is reached and continues until a stopping point is reached or until you halt the profiling session by doing one of the following:

- Click the Halt icon on the toolbar:

- From the Target menu, select Halt!

- Press (ESC) .
Resuming a profiling session that has halted

To resume a profiling session that has halted, follow these steps:

1) Open the Profile Run dialog box by using one of these methods:
   - Click the Run icon on the toolbar:
   - From the Profile menu, select Run.
   - Press F5.

   This displays the Profile Run dialog box:

   ![Profile Run dialog box]

   - To resume a profiling session that has halted, select Resume.
   - To clear out previously collected profile data, select Clear data.
   - Slide the scale bar to specify how often the display is updated.

2) In the Run Method box, select Resume.

3) If you want to clear out the previously collected data, select Clear data in the Run Method box.

4) Slide the Display Rate scale to specify how often the display is updated. You can choose a Display Rate from Often to Never. A Display Rate of Never causes the profiler to display profiling information only when the profiling session is complete.

5) In the Start Point field, enter the starting point for the profiling session. The starting point can be a label, a function name, or a memory address. You can choose from a list of starting points by clicking on the arrow at the end of the Start Point field.

6) Click OK.
9.7 Viewing Profile Data

The statistics collected during a profiling session are displayed in the Profile window. Figure 9–1 shows an example of this window.

Figure 9–1. An Example of the Profile Window

The example in Figure 9–1 shows the Profile window with some default conditions:

- Column headings show the labels for the default set of profile data, including Count, Inclusive, Incl-Max, Exclusive, and Excl-Max.
- The data is sorted on the address of the first line in each area.
- All marked areas are listed, including disabled areas.

You can modify the Profile window to display selected profile areas or different data; you can also sort the data differently. The following subsections explain how to do these things.
**Viewing different profile data**

By default, the Profile window shows a set of statistics labeled as Count, Inclusive, Incl-Max, Exclusive, and Excl-Max. The Address field, which is not part of the default statistics, can also be displayed. Table 9–4 describes the statistic that each field represents.

**Table 9–4. Types of Data Shown in the Profile Window**

<table>
<thead>
<tr>
<th>Label</th>
<th>Profile Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>The number of times a profile area is entered during a session</td>
</tr>
<tr>
<td>Inclusive</td>
<td>The total execution time (cycle count) of a profile area, including the execution time of any subroutines called from within the profile area</td>
</tr>
<tr>
<td>Incl-Max (inclusive maximum)</td>
<td>The maximum inclusive time for one iteration of a profile area</td>
</tr>
<tr>
<td>Exclusive</td>
<td>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.</td>
</tr>
<tr>
<td>Excl-Max (exclusive maximum)</td>
<td>The maximum exclusive time for one iteration of a profile area</td>
</tr>
<tr>
<td>Address</td>
<td>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.</td>
</tr>
</tbody>
</table>

In addition to viewing this data in the default manner, you can view each of these statistics individually. The benefit of viewing them individually is that in addition to a cycle count, you are also supplied with a percentage indication and a histogram.

To view the fields individually, click the Profile window.
When you click the Profile window, fields are displayed individually in the order shown in Figure 9–2.

Figure 9–2. Cycling Through the Profile Window Fields

Count
Inclusive
Incl-Max
Exclusive
Excl-Max
Address
Default

Note: Exclusive and Excl-Max are shown only when you run a full profile.

One advantage of using the mouse is that you can change the display while you are profiling.

You can also use the Profile View dialog box to select the field you want to display. To do so, follow these steps:

1) Open the Profile View dialog box by using one of these methods:
   - From the Profile menu, select Change View.
   - From the context menu for the Profile window, select Change View.

   This displays the Profile View dialog box.

2) In the Display Field box, select the data field that you want to display:

3) Click OK.
Sorting profile data

By default, the data displayed in the Profile window is sorted according to the memory addresses of the displayed areas. The area with the least significant address is listed first, followed by the area with the next least significant address, etc. When you view fields individually, the data is automatically sorted from highest cycle count to lowest (instead of by address).

To sort the data on any of the data fields, follow these steps:

1) Open the Profile View dialog box by using one of these methods:
   - From the Profile menu, select Change View.
   - From the context menu for the Profile window, select Change View.

   This displays the Profile View dialog box.

2) In the Sort Field box, select the data field that you want to sort on:

   ![Sort Field Box]

3) Click OK.

   For example, to sort all of the data on the basis of the values of the Inclusive field, select Inclusive in the Sort Field box. The area with the highest Inclusive field displays first, and the area with the lowest Inclusive field displays last. This applies even when you are viewing individual fields.
Viewing different profile areas

By default, all marked areas are listed in the Profile window. You can modify the window to display selected areas. To do this, follow these steps:

1) Open the Profile View dialog box by using one of these methods:
   - From the Profile menu, select Change View.
   - From the context menu for the Profile window, select Change View.

   This displays the Profile View dialog box.

2) In the Level box, select C, Assembly, or Both.

3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 9–3 on page 9-13 on for a list of valid combinations.

4) If you want to view areas in a specific file or function, do one of the following:
   - From the Module combo box, select a specific filename.
   - From the Function combo box, select a specific function name.

   See Table 9–3 on page 9-13 on for a list of valid combinations.

5) Click OK.
If you want to reset the Profile window to its default characteristics, use the Profile View dialog box (Profile→Change View). Click the Defaults button, then click OK.

Interpreting session data

General information about a profiling session is displayed in the Command window during and after the session. This information identifies the starting and stopping points. It also lists statistics for three important areas:

- **Run cycles** shows the number of execution cycles consumed by the program from the starting point to the stopping point.
- **Profile cycles** equals the run cycles minus the cycles consumed by disabled areas.
- **Hits** shows the number of internal breakpoints encountered during the profiling session.

Viewing code associated with a profile area

You can view the code associated with a displayed profile area. The debugger updates the display so that the associated C or disassembly statements are shown in the File or Disassembly window.

To select the profile area in the Profile window and display the associated code, double-click the area that you want to display:

If the area is a function name, the debugger opens a File window and displays that function:
If the area is in disassembly code, the debugger displays that code in the Disassembly window.

To view the code associated with another area, double-click another area.

If you are attempting to show disassembly, you might need to make several attempts because you can access program memory only when the target is not running.
9.8 Saving Profile Data to a File

You may want to run several profiling sessions during a debugging session. Whenever you start a new profiling session, the results of the previous session are lost. However, you can save the results of the current profiling session to a system file.

The saved file contents are in ASCII and are formatted in exactly the same manner as they are displayed (or would be displayed) in the Profile window. The general profiling-session information that is displayed in the Command window is also written to the file.

**Saving the contents of the Profile window**

To save the contents of the Profile window to a system file, follow these steps:

1) From the Profile menu, select Save View. This displays the Save Profile View File dialog box:

![Save Profile View File dialog box]

   - **Save in:** source
   - **File name:** Enter a name for the file. Use a .prf extension.
   - **Save as type:** Profile Data Files (*.prf)

2) In the File name field, enter a name for the file. You can use a .prf extension to identify the file as a profile data file.

3) Click Save.

This saves only the current view; if, for example, you are viewing only the Count field, then only that information will be saved. If the file already exists, debugger overwrites the file with the new data.
Saving Profile Data to a File

Saving all data for currently displayed areas

To save all data for the currently displayed areas, follow these steps:

1) From the Profile menu, select Save All. This displays the Save Profile File dialog box.

2) In the File name field, enter a name for the file. You can use a .prf extension to identify the file as a profile data file.

3) Click Save.

This saves all views of the data—including the individual count, inclusive, etc.—with the percentage indications and histograms. If the file already exists, debugger overwrites the file with the new data.
Summary of Commands

This chapter describes the basic debugger commands and profiling commands.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 Functional Summary of Debugger Commands</td>
<td>10-2</td>
</tr>
<tr>
<td>10.2 Alphabetical Summary of Debugger Commands</td>
<td>10-9</td>
</tr>
<tr>
<td>10.3 Summary of Profiling Commands</td>
<td>10-49</td>
</tr>
</tbody>
</table>
10.1 Functional Summary of Debugger Commands

This section summarizes the debugger commands according to these categories:

- **Changing modes.** These commands (listed on page 10-3) allow you to switch freely between the debugging modes (auto, mixed, and assembly).

- **Managing windows.** These commands (listed on page 10-3) allow you to make a window active and move or resize the active window.

- **Displaying and changing data.** These commands (listed on page 10-3) allow you to display and evaluate a variety of data items.

- **Performing system tasks.** These commands (listed on page 10-4) allow you to perform several system functions and provide you with some control over the target system.

- **Managing breakpoints.** These commands (listed on page 10-5) provide you with a command line method for controlling software breakpoints.

- **Displaying files and loading programs.** These commands (listed on page 10-5) allow you to change the displays in the File and Disassembly windows and to load object files into memory.

- **Customizing the screen.** These commands (listed on page 10-6) allow you to customize the debugger display, then save and later reuse the customized displays.

- **Memory mapping.** These commands (listed on page 10-6) allow you to define the areas of target memory that the debugger can access.

- **Running programs.** These commands (listed on page 10-7) provide you with a variety of methods for running your programs in the debugger environment.

- **Profiling commands.** These commands (listed on page 10-8) allow you to collect execution statistics for your code.
## Changing modes

<table>
<thead>
<tr>
<th>To put the debugger in...</th>
<th>Use this command...</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly mode</td>
<td>asm</td>
<td>10-11</td>
</tr>
<tr>
<td>Auto mode for debugging C code</td>
<td>c</td>
<td>10-13</td>
</tr>
<tr>
<td>Mixed mode</td>
<td>mix</td>
<td>10-28</td>
</tr>
</tbody>
</table>

## Managing windows

<table>
<thead>
<tr>
<th>To do this...</th>
<th>Use this command...</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reposition a window</td>
<td>move</td>
<td>10-28</td>
</tr>
<tr>
<td>Resize a window</td>
<td>size</td>
<td>10-38</td>
</tr>
<tr>
<td>Make a window active</td>
<td>win</td>
<td>10-47</td>
</tr>
<tr>
<td>Make a window as large as possible</td>
<td>zoom</td>
<td>10-48</td>
</tr>
</tbody>
</table>

## Displaying and changing data

<table>
<thead>
<tr>
<th>To do this...</th>
<th>Use this command...</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluate and display the result of a C expression</td>
<td>?</td>
<td>10-9</td>
</tr>
<tr>
<td>Display the values in an array or structure, or display the value that a pointer is pointing to</td>
<td>disp</td>
<td>10-17</td>
</tr>
<tr>
<td>Evaluate a C expression without displaying the results</td>
<td>eval</td>
<td>10-19</td>
</tr>
<tr>
<td>Change the range of memory displayed in the Memory window or display an additional Memory window</td>
<td>mem</td>
<td>10-27</td>
</tr>
<tr>
<td>Change the format for displaying data values</td>
<td>setf</td>
<td>10-37</td>
</tr>
<tr>
<td>Continuously display the value of a variable, register, or memory location within the Watch window</td>
<td>wa</td>
<td>10-46</td>
</tr>
<tr>
<td>Delete a data item from the Watch window</td>
<td>wd</td>
<td>10-47</td>
</tr>
<tr>
<td>Show the type of a data item</td>
<td>whatis</td>
<td>10-47</td>
</tr>
<tr>
<td>Delete all data items from the Watch window</td>
<td>wr</td>
<td>10-48</td>
</tr>
</tbody>
</table>
### Performing system tasks

<table>
<thead>
<tr>
<th>To do this...</th>
<th>Use this command...</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define your own command string</td>
<td>alias</td>
<td>10-11</td>
</tr>
<tr>
<td>Change the current working directory from within the debugger environment</td>
<td>cd, chdir</td>
<td>10-14</td>
</tr>
<tr>
<td>Clear all displayed information from the display area of the Command window</td>
<td>cls</td>
<td>10-14</td>
</tr>
<tr>
<td>List the contents of the current directory or any other directory</td>
<td>dir</td>
<td>10-16</td>
</tr>
<tr>
<td>Record the information shown in the display area of the Command window</td>
<td>dlog</td>
<td>10-18</td>
</tr>
<tr>
<td>Display a string to the Command window while executing a batch file</td>
<td>echo</td>
<td>10-18</td>
</tr>
<tr>
<td>Display a help topic for a debugger command</td>
<td>help</td>
<td>10-21</td>
</tr>
<tr>
<td>Conditionally execute debugger commands in a batch file</td>
<td>if/else/endif</td>
<td>10-22</td>
</tr>
<tr>
<td>Loop debugger commands in a batch file</td>
<td>loop/endloop</td>
<td>10-23</td>
</tr>
<tr>
<td>Pause the execution of a batch file</td>
<td>pause</td>
<td>10-30</td>
</tr>
<tr>
<td>Exit the debugger</td>
<td>quit</td>
<td>10-33</td>
</tr>
<tr>
<td>Reset the target system</td>
<td>reset</td>
<td>10-33</td>
</tr>
<tr>
<td>Associate a beeping sound with the display of error messages</td>
<td>sound</td>
<td>10-39</td>
</tr>
<tr>
<td>Enter any operating-system command or exit to a system shell</td>
<td>system</td>
<td>10-42</td>
</tr>
<tr>
<td>Execute commands from a batch file</td>
<td>take</td>
<td>10-43</td>
</tr>
<tr>
<td>Delete an alias definition</td>
<td>unalias</td>
<td>10-43</td>
</tr>
<tr>
<td>Name additional directories that can be searched when you load source files</td>
<td>use</td>
<td>10-44</td>
</tr>
</tbody>
</table>
Functional Summary of Debugger Commands

Managing breakpoints

<table>
<thead>
<tr>
<th>To do this...</th>
<th>Use this command...</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a software breakpoint</td>
<td>ba</td>
<td>10-12</td>
</tr>
<tr>
<td>Delete a software breakpoint</td>
<td>bd</td>
<td>10-12</td>
</tr>
<tr>
<td>Display a list of all the software breakpoints that are set</td>
<td>bl</td>
<td>10-12</td>
</tr>
<tr>
<td>Reset (delete) all software breakpoints</td>
<td>br</td>
<td>10-13</td>
</tr>
</tbody>
</table>

Displaying files and loading programs

<table>
<thead>
<tr>
<th>To do this...</th>
<th>Use this command...</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display C and/or assembly language code at a specific point</td>
<td>addr</td>
<td>10-10</td>
</tr>
<tr>
<td>Display the Calls window</td>
<td>calls</td>
<td>10-13</td>
</tr>
<tr>
<td>Display assembly language code at a specific address</td>
<td>dasm</td>
<td>10-16</td>
</tr>
<tr>
<td>Display a text file in a File window</td>
<td>file</td>
<td>10-19</td>
</tr>
<tr>
<td>Display a specific C function</td>
<td>func</td>
<td>10-20</td>
</tr>
<tr>
<td>Load an object file and its symbol table</td>
<td>load</td>
<td>10-22</td>
</tr>
<tr>
<td>Load only the object-code portion of an object file</td>
<td>reload</td>
<td>10-33</td>
</tr>
<tr>
<td>Load only the symbol-table portion of an object file</td>
<td>sload</td>
<td>10-38</td>
</tr>
</tbody>
</table>
### Customizing the screen

<table>
<thead>
<tr>
<th>To do this...</th>
<th>Use this command...</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change the command-line prompt</td>
<td>prompt</td>
<td>10-32</td>
</tr>
<tr>
<td>Load and use a previously saved custom screen configuration</td>
<td>sconfig</td>
<td>10-36</td>
</tr>
<tr>
<td>Save a custom screen configuration</td>
<td>ssave</td>
<td>10-40</td>
</tr>
</tbody>
</table>

### Memory mapping

<table>
<thead>
<tr>
<th>To do this...</th>
<th>Use this command...</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize a block of memory word by word</td>
<td>fill</td>
<td>10-20</td>
</tr>
<tr>
<td>Initialize a block of memory byte by byte</td>
<td>fillb</td>
<td>10-20</td>
</tr>
<tr>
<td>Add an address range to the memory map</td>
<td>ma</td>
<td>10-24</td>
</tr>
<tr>
<td>Enable or disable memory mapping</td>
<td>map</td>
<td>10-25</td>
</tr>
<tr>
<td>Connect a simulated I/O port to an input or output file</td>
<td>mc</td>
<td>10-25</td>
</tr>
<tr>
<td>Delete an address range from the memory map</td>
<td>md</td>
<td>10-26</td>
</tr>
<tr>
<td>Disconnect a simulated I/O port</td>
<td>mi</td>
<td>10-27</td>
</tr>
<tr>
<td>Display a list of the current memory map settings</td>
<td>ml</td>
<td>10-28</td>
</tr>
<tr>
<td>Reset the memory map (delete all range definitions)</td>
<td>mr</td>
<td>10-29</td>
</tr>
<tr>
<td>Save a block of memory to a system file</td>
<td>ms</td>
<td>10-29</td>
</tr>
</tbody>
</table>
## Running programs

<table>
<thead>
<tr>
<th>To do this..</th>
<th>Use this command...</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-step through assembly language or C code, one C statement at a time; step over function calls</td>
<td>cnext</td>
<td>10-15</td>
</tr>
<tr>
<td>Single-step through assembly language or C code, one C statement at a time</td>
<td>cstep</td>
<td>10-15</td>
</tr>
<tr>
<td>Run a program up to a certain point</td>
<td>go</td>
<td>10-21</td>
</tr>
<tr>
<td>Halt the target system</td>
<td>halt</td>
<td>10-21</td>
</tr>
<tr>
<td>Single-step through assembly language or C code; step over function calls</td>
<td>next</td>
<td>10-30</td>
</tr>
<tr>
<td>Reset the target system</td>
<td>reset</td>
<td>10-33</td>
</tr>
<tr>
<td>Reset the program entry point</td>
<td>restart</td>
<td>10-34</td>
</tr>
<tr>
<td>Execute code in a function and return to the function's caller</td>
<td>return</td>
<td>10-34</td>
</tr>
<tr>
<td>Run a program</td>
<td>run</td>
<td>10-35</td>
</tr>
<tr>
<td>Run a program with benchmarking—count the number of CPU clock cycles consumed by the executing portion of code</td>
<td>runb</td>
<td>10-35</td>
</tr>
<tr>
<td>Single-step through assembly language or C code</td>
<td>step</td>
<td>10-41</td>
</tr>
<tr>
<td>Execute commands from a batch file</td>
<td>take</td>
<td>10-43</td>
</tr>
</tbody>
</table>
Profiling commands

All of the profiling commands can be entered from the Profile menu and associated dialog boxes. In many cases, using the Profile menu and dialog boxes 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; these commands are also described in the alphabetical command summary. The remaining profiling commands are summarized in Section 10.3, Summary of Profiling Commands, on page 10-49.

<table>
<thead>
<tr>
<th>To do this...</th>
<th>Use this command...</th>
<th>See page...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run a full profiling session</td>
<td>pf</td>
<td>10-31</td>
</tr>
<tr>
<td>Run a quick profiling session</td>
<td>pq</td>
<td>10-31</td>
</tr>
<tr>
<td>Resume a profiling session</td>
<td>pr</td>
<td>10-32</td>
</tr>
<tr>
<td>Add a stopping point</td>
<td>sa</td>
<td>10-36</td>
</tr>
<tr>
<td>Delete a stopping point</td>
<td>sd</td>
<td>10-36</td>
</tr>
<tr>
<td>List all the stopping points</td>
<td>sl</td>
<td>10-38</td>
</tr>
<tr>
<td>Delete all the stopping points</td>
<td>sr</td>
<td>10-39</td>
</tr>
<tr>
<td>Save all the profile data to a file</td>
<td>vaa</td>
<td>10-44</td>
</tr>
<tr>
<td>Save currently displayed profile data to a file</td>
<td>vac</td>
<td>10-44</td>
</tr>
<tr>
<td>Reset the display in the Profile window to show all areas and the default set of data</td>
<td>vr</td>
<td>10-45</td>
</tr>
</tbody>
</table>
10.2 Alphabetical Summary of Debugger Commands

There are two debugger environments: the basic debugger environment and the profiling environment. Some commands can be used in both environments; other commands can be used in only one of the environments. Each command description identifies the applicable environments for the command.

Commands are not case sensitive; to emphasize this, command names are shown in both uppercase and lowercase throughout this book.

Evaluate Expression

Syntax

? expression [, display format]

Menu selection

none

Toolbar 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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Result is displayed in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Default for the data type</td>
</tr>
<tr>
<td>c</td>
<td>ASCII character (bytes)</td>
</tr>
<tr>
<td>d</td>
<td>Decimal</td>
</tr>
<tr>
<td>e</td>
<td>Exponential floating point</td>
</tr>
<tr>
<td>f</td>
<td>Decimal floating point</td>
</tr>
<tr>
<td>o</td>
<td>Octal</td>
</tr>
<tr>
<td>p</td>
<td>Valid address</td>
</tr>
<tr>
<td>s</td>
<td>ASCII string</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal</td>
</tr>
<tr>
<td>x</td>
<td>Hexadecimal</td>
</tr>
</tbody>
</table>

Summary of Commands
**Display Code at Specified Address**

**Syntax**

`addr { address | function name }`

**Menu selection**

none

**Toolbar 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` as the first line of code in the File window.

- In mixed mode, ADDR affects both the Disassembly and File windows by displaying code starting at `address` or at `function name` as the first line of code in the Disassembly and File window.

**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**

Setup→Alias Commands

**Toolbar 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 as you separate them with semicolons and enclose the entire string of commands in quotation marks. Also, you can identify command parameters by a percent sign followed by a number (%1, %2, etc.). The total number of characters for an individual command (expanded to include parameter values) is limited to 132.

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

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

### asm

**Enter Assembly Mode**

**Syntax**

`asm`

**Menu selection**

Mode→Assembly

**Toolbar selection**

none

**Environments**

- basic debugger
- profiling

**Description**

The ASM command changes from the current debugging mode to assembly mode. If you are already in assembly mode, the ASM command has no effect.
**ba, bd, bl  Alphabetical Summary of Debugger Commands**

---

### ba

**Add Software Breakpoint**

**Syntax**

\[ ba \ address \]

**Menu selection**

Setup → Breakpoints

**Toolbar selection**

[ ]

**Environments**

☑ basic debugger ✔ profiling

**Description**

The BA command sets a software breakpoint at a specific address. The address can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label.

You can set breakpoints in program memory (RAM) only; the address parameter is treated as a program-memory address.

---

### bd

**Delete Software Breakpoint**

**Syntax**

\[ bd \ address \]

**Menu selection**

Setup → Breakpoints

**Toolbar selection**

[ ]

**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**

Setup → Breakpoints

**Toolbar selection**

[ ]

**Environments**

☑ basic debugger ✔ profiling

**Description**

The BL command lists all the software breakpoints that are currently set in your program. It displays a table of breakpoints in the display area of the Command window. BL lists all the breakpoints that are set in the order in which you set them.
### br
**Reset Software Breakpoint**

**Syntax**
`br`

**Menu selection**
Setup→Breakpoints

**Toolbar selection**

**Environments**
✓ basic debugger
✓ profiling

**Description**
The BR command clears all software breakpoints that are set.

### c
**Enter Auto Mode**

**Syntax**
c

**Menu selection**
Mode→C (Auto)

**Toolbar selection**
none

**Environments**
✓ basic debugger

**Description**
The C command changes from the current debugging mode to auto mode. If you are already in auto mode, the C command has no effect.

### calls
**Open Calls Window**

**Syntax**
calls

**Menu selection**
View→Call Stack Window

**Toolbar selection**
none

**Environments**
✓ basic debugger

**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.
Alphabetical Summary of Debugger Commands

**cd, chdir, cls**

**cd, chdir**  
*Change Directory*

**Syntax**

- `cd [directory name]`
- `chdir [directory name]`

**Menu selection** none

**Toolbar selection** none

**Environments**

- basic debugger
- profiling

**Description**

The CD or CHDIR command changes the current working directory from within the debugger. You can use relative pathnames as part of the *directory name*. If you do not use a *directory name*, the CD command displays the name of the current directory. You can also use the CD command to change the current drive. For example,

- `cd c:`
- `cd d:\csource`
- `cd c:\asmsrc`

**cls**  
*Clear Screen*

**Syntax** `cls`

**Menu selection** none

**Toolbar selection** none

**Environments**

- basic debugger
- profiling

**Description**

The CLS command clears all displayed information from the display area of the Command window.
Alphabetical Summary of Debugger Commands

**cnext**

*Single-Step C, Next Statement*

**Syntax**

`cnext [expression]`

**Menu selection**

`Target→Next C`

**Toolbar selection**

![Toolbar selection]

**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 are using CNEXT to step through assembly language code, the debugger does not update the display until it has executed all assembly language statements associated with a single C statement. Unlike CSTEP, CNEXT steps over function calls rather than stepping into them—you do not see the single-step execution of the function call.

The `expression` parameter specifies the number of statements that you want to single-step. You can use a conditional `expression` for conditional single-step execution. *(Running code conditionally, page 7-10, discusses this in detail.)*

**cstep**

*Single-Step C*

**Syntax**

`cstep [expression]`

**Menu selection**

`Target→Step C`

**Toolbar selection**

![Toolbar selection]

**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 are using CSTEP to step through assembly language code, the debugger does not update the display until it has executed all assembly language statements associated with a single C statement.

The `expression` parameter specifies the number of statements that you want to single-step. You can use a conditional `expression` for conditional single-step execution. *(Running code conditionally, page 7-10, discusses this in detail.)*
dasm, dir  Alphabetical Summary of Debugger Commands

**dasm**  
Display Disassembly at Specific Address

**Syntax**  
dasm  \{address | function name\}

**Menu selection**  
none

**Toolbar selection**  
none

**Environments**  
✔ basic debugger  ✔ profiling

**Description**  
The DASM command displays code beginning at a specific point within the Disassembly window.

**dir**  
List Directory Contents

**Syntax**  
dir  [directory name]

**Menu selection**  
none

**Toolbar selection**  
none

**Environments**  
✔ basic debugger  ✔ profiling

**Description**  
The DIR command displays a directory listing in the display area of the Command window. If you use the optional directory name parameter, the debugger displays a list of the specified directory’s contents. If you do not use the parameter, the debugger lists the contents of the current directory.
**disp**

Add Structure, Array, or Pointer to Watch Window

**Syntax**

disp expression [, display format]

**Menu selection**

none

**Toolbar selection**

none

**Environments**

☑ basic debugger ☐ profiling

**Description**

The DISP command opens a Watch window to display the contents of one of the following:

- An array
- A structure
- Pointer expressions to a scalar type (of the form *pointer)

If the expression is not one of these types, then DISP acts like a ? command.

When the Watch window is open, you can display the data pointed to by a pointer or display the members of the array or structure by clicking the box icon next to watched item:

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Result is displayed in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Default for the data type</td>
</tr>
<tr>
<td>c</td>
<td>ASCII character (bytes)</td>
</tr>
<tr>
<td>d</td>
<td>Decimal</td>
</tr>
<tr>
<td>e</td>
<td>Exponential floating point</td>
</tr>
<tr>
<td>f</td>
<td>Decimal floating point</td>
</tr>
<tr>
<td>o</td>
<td>Octal</td>
</tr>
<tr>
<td>p</td>
<td>Valid address</td>
</tr>
<tr>
<td>s</td>
<td>ASCII string</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal</td>
</tr>
<tr>
<td>x</td>
<td>Hexadecimal</td>
</tr>
</tbody>
</table>

You can use the display format parameter only when you are displaying a scalar type, an array of scalar type, or an individual member of an aggregate type.

You can also use the DISP command with a typecast expression to display memory contents in any format. Here are some examples:

```c
disp *0
disp *(float *)123
disp *(char *)0x111
```

This shows memory in the Watch window as an array of locations; the location that you specify with the expression parameter is member [0], and all other locations are offset from that location.
### dlog

**Record Display Area**

**Syntax**

```
dlog filename [,a | w]
or
dlog close
```

**Menu selection**

File→Log File

**Toolbar 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 and to record all commands that you enter from the command line, from the toolbar, from the menus, or with function keys.

To begin a recording session, use:

```
dlog filename
```

To end the recording session, enter:

```
dlog close
```

You can write over existing log files or append additional information to existing files. The optional parameters of the DLOG command control how existing log files are used:

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

- **Writing over an existing file.** Use the `w` parameter to open an existing file and write over the current contents of the file. This is the default action if you specify an existing filename without using either the `a` or `w` options; you will lose the contents of an existing file if you do not use the append (a) option.

### echo

**Echo String to Display Area**

**Syntax**

```
echo string
```

**Menu selection**

none

**Toolbar selection**

none

**Environments**

- basic debugger
- profiling

**Description**

The ECHO command displays `string` in the display area of the Command window. You cannot use quote marks around the `string`, and any leading blanks in your command string are removed when the ECHO command is executed. You can execute the ECHO command only in a batch file.
**Alphabetical Summary of Debugger Commands**

**else**, **endif**, **endloop**, **eval**, **file**

---

**else**

<table>
<thead>
<tr>
<th>Description</th>
<th>Execute Alternative Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELSE provides an alternative list of commands in the IF/ELSE/ENDIF command sequence. See page 10-22 for more information about these commands.</td>
<td></td>
</tr>
</tbody>
</table>

---

**endif**

<table>
<thead>
<tr>
<th>Description</th>
<th>Terminate Conditional Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENDIF identifies the end of a conditional-execution command sequence begun with an IF command. See page 10-22 for more information about these commands.</td>
<td></td>
</tr>
</tbody>
</table>

---

**endloop**

<table>
<thead>
<tr>
<th>Description</th>
<th>Terminate Looping Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENDLOOP identifies the end of the LOOP/ENDLOOP command sequence. See page 10-23 for more information about the LOOP/ENDLOOP commands.</td>
<td></td>
</tr>
</tbody>
</table>

---

**eval**

| Syntax | ```eval expression```
| Menu selection | none |
| Toolbar 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 is not necessary to display the result). |

---

**file**

| Syntax | ```file filename``` |
| Menu selection | File→Open |
| Toolbar selection | [File] |
| Environments | basic debugger, profiling |
| Description | The FILE command displays the contents of any text file in the File window. This command is intended primarily for displaying C source code. You can view only one text file at a time. |
### fill, fillb, func

**Alphabetical Summary of Debugger Commands**

---

**fill**

**Fill Memory Word by Word**

**Syntax**  
`fill address, length, data`

**Menu selection**  
Memory → Fill Word

**Toolbar selection**  
none

**Environments**  
✔️ basic debugger  
☐ profiling

**Description**  
The FILL command fills a block of memory word by word with a specified value.

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

---

**fillb**

**Fill Memory Byte by Byte**

**Syntax**  
`fillb address, length, data`

**Menu selection**  
Memory → Fill Byte

**Toolbar selection**  
none

**Environments**  
✔️ basic debugger  
☐ profiling

**Description**  
The FILLB command fills a block of memory byte by byte with a specified value.

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

---

**func**

**Display Function**

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

**Menu selection**  
none

**Toolbar selection**  
none

**Environments**  
✔️ basic debugger  
✔️ profiling

**Description**  
The FUNC command displays a specified C function in the File window. You can identify the function by its name or by an address in the function. FUNC works the same way FILE works, but with FUNC you do not need to identify the name of the file that contains the function.
Alphabetical Summary of Debugger Commands  

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>go</strong></td>
<td>Run to Specified Address</td>
</tr>
<tr>
<td>Syntax</td>
<td><code>go [address]</code></td>
</tr>
<tr>
<td>Menu selection</td>
<td>none</td>
</tr>
<tr>
<td>Toolbar selection</td>
<td>none</td>
</tr>
<tr>
<td>Environments</td>
<td>basic debugger, profiling</td>
</tr>
<tr>
<td>Description</td>
<td>The GO command executes code up to a specific point in your program. If you do not supply an address, then GO acts like a RUN command without an expression parameter.</td>
</tr>
</tbody>
</table>

| **halt** | Halt Target System |
| Syntax  | `halt` |
| Menu selection | Target→Halt! |
| Toolbar selection |  |
| Environments | basic debugger, profiling |
| Description | The HALT command halts the target system. When you invoke the debugger, it automatically executes a HALT command. |

| **help** | Display Help Topic for Debugger Command |
| Syntax  | `help [debugger command]` |
| Menu selection | none |
| Toolbar selection | none |
| Environments | basic debugger, profiling |
| Description | The HELP command opens a help topic that describes the debugger command. If you omit the debugger command, the debugger displays a list of help topics. |
if/else/endif, load  Alphabetical Summary of Debugger Commands

if/else/endif  Conditionally Execute Debugger Commands

Syntax  

```
if  expression
depthographer commands
[else
depthographer commands]
endif
```

Menu selection  none

Toolbar 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. 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 cannot nest conditional commands within the same batch file.

load  Load Executable Object File

Syntax  

```
load  object filename
```

Menu selection  File→Load Program

Toolbar selection  none

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 do not supply an extension, the debugger looks for filename.out. The LOAD command clears the old symbol table and closes any Watch windows.
Loop Through Debugger Commands

Syntax

```plaintext
loop expression
deplexer commands
dendloop
```

Menu selection

none

Toolbar 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 cannot nest LOOP/ENDLOOP commands within the same file.
ma  Alphabetical Summary of Debugger Commands

Add Block to Memory Map

Syntax

ma address, length, type

Menu selection

Memory→ Mapping

Toolbar selection

none

Environments

✓ basic debugger    ✓ profiling

Description

The MA command identifies valid ranges of target memory. A new memory range must not overlap an existing entry; if you define a range that overlaps an existing range, the debugger ignores the new range.

- The address parameter defines the starting address of a range in 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:

<table>
<thead>
<tr>
<th>To identify this kind of memory...</th>
<th>Use this keyword as the type parameter...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-only memory</td>
<td>R, ROM, or READONLY</td>
</tr>
<tr>
<td>Write-only memory</td>
<td>W, WOM, or WRITEONLY</td>
</tr>
<tr>
<td>Read/write memory</td>
<td>WR or RAM</td>
</tr>
<tr>
<td>No-access memory</td>
<td>PROTECT</td>
</tr>
<tr>
<td>Input port</td>
<td>INPORT or P</td>
</tr>
<tr>
<td>Output port</td>
<td>OUTPORT or P</td>
</tr>
<tr>
<td>Input/output port</td>
<td>IOPORT or P</td>
</tr>
</tbody>
</table>
map | Enable/Disable Memory Mapping

Syntax
map \{on \textbar off\}

Menu selection
Memory→Mapping

Toolbar selection
none

Environments
\checkmark basic debugger  \checkmark profiling

Description
The MAP command enables or disables memory mapping. Disabling memory mapping can cause bus fault problems in the target because the debugger may attempt to access nonexistent memory.

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

mc | Connect Simulated I/O Port to a File

Syntax
mc \textit{port address, filename,} \{READ \textbar WRITE\}

Menu selection
none

Toolbar selection
none

Environments
\checkmark basic debugger  \empty

Description
The MC command connects INPORT, OUTPORT, or IOPORT to an input or output file. Before you can connect the port, you must add it to the memory map with the MA command.

- The \textit{port address} parameter defines the address of the I/O port. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.

- The \textit{filename} parameter can be any filename. If you connect a port to read from a file, the file must exist or the MC command will fail.

- The final parameter is specified as \texttt{READ} or \texttt{WRITE} and defines how the file will be used (for input or output, respectively).

The file is accessed during an IN or OUT instruction to the associated port address. Any port in I/O space can be connected to a file. A maximum of one input and one output file can be connected to a single port; multiple ports can be connected to a single file. Memory-mapped ports can also be connected to files; any instruction that reads or writes to the memory-mapped port reads or writes to the associated file.
### md  Alphabetical Summary of Debugger Commands

<table>
<thead>
<tr>
<th>Syntax</th>
<th>md address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu selection</td>
<td>Memory→Mapping</td>
</tr>
<tr>
<td>Toolbar selection</td>
<td>none</td>
</tr>
<tr>
<td>Environments</td>
<td>✔ basic debugger ✔ profiling</td>
</tr>
</tbody>
</table>

**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:

```plaintext
Specified map not found
```

**Note:**

If you want to use the MD command to remove a simulated I/O port, you must first disconnect the port with the MI command.
**mem**

*Modify Memory Window Display*

**Syntax**

`mem  expression [, [display format] [, window name] ]`

**Menu selection**

none

**Toolbar 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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Result is displayed in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Default for the data type</td>
</tr>
<tr>
<td>c</td>
<td>ASCII character (bytes)</td>
</tr>
<tr>
<td>d</td>
<td>Decimal</td>
</tr>
<tr>
<td>e</td>
<td>Exponential floating point</td>
</tr>
<tr>
<td>f</td>
<td>Decimal floating point</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Result is displayed in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>Octal</td>
</tr>
<tr>
<td>p</td>
<td>Valid address</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal</td>
</tr>
<tr>
<td>x</td>
<td>Hexadecimal</td>
</tr>
</tbody>
</table>

**mi**

*Disconnect I/O Port*

**Syntax**

`mi  port address, {READ | WRITE}`

**Menu selection**

none

**Toolbar selection**

none

**Environments**

- [ ] basic debugger
- [ ] profiling

**Description**

The MI command disconnects a simulated I/O port from its associated system file.

The `port address` parameter identifies the address of the I/O port, which must have been previously defined with the MC command. The read/write characteristics must match the parameter used when the port was connected.
**mix, ml, move**  
*Alphabetical Summary of Debugger Commands*

### mix

**Enter Mixed Mode**

**Syntax**  
mix

**Menu selection**  
Mode→Mixed

**Toolbar selection**  
none

**Environments**  
☑ basic debugger  
☐ profiling

**Description**  
The MIX command changes from the current debugging mode to mixed mode. If you are already in mixed mode, the MIX command has no effect.

### ml

**List Memory Map**

**Syntax**  
ml

**Menu selection**  
Memory→Mapping

**Toolbar selection**  
none

**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 a Window**

**Syntax**  
move window name [, [X position] [, [Y position] [, [width] [, length]]]]

**Menu selection**  
none

**Toolbar selection**  
none

**Environments**  
☑ basic debugger  
☑ profiling

**Description**  
The MOVE command moves the upper left corner of the window to the specified XY position, repositioning the rest of the window relative to that corner. If you choose, you can resize the window while you move it (see the SIZE command for valid width and length values). Specify the X position, Y position, width, and length parameters in pixels. If you omit these parameters, the MOVE command defaults to the window’s current position and size.

You can spell out the entire window name, but you really need to specify only enough letters to identify the window.
### mr  Reset Memory Map

**Syntax**  
```  
```

**Menu selection**  
none

**Toolbar selection**  
Memory → Mapping

**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**  
Memory → Save

**Toolbar selection**  
none

**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 do not supply an extension, the debugger adds an .obj extension.
### next

**Single-Step, Next Statement**

<table>
<thead>
<tr>
<th>Syntax</th>
<th><code>next [expression]</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu selection</td>
<td>Target→Next</td>
</tr>
<tr>
<td>Toolbar selection</td>
<td>![Toolbar Icon]</td>
</tr>
<tr>
<td>Environments</td>
<td>basic debugger, profiling</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The NEXT command is similar to the STEP command. If you are in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time. Unlike STEP, NEXT never updates the display when executing called functions; NEXT always steps to the next consecutive statement. Unlike STEP, NEXT steps over function calls rather than stepping into them—you do not see the single-step execution of the function call. The optional expression parameter specifies the number of statements that you want to single-step. You can use a conditional expression for conditional single-step execution. <em>(Running code conditionally, page 7-10, discusses this in detail.)</em></td>
</tr>
</tbody>
</table>

### pause

**Pause Execution**

<table>
<thead>
<tr>
<th>Syntax</th>
<th><code>pause</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu selection</td>
<td>none</td>
</tr>
<tr>
<td>Toolbar selection</td>
<td>none</td>
</tr>
<tr>
<td>Environments</td>
<td>basic debugger, profiling</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>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 a dialog box. To continue processing, click OK or press Enter.</td>
</tr>
</tbody>
</table>

10-30
Alphabetical Summary of Debugger Commands

**pf**

**Profile, Full**

**Syntax**

pf  starting point [, update rate]

**Menu selection**

Profile→Run

**Toolbar selection**

Menu selection

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 stopping point encountered. The starting point parameter can be a label, a function name, or a memory address.

The optional update rate parameter determines how often the Profile window is updated. The update rate parameter can have one of these values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>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).</td>
</tr>
<tr>
<td>Greater than or equal to 1</td>
<td>Statistics are updated during the session. A value of 1 means that data is updated as often as possible.</td>
</tr>
</tbody>
</table>

**pq**

**Profile, Quick**

**Syntax**

pq  starting point [, update rate]

**Menu selection**

Profile→Run

**Toolbar selection**

Menu selection

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 stopping point encountered. PQ is similar to PF, except that PQ does not collect exclusive or exclusive max data.

The update rate parameter is the same as for the PF command.
**pr, prompt**   
*Alphabetical Summary of Debugger Commands*

### pr  
**Resume Profiling Session**

**Syntax**

`pr [clear data [, update rate]]`

**Menu selection**

Profile→Run

**Toolbar selection**

- ![Profile](image)

**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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>This is the default. The profiler continues to collect data (adding the data to the existing data for the profiled areas) and to use the previous internal profile stacks.</td>
</tr>
<tr>
<td>nonzero</td>
<td>All previously collected profile data and internal profile stacks are cleared.</td>
</tr>
</tbody>
</table>

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**

none

**Toolbar selection**

none

**Environments**

- basic debugger  
- profiling

**Description**

The PROMPT command changes the command-line prompt. The `new prompt` can be any string of characters (a semicolon or comma ends the string). The `new prompt` cannot be longer than 132 characters.
### Alphabetical Summary of Debugger Commands

#### quit

<table>
<thead>
<tr>
<th>Syntax</th>
<th>quit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu selection</td>
<td>File→Exit</td>
</tr>
<tr>
<td>Toolbar selection</td>
<td>none</td>
</tr>
<tr>
<td>Environments</td>
<td>basic debugger, profiling</td>
</tr>
<tr>
<td>Description</td>
<td>The QUIT command exits the debugger and returns to the operating system.</td>
</tr>
</tbody>
</table>

#### reload

<table>
<thead>
<tr>
<th>Syntax</th>
<th>reload [object filename]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu selection</td>
<td>File→Reload Program</td>
</tr>
<tr>
<td>Toolbar selection</td>
<td>none</td>
</tr>
<tr>
<td>Environments</td>
<td>basic debugger, profiling</td>
</tr>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
</tbody>
</table>

#### reset

<table>
<thead>
<tr>
<th>Syntax</th>
<th>reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu selection</td>
<td>Target→Reset Target</td>
</tr>
<tr>
<td>Toolbar selection</td>
<td>none</td>
</tr>
<tr>
<td>Environments</td>
<td>basic debugger, profiling</td>
</tr>
<tr>
<td>Description</td>
<td>The RESET command resets the simulator. This is a software reset. If you execute the RESET command, the simulator simulates the processor and peripheral reset operation, putting the processor in a known state.</td>
</tr>
</tbody>
</table>
restart, return  Alphabetical Summary of Debugger Commands

**restart**  
*Reset PC to Program Entry Point*

**Syntax**  
restart  
ret

**Menu selection**  
Target→Restart

**Toolbar selection**

**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**  
Target→Return

**Toolbar selection**

**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 doing one of the following:

- Click the Halt icon on the toolbar:

- From the Target menu, select Halt!

- Press \( \text{ESC} \).
run

**Run Code**

**Syntax**

run  [expression]

**Menu selection**

Target→Run

**Toolbar selection**

Run

**Environments**

✔ basic debugger

**Description**

The RUN command is the basic command for running an entire program. The command’s behavior depends on the type of parameter you supply:

- If you do not supply an expression, the program executes until it encounters a breakpoint or until you do one of the following:
  - Click the Halt icon on the toolbar:
  - From the Target menu, select Halt!.
  - Press (ESC).

- If you supply a logical or relational expression, the run becomes conditional (see Running code conditionally, page 7-10).

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

runb

**Benchmark Code**

**Syntax**

runb

**Menu selection**

none

**Environments**

✔ basic debugger

**Description**

The RUNB command executes a specific section of code and counts the number of CPU clock cycles consumed by the execution. In order to operate correctly, execution must be halted by a software breakpoint. After RUNB execution halts, the debugger stores the number of cycles into the CLK pseudoregister. For a complete explanation of the RUNB command and the benchmarking process, read Section 7.5, Benchmarking, on page 7-11.
**sa, sconfig, sd**  
*Alphabetical Summary of Debugger Commands*

### sa  
*Add Stopping Point*

**Syntax**  
```sh
sa  address
```

**Menu selection**  
none

**Toolbar selection**  
none

**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.

### sconfig  
*Load Screen Configuration*

**Syntax**  
```sh
sconfig  [filename]
```

**Menu selection**  
File→Load Config

**Toolbar selection**  
none

**Environments**  
- basic debugger
- profiling

**Description**  
The SCONFIG command restores the display to a specified configuration. This restores the window locations and sizes that were saved with the SSAVE command into `filename`. The debugger searches for the specified file in the current directory and then in directories named with the D_DIR environment variable. If you do not supply a `filename`, the debugger looks for init.clr.

When you use SCONFIG to restore a configuration that includes multiple File, Watch, or Memory windows, the additional windows are not opened automatically. However, when you open an additional window and use a `window name` that matches a window name that you used before you saved the configuration, the window is placed in the saved location.

### sd  
*Delete Stopping Point*

**Syntax**  
```sh
sd  address
```

**Menu selection**  
none

**Toolbar selection**  
none

**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

**Toolbar 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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Result is displayed in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Default for the data type</td>
</tr>
<tr>
<td>c</td>
<td>ASCII character (bytes)</td>
</tr>
<tr>
<td>d</td>
<td>Decimal</td>
</tr>
<tr>
<td>e</td>
<td>Exponential floating point</td>
</tr>
<tr>
<td>f</td>
<td>Decimal floating point</td>
</tr>
<tr>
<td>o</td>
<td>Octal</td>
</tr>
<tr>
<td>p</td>
<td>Valid address</td>
</tr>
<tr>
<td>s</td>
<td>ASCII string</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal</td>
</tr>
<tr>
<td>x</td>
<td>Hexadecimal</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Valid Display Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>char (c)</td>
<td>√ √ √ √ √</td>
</tr>
<tr>
<td>uchar (d)</td>
<td>√ √ √ √ √</td>
</tr>
<tr>
<td>short (d)</td>
<td>√ √ √ √ √</td>
</tr>
<tr>
<td>int (d)</td>
<td>√ √ √</td>
</tr>
<tr>
<td>uint (d)</td>
<td>√ √ √</td>
</tr>
<tr>
<td>long (d)</td>
<td>√ √ √</td>
</tr>
<tr>
<td>ulong (d)</td>
<td>√ √ √</td>
</tr>
<tr>
<td>float (e)</td>
<td>√ √ √</td>
</tr>
<tr>
<td>double (e)</td>
<td>√ √ √</td>
</tr>
<tr>
<td>ptr (p)</td>
<td>√ √ √</td>
</tr>
</tbody>
</table>

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

setf *
### size, sl, sload  
*Alphabetical Summary of Debugger Commands*

#### size

**Size a Window**

**Syntax**

```
size window name [, [width] [, length]]
```

**Menu selection**

none

**Toolbar selection**

none

**Environments**

- basic debugger
- profiling

**Description**

The `SIZE` command changes the size of the window. Specify the `width` and `length` parameters in pixels. If you omit these parameters, the `SIZE` command defaults to the window's current size.

You can spell out the entire `window name`, but you really need to specify only enough letters to identify the window.

#### sl

**List Stopping Point**

**Syntax**

```
sl
```

**Menu selection**

none

**Toolbar selection**

none

**Environments**

- basic debugger
- profiling

**Description**

The `SL` command lists all of the currently set stopping points.

#### sload

**Load Symbol Table**

**Syntax**

```
sload object filename
```

**Menu selection**

File → Load Symbols

**Toolbar selection**

none

**Environments**

- basic debugger
- profiling

**Description**

The `SLOAD` command loads the symbol table of the specified object file. `SLOAD` is useful in an emulation environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). In such an environment, loading the symbol table allows you to perform symbolic debugging and examine the values of C variables.

`SLOAD` clears the existing symbol table before loading the new one but does not modify memory or set the program entry point. `SLOAD` closes any Watch windows.
sound

Enable Error Beeping

Syntax  
sound  (on | off)

Menu selection  none

Toolbar selection  none

Environments  

basic debugger  profiling

Description  You can cause a beep to sound every time a debugger error message is displayed. This is useful if the Command window is hidden (because you would not see the error message). By default, sound is off.

sr

Reset Stopping Point

Syntax  
sr

Menu selection  none

Toolbar selection  none

Environments  

basic debugger  profiling

Description  The SR command resets (deletes) all currently set stopping points.
### ssave (Save Screen Configuration)

**Syntax**
```
ssave [filename]
```

**Menu selection**
File→Save As Config

**Toolbar selection**
none

**Environments**
☑️ basic debugger
☑️ profiling

**Description**
The `ssave` command saves the current screen configuration to a file. This saves the window locations and window sizes for all debugging modes, including the size and location for multiple File, Watch, and Memory windows. However, the debugger does not save docking information about docked windows. If you have one or more docked windows, you save the screen configuration, and you load that configuration, the debugger does not display any windows as docked. If you want the windows docked, you must follow the docking procedure again.

The `filename` parameter names the screen configuration file. You can include path information (including relative pathnames); if you do not supply path information, the debugger places the file in the current directory. If you do not supply a `filename`, the debugger saves the current configuration into a file named `init.clr` and places the file in the current directory.

If you use a filename that already exists, the debugger overwrites the file with the current configuration.
Alphabetical Summary of Debugger Commands

**step**

**Single-Step**

**Syntax**

`step [expression]`

**Menu selection**

Target→Step

**Toolbar selection**

![Step button]

**Environments**

- ✔ basic debugger
- □ profiling

**Description**

The **STEP** command single-steps through assembly language or C code. If you are in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time.

If you are single-stepping through C code and encounter a function call, the **STEP** command shows you the single-step execution of the called function (assuming that the function was compiled with the compiler’s –g option). When function execution is complete, single-step execution returns to the caller. If the function was not compiled with the –g option, the debugger executes the function but does not show single-step execution of the function.

The **expression** parameter specifies the number of statements that you want to single-step. You can use a conditional **expression** for conditional single-step execution. *(Running code conditionally, page 7-10, discusses this in detail.)*
**system** *Alphabetical Summary of Debugger Commands*

### system

**Enter Operating-System Command**

**Syntax**

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

**Menu selection**

none

**Environments**

- ✓ basic debugger
- ✓ profiling

**Description**

The SYSTEM command allows you to enter operating-system commands without explicitly exiting the debugger environment. If you enter SYSTEM with no parameters, the debugger opens a system shell and displays the operating-system prompt. At this point, you can enter any operating-system command. When you finish, enter:

```
exit ②
```

If you prefer, you can supply the operating-system command as a parameter to the SYSTEM command. If the result of the command is a message or other display, the debugger blanks the top of the debugger display to show the information. In this case, you can use the flag parameter to tell the debugger whether or not it should hesitate after displaying the information. The flag can be 0 or 1.

- 0 If you supply a value of 0 for flag, the debugger immediately returns to the debugger environment after the last item of information is displayed.

- 1 If you supply a value of 1 for flag, the debugger does not return to the debugger environment until you press ②. (This is the default.)
Alphabetical Summary of Debugger Commands

take, unalias

**take**

*Execute Batch File*

**Syntax**

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

**Menu selection**

File→Take

**Toolbar 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 do not 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. To suppress the echoing and updating, enter a 0 as the `suppress echo flag` parameter. If you omit the `suppress echo flag` parameter or enter a nonzero value for that parameter, the debugger behaves in the default manner.

**unalias**

*Delete Alias Definition*

**Syntax**

```
unalias  alias name
unalias  *
```

**Menu selection**

Setup→Alias Commands

**Toolbar 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 *

  The * symbol does not work as a wildcard.
**use, vaa, vac**  
*Alphabetical Summary of Debugger Commands*

### use

**Use Additional Directory**

**Syntax**

use [directory name]

**Menu selection**

none

**Toolbar 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 in the display area of the Command window all of the current directories.

### vaa

**Save All Profile Data to a File**

**Syntax**

vaa filename

**Menu selection**

Profile→Save All

**Toolbar selection**

none

**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**

Profile→Save View

**Toolbar selection**

none

**Environments**

- basic debugger
- profiling

**Description**

The VAC command saves all statistics currently displayed in the Profile window. (Statistics that are not displayed are not saved.) The data is stored in a system file.
### version

**Display the Current Debugger Version**

**Syntax**

version

**Menu selection**

none

**Toolbar selection**

none

**Environments**

- basic debugger
- profiling

**Description**

The VERSION command displays the debugger’s copyright date and the version number of the debugger and the simulator.

### vr

**Reset Profile Window Display**

**Syntax**

vr

**Menu selection**

none

**Toolbar selection**

none

**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.
**Add Item to Watch Window**

**Syntax**

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

**Menu selection**

Setup → Watch Variable

**Toolbar selection**

none

**Environments**

- basic debugger
- profiling

**Description**

The WA command displays the value of expression in a Watch window. If a Watch window is not open, executing WA opens a Watch window. The expression parameter can be any C expression, including an expression that has side effects.

WA is most useful for watching an expression whose value changes over time; constant expressions serve no useful function in the watch window. The label parameter is optional. When used, it provides a label for the watched entry. If you do not use a label, the debugger displays the expression in the label field.

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Result is displayed in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Default for the data type</td>
</tr>
<tr>
<td>c</td>
<td>ASCII character (bytes)</td>
</tr>
<tr>
<td>d</td>
<td>Decimal</td>
</tr>
<tr>
<td>e</td>
<td>Exponential floating point</td>
</tr>
<tr>
<td>f</td>
<td>Decimal floating point</td>
</tr>
<tr>
<td>o</td>
<td>Octal</td>
</tr>
<tr>
<td>p</td>
<td>Valid address</td>
</tr>
<tr>
<td>s</td>
<td>ASCII string</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal</td>
</tr>
<tr>
<td>x</td>
<td>Hexadecimal</td>
</tr>
</tbody>
</table>

If you want to use a display format parameter without a label parameter, be sure to include an extra comma. For example:

```
wa PC, , o
```

You can open additional Watch windows by using the window name parameter. When you open an additional Watch window, the debugger appends the window name to the Watch window label. You can create as many Watch windows as you need.

If you omit the window name parameter, the debugger displays the expression in the default Watch window (labeled Watch).
## Alphabetical Summary of Debugger Commands

### wd

**Delete Item From Watch Window**

**Syntax**

\[ wd \ expression [ , \ window \ name ] \]

**Menu selection**

Setup→Watch Variable

**Toolbar selection**

none

**Environments**

- basic debugger
- profiling

**Description**

The WD command deletes a specific item from the Watch window. The WD command's `expression` parameter must correspond to one of the variable names listed in the Watch window. The optional `window name` parameter specifies a particular Watch window. If no window names is given, the expression is deleted from the default Watch window.

### whatis

**Find Data Type**

**Syntax**

\[ whatis \ symbol \]

**Menu selection**

none

**Toolbar selection**

none

**Environments**

- basic debugger
- profiling

**Description**

The WHATIS command shows the data type of `symbol` in the display area of the Command window. The `symbol` can be any variable (local, global, or static), a function name, structure tag, typedef name, or enumeration constant.

### win

**Make a Window Active**

**Syntax**

\[ win \ window \ name \]

**Menu selection**

View menu options

**Toolbar selection**

none

**Environments**

- basic debugger
- profiling

**Description**

The WIN command allows you to make a window active by name. You can spell out the entire window name, but you really need to specify only enough letters to identify the window.

If you supply an ambiguous name (such as C, which could stand for CPU or Calls), the debugger selects the first window it finds whose name matches the name you supplied. If the debugger does not find the window you asked for (because you closed the window or misspelled the name), then the WIN command has no effect.
**wr**  
*Close Watch Window*

**Syntax**
wr  [* | window name*] 

**Menu selection**
Setup→Watch Variable

**Toolbar selection**
none

**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  window name
  ```

- To close all Watch windows, enter:
  ```
  wr *
  ```

**zoom**  
*Zoom a Window*

**Syntax**
zoom  [window name]

**Menu selection**
none

**Toolbar selection**
none

**Environments**
- basic debugger
- profiling

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

You can spell out the entire window name, but you really need to specify only enough letters to identify the window.
10.3 Summary of Profiling Commands

The following tables summarize the profiling commands that are used for marking, enabling, disabling, and unmarking areas and for changing the display in the Profile window. These commands are easiest to use from the Profile menu and associated dialog boxes, so they are not included in the alphabetical command summary. The syntaxes for these commands are provided here so that you can include them in batch files.

**Table 10–1. Marking Areas**

<table>
<thead>
<tr>
<th>To mark this area...</th>
<th>In C only</th>
<th>In disassembly only</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lines</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>☐ By line number, address</td>
<td>MCLE filename, line number</td>
<td>MALE address</td>
</tr>
<tr>
<td>☐ All lines in a function</td>
<td>MCLF function</td>
<td>MALF function</td>
</tr>
<tr>
<td><strong>Ranges</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>☐ By line numbers</td>
<td>MCRE filename, line number, line number</td>
<td>MARE address, address</td>
</tr>
<tr>
<td><strong>Functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>☐ By function name</td>
<td>MCFE function</td>
<td>not applicable</td>
</tr>
<tr>
<td>☐ All functions in a module</td>
<td>MCFM filename</td>
<td>MCFG</td>
</tr>
<tr>
<td>☐ All functions everywhere</td>
<td>MCFG</td>
<td></td>
</tr>
</tbody>
</table>

**Table 10–2. Disabling Marked Areas**

<table>
<thead>
<tr>
<th>To disable this area...</th>
<th>In C only</th>
<th>In disassembly only</th>
<th>In C and disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lines</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>☐ By line number, address</td>
<td>DCLE filename, line number</td>
<td>DALE address</td>
<td>not applicable</td>
</tr>
<tr>
<td>☐ All lines in a function</td>
<td>DCLF function</td>
<td>DALF function</td>
<td>DBLF function</td>
</tr>
<tr>
<td>☐ All lines in a module</td>
<td>DCLM filename</td>
<td>DALM filename</td>
<td>DBLM filename</td>
</tr>
<tr>
<td>☐ All lines everywhere</td>
<td>DCLG</td>
<td>DALG</td>
<td>DBLG</td>
</tr>
<tr>
<td><strong>Ranges</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>☐ By line number, address</td>
<td>DCRE filename, line number</td>
<td>DARE address</td>
<td>not applicable</td>
</tr>
<tr>
<td>☐ All ranges in a function</td>
<td>DCRF function</td>
<td>DARF function</td>
<td>DBRF function</td>
</tr>
<tr>
<td>☐ All ranges in a module</td>
<td>DCRM filename</td>
<td>DARM filename</td>
<td>DBRM filename</td>
</tr>
<tr>
<td>☐ All ranges everywhere</td>
<td>DCRG</td>
<td>DARG</td>
<td>DBRG</td>
</tr>
</tbody>
</table>
### Table 10–2. Disabling Marked Areas (Continued)

<table>
<thead>
<tr>
<th>To disable this area...</th>
<th>In C only</th>
<th>In disassembly only</th>
<th>In C and disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By function name</td>
<td>DCFE function</td>
<td>not applicable</td>
<td>not applicable</td>
</tr>
<tr>
<td>All functions in a module</td>
<td>DCFM filename</td>
<td></td>
<td>DBFM filename</td>
</tr>
<tr>
<td>All functions everywhere</td>
<td>DCFG</td>
<td></td>
<td>DBFG</td>
</tr>
<tr>
<td><strong>All areas</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All areas in a function</td>
<td>DCAF function</td>
<td>DAAF function</td>
<td>DBAF function</td>
</tr>
<tr>
<td>All areas in a module</td>
<td>DCAM filename</td>
<td>DAAM filename</td>
<td>DBAM filename</td>
</tr>
<tr>
<td>All areas everywhere</td>
<td>DCAG</td>
<td>DAAG</td>
<td>DBAG</td>
</tr>
</tbody>
</table>

### Table 10–3. Enabling Disabled Areas

<table>
<thead>
<tr>
<th>To enable this area...</th>
<th>In C only</th>
<th>In disassembly only</th>
<th>In C and disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lines</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By line number, address</td>
<td>ECLE filename, line number</td>
<td>EALE address</td>
<td>not applicable</td>
</tr>
<tr>
<td>All lines in a function</td>
<td>ECLF function</td>
<td>EALF function</td>
<td>EBLF function</td>
</tr>
<tr>
<td>All lines in a module</td>
<td>ECLM filename</td>
<td>EALM filename</td>
<td>EBLM filename</td>
</tr>
<tr>
<td>All lines everywhere</td>
<td>ECLG</td>
<td>EALG</td>
<td>EBLG</td>
</tr>
<tr>
<td><strong>Ranges</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By line number, address</td>
<td>ECRE filename, line number</td>
<td>EARE address</td>
<td>not applicable</td>
</tr>
<tr>
<td>All ranges in a function</td>
<td>ECRF function</td>
<td>EARF function</td>
<td>EBRF function</td>
</tr>
<tr>
<td>All ranges in a module</td>
<td>ECRM filename</td>
<td>EARM filename</td>
<td>EBRM filename</td>
</tr>
<tr>
<td>All ranges everywhere</td>
<td>ECRG</td>
<td>EARG</td>
<td>EBRG</td>
</tr>
<tr>
<td><strong>Functions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By function name</td>
<td>ECFE function</td>
<td>not applicable</td>
<td>not applicable</td>
</tr>
<tr>
<td>All functions in a module</td>
<td>ECFM filename</td>
<td></td>
<td>EBFM filename</td>
</tr>
<tr>
<td>All functions everywhere</td>
<td>ECFG</td>
<td></td>
<td>EBFG</td>
</tr>
<tr>
<td><strong>All areas</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All areas in a function</td>
<td>ECAF function</td>
<td>EAAF function</td>
<td>EBAF function</td>
</tr>
<tr>
<td>All areas in a module</td>
<td>ECAM filename</td>
<td>EAAM filename</td>
<td>EBAM filename</td>
</tr>
<tr>
<td>All areas everywhere</td>
<td>ECAG</td>
<td>EAAG</td>
<td>EBAG</td>
</tr>
</tbody>
</table>

10-50
### Table 10–4. Unmarking Areas

<table>
<thead>
<tr>
<th>To unmark this area...</th>
<th>In C only</th>
<th>In disassembly only</th>
<th>In C and disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lines</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By line number, address</td>
<td>UCLE</td>
<td>UALE</td>
<td>not applicable</td>
</tr>
<tr>
<td></td>
<td>filename, line number</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>All lines in a function</td>
<td>UCLF</td>
<td>UALF</td>
<td>UBFL</td>
</tr>
<tr>
<td></td>
<td>function</td>
<td>function</td>
<td>function</td>
</tr>
<tr>
<td>All lines in a module</td>
<td>UCLM</td>
<td>UALM</td>
<td>UBLM</td>
</tr>
<tr>
<td></td>
<td>filename</td>
<td>filename</td>
<td>filename</td>
</tr>
<tr>
<td>All lines everywhere</td>
<td>UCLG</td>
<td>UALG</td>
<td>UBLG</td>
</tr>
<tr>
<td><strong>Ranges</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By line number, address</td>
<td>UCRE</td>
<td>UARE</td>
<td>not applicable</td>
</tr>
<tr>
<td></td>
<td>filename, line number</td>
<td>address</td>
<td></td>
</tr>
<tr>
<td>All ranges in a function</td>
<td>UCRF</td>
<td>UARF</td>
<td>UBRF</td>
</tr>
<tr>
<td></td>
<td>function</td>
<td>function</td>
<td>function</td>
</tr>
<tr>
<td>All ranges in a module</td>
<td>UCRM</td>
<td>UARM</td>
<td>UBRM</td>
</tr>
<tr>
<td></td>
<td>filename</td>
<td>filename</td>
<td>filename</td>
</tr>
<tr>
<td>All ranges everywhere</td>
<td>UCRG</td>
<td>UARG</td>
<td>UBRG</td>
</tr>
<tr>
<td><strong>Functions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By function name</td>
<td>UCFE</td>
<td>not applicable</td>
<td>not applicable</td>
</tr>
<tr>
<td></td>
<td>function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All functions in a module</td>
<td>UCFM</td>
<td>UBFM</td>
<td>not applicable</td>
</tr>
<tr>
<td></td>
<td>filename</td>
<td>filename</td>
<td></td>
</tr>
<tr>
<td>All functions everywhere</td>
<td>UCFG</td>
<td>UBFG</td>
<td></td>
</tr>
<tr>
<td><strong>All areas</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All areas in a function</td>
<td>UCAF</td>
<td>UAAF</td>
<td>UBAF</td>
</tr>
<tr>
<td></td>
<td>function</td>
<td>function</td>
<td>function</td>
</tr>
<tr>
<td>All areas in a module</td>
<td>UCAM</td>
<td>UAAM</td>
<td>UBAM</td>
</tr>
<tr>
<td></td>
<td>filename</td>
<td>filename</td>
<td>filename</td>
</tr>
<tr>
<td>All areas everywhere</td>
<td>UCAG</td>
<td>UAAG</td>
<td>UBAG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Summary of Profiling Commands

Table 10–5. Changing the Profile Window Display

(a) Viewing specific areas

<table>
<thead>
<tr>
<th>To view this area...</th>
<th>In C only</th>
<th>In disassembly only</th>
<th>In C and disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lines</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By line number, address</td>
<td>VFCLE filename, line number</td>
<td>VFALE address</td>
<td>not applicable</td>
</tr>
<tr>
<td>All lines in a function</td>
<td>VFCLF function</td>
<td>VFALF function</td>
<td>VFBLF function</td>
</tr>
<tr>
<td>All lines in a module</td>
<td>VFCLM filename</td>
<td>VFALM filename</td>
<td>VFBLM filename</td>
</tr>
<tr>
<td>All lines everywhere</td>
<td>VFCLG</td>
<td>VFALG</td>
<td>VFBLG</td>
</tr>
<tr>
<td><strong>Ranges</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By line number, address</td>
<td>VFCRE filename, line number</td>
<td>VFARE address</td>
<td>not applicable</td>
</tr>
<tr>
<td>All ranges in a function</td>
<td>VFCRF function</td>
<td>VFARF function</td>
<td>VFBRF function</td>
</tr>
<tr>
<td>All ranges in a module</td>
<td>VFCRM filename</td>
<td>VFARM filename</td>
<td>VFBRM filename</td>
</tr>
<tr>
<td>All ranges everywhere</td>
<td>VFCRG</td>
<td>VFARG</td>
<td>VFBRG</td>
</tr>
<tr>
<td><strong>Functions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By function name</td>
<td>VFCFE function</td>
<td>not applicable</td>
<td>not applicable</td>
</tr>
<tr>
<td>All functions in a module</td>
<td>VFCFM filename</td>
<td>VFBFM filename</td>
<td></td>
</tr>
<tr>
<td>All functions everywhere</td>
<td>VFCFG</td>
<td>VFBFG</td>
<td></td>
</tr>
<tr>
<td><strong>All areas</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All areas in a function</td>
<td>VFCAF function</td>
<td>VFAAF function</td>
<td>VFBAF function</td>
</tr>
<tr>
<td>All areas in a module</td>
<td>VFCAM filename</td>
<td>VFAAM filename</td>
<td>VFBAAM filename</td>
</tr>
<tr>
<td>All areas everywhere</td>
<td>VFCAG</td>
<td>VFAAG</td>
<td>VFBAG</td>
</tr>
</tbody>
</table>

(b) Viewing different data

<table>
<thead>
<tr>
<th>To view this information...</th>
<th>Use this command...</th>
<th>To sort on this data...</th>
<th>Use this command...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>VDC</td>
<td>Count</td>
<td>VSC</td>
</tr>
<tr>
<td>Inclusive</td>
<td>VDI</td>
<td>Inclusive</td>
<td>VSI</td>
</tr>
<tr>
<td>Inclusive, maximum</td>
<td>VDN</td>
<td>Inclusive, maximum</td>
<td>VSN</td>
</tr>
<tr>
<td>Exclusive</td>
<td>VDE</td>
<td>Exclusive</td>
<td>VSE</td>
</tr>
<tr>
<td>Exclusive, maximum</td>
<td>VDX</td>
<td>Exclusive, maximum</td>
<td>VSX</td>
</tr>
<tr>
<td>Address</td>
<td>VDA</td>
<td>Address</td>
<td>VSA</td>
</tr>
<tr>
<td>All</td>
<td>VDL</td>
<td>Data</td>
<td>VSD</td>
</tr>
</tbody>
</table>

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

### Chapter 11

**Using Batch Files to Enter Commands**

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

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 Creating a Batch File</td>
<td>11-2</td>
</tr>
<tr>
<td>11.2 Executing a Batch File</td>
<td>11-6</td>
</tr>
<tr>
<td>11.3 Recording Information From the Display Area</td>
<td>11-7</td>
</tr>
</tbody>
</table>
11.1 Creating a Batch File

You can create the batch file in any text editor. For each debugger command that you include in the batch file, use the same syntax that you would use if you were entering the command from the debugger's command line. Example 11–1 shows a sample batch file that you can create.

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

Example 11–1. Sample Batch File for Use With the Debugger

```
echo Loading object code
load testcode.out

echo Loading screen configuration
sconfig myconfig.clr

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

Echoing strings in a batch file

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

```
echo string
```

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

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

```
echo Creating new memory map
```

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

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

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

Any leading blanks in your string are removed when the ECHO command is executed.
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 execute debugger commands conditionally in a batch file, use the IF/ELSE/ENDIF commands. The syntax is:

```
if Boolean expression
decoder commands
[else
decoder commands]
endif
```

The debugger includes some predefined constants for use with IF. These constants evaluate to 0 (false) or 1 (true). Table 11–1 shows the constants and their corresponding tools.

Table 11–1. Predefined Constants for Use With Conditional Commands

<table>
<thead>
<tr>
<th>Constant</th>
<th>Debugger Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$SIM$$</td>
<td>Simulator</td>
</tr>
</tbody>
</table>

If the Boolean expression evaluates to true (1), the debugger executes all commands between the IF and ELSE or ENDIF. The ELSE portion of the command is optional. (See Chapter 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
decoder commands
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:

```plaintext
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:

```plaintext
>, >=, <, <=, ==, !=, &&, ||,!
```

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

```plaintext
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 cannot nest conditional and looping commands within the same batch file.
Pausing the execution of a batch file

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

```
pause
```

When the debugger reads this command in a batch file, the debugger stops execution and displays a dialog box. To continue processing, click OK or press Enter.
11.2 Executing a Batch File

Once you create a batch file, you can tell the debugger to read and execute commands from the batch file. To do so, follow these steps:

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

![Open Take File dialog box]

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

3) Click Open.

This causes the debugger to read and execute the commands in the batch file. To halt the debugger’s execution of a batch file, press **ESC**.
11.3 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 have entered from the command line, from the toolbar, from the menus, or with function keys. The log file also contains the results from commands and error or progress messages. The debugger automatically precedes all error or progress messages and command results with a semicolon to turn them into comments. This way, you can easily reexecute the commands in your log file by using the File→Take menu option.

To begin a recording session, follow these steps:

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

2) To open a new log file, click Open File. This displays the Open Log File dialog box:

   Enter a name for the log file. Use a .log extension.
Recording Information From the Display Area

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

4) In the File name field, enter a name for the log file. You can use a .log extension to identify the file as a log file.

5) Click Open.

6) In the Log File dialog box, select one of the following actions if the file that you opened already exists:
   - Append to add the log information to an existing file
   - Overwrite to write over the contents of an existing file

7) Click OK.

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

To end the recording session, follow these steps:

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

```

Log File

Current File
File: C:\source\newlog.log

Change File
☐ Append ☐ Overwrite
[Open File...]

Close File

OK Help
```

The dialog box lists the log file that you are currently using.

2) Click Close File.

3) Click OK.
Many of the debugger commands take C expressions as parameters. This allows the debugger to have a relatively small, yet powerful, instruction set. Because C expressions can have side effects—that is, the evaluation of some types of expressions can affect existing values—you can use the same command to display or to change a value. This reduces the number of commands in the command set.

This chapter contains basic information that helps you use C expressions as debugger command parameters.
12.1 C Expressions for Assembly Language Programmers

It is not necessary for you to be an experienced C programmer to use the debugger. However, to use the debugger’s full capabilities, you should be familiar with the rules governing C expressions. You should obtain a copy of The C Programming Language (first or second edition) by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey. This book is referred to in the C community, and in Texas Instruments documentation, as K&R.

Note:
A single value or symbol is a legal C expression.

K&R contains a complete description of C expressions; to get you started, here is a summary of the operators that you can use in expression parameters.

- **Reference operators**
  - `->` indirect structure reference
  - `.` direct structure reference
  - `[]` array reference
  - `*` indirection (unary)
  - `&` address (unary)

- **Arithmetic operators**
  - `+` addition (binary)
  - `-` subtraction (binary)
  - `*` multiplication
  - `/` division
  - `%` modulo
  - `(` type `)` type cast

- **Relational and logical operators**
  - `>` greater than
  - `>=` greater than or equal to
  - `<` less than
  - `<=` less than or equal to
  - `==` is equal to
  - `!=` is not equal to
  - `&&` logical AND
  - `||` logical OR
  - `!` logical NOT (unary)
Increment and decrement operators

`++` increment  `--` decrement

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

Bitwise operators

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

Assignment operators

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

These operators support a shorthand version of the familiar binary expressions; for example, `X = X + Y` can be written in C as `X += Y`. Because these operators affect a symbol’s final value, the parameters they are used with have side effects.
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 * ... )

  You can use up to six * characters in a cast.

Additional features

- All floating-point operations are performed in double precision using standard widening. (This is transparent.) Floats are represented in IEEE floating-point format.
- All registers can be referenced by name. The 'C6x auxiliary registers are treated as integers and/or pointers.
- Void expressions are legal (treated like integers).
- The specification of variables and functions can be qualified with context information. Local variables (including local statics) can be referenced with the expression form:
  
  function name.local name
This expression format is useful for examining the automatic variables of a function that is not currently being executed. Unless the variable is static, however, the function must be somewhere in the current call stack. If you want to see local variables from the currently executing function, you need not use this form; you can simply specify the variable name (just as in your C source).

File-scoped variables (such as statics or functions) can be referenced with the following expression form:

```
filename.function name
```
or

```
filename.variable name
```

This expression format is useful for accessing a file-scoped static variable (or function) that may share its name with variables in other files.

In this expression form, `filename` does not include the file extension; the debugger searches the object symbol table for any source filename that matches the input name, disregarding any extension. Thus, if the variable ABC is in file source.c, you can specify it as source.ABC.

These expression forms can be combined into an expression of the form:

```
filename.function name.variable name
```

Any integral or void expression can be treated as a pointer and used with the indirection operator (*). Here are several examples of valid use of a pointer in an expression:

```
*123
*A5
*(A2 + 123)
*(I*J)
```

By default, the values are treated as integers (that is, these expressions point to integer values).

Any expression can be type cast to a pointer to a specific type (overriding the default of pointing to an integer, as described above).

**Hint:** You can use casting with the WA and DISP commands to display data in a desired format.

For example, the expression:

```
*(float *)10
```

treats 10 as a pointer to a floating-point value at location 10 in memory. In this case, the debugger fetches the contents of memory location 10 and treats the contents as a floating-point value. If you use this expression as a parameter for the DISP command, the debugger displays memory contents as an array of floating-point values within the DISP window, beginning with memory location 10 as array member [0].
Note how the first expression differs from the expression:

\[(\text{float})*10\]

In this case, the debugger fetches an integer from address 10 and converts the integer to a floating-point value.

You can also type cast to user-defined types such as structures. For example, in the expression:

\[ ((\text{struct STR *} )10)\rightarrow \text{field} \]

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.
Alternate Methods for Performing Tasks

In addition to the toolbar, menu options, and commands, there are various function key methods for entering commands and other alternatives that you can use. This appendix describes these alternatives.

This appendix also describes the search mechanism that the debugger uses when looking for files.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Scrolling a Window’s Contents</td>
<td>A-2</td>
</tr>
<tr>
<td>A.2 Editing Data or Selecting the Active Field</td>
<td>A-3</td>
</tr>
<tr>
<td>A.3 Escaping From an Action</td>
<td>A-3</td>
</tr>
<tr>
<td>A.4 Where the Debugger Looks for Files</td>
<td>A-4</td>
</tr>
</tbody>
</table>
A.1 Scrolling a Window’s Contents

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

- Pressing 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 Watch window.

- Pressing 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 Watch window.

- The HOME key affects these windows as follows:
  - File window: Pressing HOME adjusts the File window’s contents so that the first line of the text file is at the top of the window.
  - Disassembly window: Pressing HOME adjusts the Disassembly window’s contents so that the current PC is shown in the window.
  - Memory window: Pressing HOME adjusts the Memory window’s contents so that the most recent starting address that you entered in the Address field of the Memory window or with the MEM command is shown in the window.

- In a File window, pressing END adjusts the window’s contents so that the last line of the file is at the bottom of the window.

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

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

- When a field is selected for editing, pressing the ← and → keys moves 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.
A.2 Editing Data or Selecting the Active Field

You can use the F9 key to make the current field (the field that the cursor is pointing to) active. This has various effects, depending on the field:

- In the File or Disassembly window, F9 sets or clears a breakpoint.
- In the Calls window, F9 displays the source to a listed function.
- In any data-display window, F9 edits the contents of the current field.

A.3 Escaping From an Action

You can use the ESC key as an end or undo key in these situations:

- To close a pulldown menu
- To undo an edit of the active field in a data-display window (pressing this key leaves the field unchanged)
- To halt the display of a long list of data in the display area of the Command window
A.4 Where the Debugger Looks for Files

You can perform all load-type commands by using menu options. However, if you choose to use the command-line equivalents to these menu options, you need to know where the debugger looks for source files.

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

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

- If you are using the FILE command, you have several options:
  - Within the operating-system environment, you can name additional directories with the D_SRC environment variable. The format for doing this is:
    
    SET D_SRC=pathname;pathname  For PCs
    setenv D_SRC "pathname;pathname"  For SPARCstations

    You can name several directories for the debugger to search.
  - When you invoke the debugger, you can use the –i option to name additional source directories for the debugger to search. The format for this option is –i pathname.

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

    You can specify only one directory at a time.

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

The debugger:

1) Reads options from the operating system’s command line.
2) Reads any information specified with the D_OPTIONS environment variable.
3) Reads information from the D_DIR and D_SRC environment variables.
4) Looks for the init.clr screen-configuration file.
   (The debugger searches for the screen-configuration file in directories named with D_DIR.)
5) Initializes the debugger screen and windows.
6) Finds the batch file (siminit.cmd) that defines your memory map by searching in directories named with D_DIR. The debugger expects this file to set up the memory map and follows these steps to look for the batch file:
   - When you invoke the debugger, it checks to see if you have used the –t debugger option. If it finds the –t option, the debugger reads and executes the specified file.
   - If you have not used the –t option, the debugger looks for the default initialization batch file. 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 files specified with D_OPTIONS or specified on the command line during invocation.
8) Determines the initial mode (auto, assembly, or mixed) and displays the appropriate windows on the screen.

At this point, the debugger is ready to process any commands that you enter.
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 listing contains both a description of the situation that causes the message and an action to take if the message indicates a problem or error.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1 Associating Sound With Error Messages</td>
<td>C-2</td>
</tr>
<tr>
<td>C.2 Alphabetical Summary of Debugger Messages</td>
<td>C-2</td>
</tr>
<tr>
<td>C.3 Additional Instructions for Expression Errors</td>
<td>C-17</td>
</tr>
</tbody>
</table>
C.1 Associating Sound With Error Messages

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

```
sound {on | off}
```

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

If you are using the debugger with Windows 95 or Windows NT, you must be sure that you have sound enabled in the control panel.

C.2 Alphabetical Summary of Debugger Messages

### ‘]’ expected

**Description** This is an expression error—it means that the parameter contained an opening bracket symbol but did not contain a closing bracket symbol.

**Action** See Section C.3, *Additional Instructions for Expression Errors*, page C-17.

### ‘)’ expected

**Description** This is an expression error—it means that the parameter contained an opening parenthesis symbol but did not contain a closing parenthesis symbol.

**Action** See Section C.3, *Additional Instructions for Expression Errors*, page C-17.

### Aborted by user

**Description** The debugger halted a long Command display listing because you pressed the `ESC` key.

**Action** None required; this is normal debugger behavior.
**Breakpoint already exists at address**

*Description:* During single-step execution, the debugger attempted to set a breakpoint where one already existed. (This is not necessarily a breakpoint that you set—it may have been an internal breakpoint that the debugger set for single-stepping).

*Action:* None should be required; you may want to reset the program entry point (Target→Restart) and reenter the single-step command.

**Breakpoint table full**

*Description:* 200 breakpoints are already set, and there was an attempt to set another. The maximum limit of 200 breakpoints includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.

*Action:* Open the Breakpoint Control dialog box by selecting Breakpoints from the Setup menu. Delete individual software breakpoints.

**Cannot allocate host memory**

*Description:* This is a fatal error—it means that the debugger is running out of memory.

*Action:* You can invoke the debugger with the –v option so that fewer symbols may be loaded, or you can relink your program and link in fewer modules at a time.

**Cannot allocate system memory**

*Description:* This is a fatal error—it means that the debugger is running out of memory.

*Action:* You can invoke the debugger with the –v option so that fewer symbols may be loaded, or you can relink your program and link in fewer modules at a time.
Cannot 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 cannot find the init.cmd file.
Action: Be sure that init.cmd is in the appropriate directory. If it is not, copy it from the debugger product diskette. If the file is already in the correct directory, verify that the D_DIR environment variable is set up to identify the directory. See the setting up the debugger environment information in your getting started guide.

Cannot halt the processor
Description: This is a fatal error—for some reason, pressing \( \text{ESC} \) did not halt program execution.
Action: Exit the debugger. Invoke the autoexec.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 cannot find the screen-customization file that you specified. The debugger also displays this message when you try to load a screen-customization file that was saved by an older version of the debugger.
Action: Be sure that the filename was typed correctly. If it was not, reenter the command with the correct name. If it was, reenter the command and specify full path information with the filename.
Be sure that the screen-customization file was saved with the current version of the debugger.
Alphabetical Summary of Debugger Messages

**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 are loading an actual object file. Be sure that the file was linked. You may want to run cl6x (with the –z option) or lnk6x 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 is not possible.
*Action* Close any unnecessary windows. Windows that can be closed include Watch, File, Calls, and Memory windows. To close any of these windows, make the desired window active and press **CONTROL** F4.

**Cannot read processor status**
*Description* This is a fatal error—for some reason, pressing **ESC** did not halt program execution.
*Action* Exit the debugger. Invoke the autoexec.bat file, then invoke the debugger again.

**Cannot reset the processor**
*Description* This is a fatal error—for some reason, pressing **ESC** did not halt program execution.
*Action* Exit the debugger. Invoke the autoexec.bat file, then invoke the debugger again.

**Cannot restart processor**
*Description* The debugger attempted to reset the PC to the program entry point, but the program does not seem to have an entry point.
*Action* Do not use Target→Restart or RESTART if your program does not have an explicit entry point.
Alphabetical Summary of Debugger Messages

**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*  

**Command “cmd” not found**

*Description*  
The debugger did not recognize the command that you typed.

*Action*  
Reenter the correct command. Refer to Chapter 10, *Summary of Commands*.

**Conflicting map range**

*Description*  
A block of memory specified with the Memory→Mapping menu option or the MA command overlaps an existing memory map entry. Blocks cannot overlap.

*Action*  
Use Memory→Mapping or 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 Memory Map Control dialog box or with the MD command. Use the Memory Map Control dialog box or the MA command to redefine the block of memory. If the existing block is necessary, use the Memory Map Control dialog box or the MA command to define a range that will not overlap the existing block.
Alphabetical Summary of Debugger Messages

Corrupt call stack

*Description*  
The debugger tried to update the Calls window and could not. This message is displayed in the following situations:

- A function was called that did not return.
- The program stack was overwritten in target memory.
- You are debugging code that has optimization enabled (for example, you did not use the –g compile option); if this is the case, ignore this message—code execution is not affected.

*Action*  
If your program called a function that did not return, then this is normal behavior (as long as you intended for the function not to return). Otherwise, you may be overwriting program memory.

Error in expression

*Description*  
This is an expression error.

*Action*  
See Section C.3, Additional Instructions for Expression Errors, page C-17.

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 exceeded the maximum loadable COFF file size.

Action: Try loading the file without the symbol table (SLOAD), or use cl6x (with the --z option) or 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 C.3, Additional Instructions for Expression Errors, page C-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.

Illegal cast

Description: This is an expression error—the expression parameter uses a cast that does not meet the C language rules for casts.

Action: See Section C.3, Additional Instructions for Expression Errors, page C-17.
Alphabetical Summary of Debugger Messages

Illegal left hand side of assignment
Description This is an expression error—the lefthand side of an assignment expression does not meet C language assignment rules.
Action See Section C.3, Additional Instructions for Expression Errors, page C-17.

Illegal memory access
Description Your program tried to access unmapped memory.
Action Modify your source code. Alternatively, you can check and modify your memory map.

Illegal operand of &
Description This is an expression error—the expression attempts to take the address of an item that does not have an address.
Action See Section C.3, Additional Instructions for Expression Errors, page C-17.

Illegal pointer math
Description This is an expression error—some types of pointer math are not valid in C expressions.
Action See Section C.3, Additional Instructions for Expression Errors, page C-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 C.3, Additional Instructions for Expression Errors, page C-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 C.3, Additional Instructions for Expression Errors, page C-17.
Alphabetical Summary of Debugger Messages

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 C.3, Additional Instructions for Expression Errors, page C-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 C.3, Additional Instructions for Expression Errors, page C-17.

Integer not allowed
Description This is an expression error—the command does not accept an integer as a parameter.
Action See Section C.3, Additional Instructions for Expression Errors, page C-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 10, Summary of Commands.
Invalid memory attribute

**Description**  The third parameter of the MA command specifies the type, or attribute, of the block of memory that is added to the memory map. The parameter entered did not match one of the valid attributes.

**Action**  Reenter the MA command. Use one of the following valid parameters to identify the memory type:

- R, ROM (read-only memory)
- W, WOM (write-only memory)
- R|W, RAM (read/write memory)
- 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 File→Load Program, File→Reload Program, File→Load Symbols, the LOAD, the SLOAD, or the RELOAD command is not an object file that the debugger can load, or it has been corrupted.

**Action**  Be sure that you are loading an actual object file. Be sure that the file was linked. You may want to run cl6x (with the –z option) or lnk6x again to create an executable object file. If the file you attempted to load was a valid executable object file, then it was probably corrupted; recompile, assemble, and link with cl6x.

Invalid watch delete

**Description**  The debugger cannot delete the parameter supplied with the WD command.

**Action**  Reenter the WD command. Be sure to specify the symbol name that matches the item you want to delete.

Invalid window position

**Description**  The debugger cannot move the window to the XY position specified with the MOVE command. Either the XY parameters are not within the screen limits, or the active window may be too large to move to the desired position.

**Action**  Reenter the MOVE command. Enter the X and Y parameters in pixels.
Invalid window size
Description The width and length specified with the SIZE or MOVE command may be too large or too small. If valid width and length were specified, then the active window is already at the far right or bottom of the screen and so cannot be made larger.
Action Reenter the SIZE command. Enter the width and length in pixels.

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 C.3, Additional Instructions for Expression Errors, page C-17.

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.

More than 4 reads to register register at cycle cnum
Description Your program issued more than four reads of the same register in the same cycle, which is illegal. However, conditional registers are not included in this count. For more information about register read constraints, see the TMS320C62xx CPU and Instruction Set Reference Guide.
Action Modify your source code.
**Move path conflicts at cycle cycle**

*Description*  Your program issued two instructions that used the same functional unit in the same execute packet. For more information about functional units and resource constraints, see the *TMS320C62xx CPU and Instruction Set Reference Guide.*

*Action*  Modify your source code.

**Multiple writes to register register at cycle cnum**

*Description*  Your program issued multiple writes to the same register in the same cycle. This problem occurs due to the latency associated with previous instructions in the pipeline. For more information, see the *TMS320C62xx CPU and Instruction Set Reference Guide.*

*Action*  Modify your source code.

**Name “name” not found**

*Description*  The command cannot find the object named *name*.

*Action*  If *name* is a symbol, be sure that it was typed correctly. If it was not, reenter the command with the correct name. If it was, then be sure that the associated object file is loaded.

**Pointer not allowed**

*Description*  This is an expression error.

Alphabetical Summary of Debugger Messages

R

Read conflicts with long operand at cycle cnun
Description Your program attempted to write more than one long result in
a single cycle on each side of the register file. Because the .L
and .S units share their long read port with the store port, op-
erations that read a long value cannot be issued on the .L and/
or .S units in the same execute packet as a store. For more
information about long path conflicts, see the TMS320C62xx
CPU and Instruction Set Reference Guide.
Action Modify your source code.

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 C.3, Additional Instructions for Expression Errors,
page C-17.

Structure member name required
Description This is an expression error—a symbol name is followed by a
period but no member name.
Action See Section C.3, Additional Instructions for Expression Errors,
page C-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 C.3, Additional Instructions for Expression Errors,
page C-17.
Take file stack too deep
Description Batch files can be nested up to ten levels deep. The batch file that you tried to execute with File→Take or the TAKE command calls batch files that are nested more than ten levels deep.
Action Edit the batch file that caused the error. Instead of calling another batch file from within the offending file, you can 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. The maximum limit of 200 breakpoints includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.
Action Open the Breakpoint Control dialog box by selecting Breakpoints from the Setup menu. Delete individual software breakpoints.

Too many paths
Description More than 20 paths have been specified cumulatively with the USE command, D_SRC environment variable, and –i debugger option.
Action Do not enter the USE command before entering another command that has a filename parameter. Instead, enter the second command and specify full path information for the filename.

Undeclared port address
Description You attempted to do a connect/disconnect on an address that is not declared as a port.
Action Verify the address of the port to be connected or disconnected.
Alphabetical Summary of Debugger Messages

User halt

*Description*  The debugger halted program execution because you clicked the Halt icon on the toolbar, you selected Halt! from the Target menu, or you pressed the **ESCAPE** key.

*Action*  None required; this is normal debugger behavior.

Window not found

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

*Action*  Reenter the WIN command. Here are the valid window names; the bold letters show the smallest acceptable abbreviations:

- Calls
- CPU
- Command
- Disassembly
- Memory
- Profile
- Watch

Write conflicts with long writes at cycle *cnun*

*Description*  Your program attempted to write more than one long result in a single cycle on each side of the register file. Because the .S and .L units share a read register port for long source operands and a write register port for long results, only one of these operations can be issued per side in an execute packet. For more information about long path conflicts, see the *TMS320C62xx CPU and Instruction Set Reference Guide*.

*Action*  Modify your source code.

Write not allowed for port

*Description*  You attempted to connect a file for output operation to an address that is not configured for write.

*Action*  Either change the software to write a port that is configured for write, or change the attributes of the port.
C.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.
active window: The window that is currently selected for moving, sizing, editing, closing, or some other function.

aggregate type: A C data type, such as a structure or array, in which a variable is composed of multiple variables, called members.

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

ANSI C: A version of the C programming language that conforms to the C standards defined by the American National Standards Institute.

assembly mode: A debugging mode that shows assembly language code in the Disassembly window and does not show the File window, no matter what type of code is currently running.

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

auto mode: A context-sensitive debugging mode that automatically switches between showing assembly language code in the Disassembly window and C code in the File window, depending on what type of code is currently running.

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 does not execute debugger batch files, and the debugger does not execute PC batch files.

benchmarking: A type of program execution that allows you to track the number of CPU cycles consumed by a specific section of code.

breakpoint: A point within your program where execution will halt because of a previous request from you.
**Glossary**

**C**

**Calls window:** A window that lists the functions called by your program.

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

**cl6x:** A shell utility that invokes the 'C6x compiler, assembler, and linker to create an executable object file version of your program.

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

**code-display windows:** Windows that show code, text files, or code-specific information. This category includes the Disassembly, File, and Calls windows.

**COFF:** Common Object File Format. An implementation of the object file format of the same name developed by AT&T. This format promotes module programming by supporting the concept of sections. The 'C6x compiler, assembler, and linker use and generate COFF files.

**command line:** The portion of the Command window where you can enter commands.

**Command window:** A window that provides an area for you to enter commands and for the debugger to echo command entry, show command output, and list progress or error messages.

**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.

**cursor:** An icon on the screen (such as an arrow or a horizontal line) that is used as a pointing device. The cursor is usually under mouse or keyboard control.

**D**

**D_DIR:** An environment variable that identifies the directory containing the commands and files necessary for running the debugger.

**D_OPTIONS:** An environment variable that you can use for identifying often-used debugger options.

**D_SRC:** An environment variable that identifies directories containing program source files.
data-display windows: Windows for observing and modifying various types of data. This category includes the Memory, CPU, and Watch windows.

debugger: A window-oriented software interface that helps you to debug '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 (reverse assembly) of memory contents.

display area: The portion of the Command window where the debugger echoes command entry, shows command output, and lists progress or error messages.

dock (a window): To anchor a floating window to an outer edge of the debugger application window. A docked window has no title bar and cannot be moved. However, a docked window can be resized.

drag: To move an object on the debugger display by pressing one of the mouse buttons and moving the mouse.


environment variable: A special system symbol that the debugger uses for finding directories or obtaining debugger options.

File window: A window that displays the contents of the current C code. The File window is intended primarily for displaying C code but can be used to display any text file.

float (a window): To cause a debugger window to sit on top of the debugger application window outside the edges of the debugger application window. A floating window always appears active.

init.cmd: A batch file that contains debugger-initialization commands. If this file is not present when you first invoke the debugger, then all memory is invalid.

**Glossary**

**M**

**memory map:** A map of memory space that tells the debugger which areas of memory can and cannot be accessed.

**Memory window:** A window that displays the contents of memory.

**menu bar:** A row of pulldown menu selections found at the top of the debugger display.

**mixed mode:** A debugging mode that simultaneously shows both assembly language code in the Disassembly window and C code in the File window.

**P**

**PC:** Personal computer or program counter, depending on the context and where it is used in this book: 1) In installation instructions or information relating to hardware and boards, PC means *personal computer*. 2) In general debugger and program-related information, PC means *program counter*, which is the register that identifies the current statement in your program.

**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.

**scroll bar:** A bar on the right side or bottom of a window that allows you to adjust the contents of the window to display hidden information.

**scroll bar handle:** The rectangular box in the center of the right scroll bar in the Disassembly or Memory window that marks the center of disassembled code or memory contents.

**scrolling:** A method of moving the contents of a window up, down, left, or right to view contents that were not originally shown.

**section:** A relocatable block of code or data that will ultimately occupy contiguous space in the memory map.
**side effects:** A feature of C expressions in which using an assignment operator in an expression affects the value of one of the components used in the expression.

**simulator:** A development tool that simulates the operation of the `C6x and lets you execute and debug applications programs by using the C source debugger.

**single-step:** A form of program execution that allows you to see the effects of each statement. The program is executed statement by statement; the debugger pauses after each statement to update the data-display windows.

**status bar:** An area at the bottom of the debugger application window that displays context-sensitive help and the status of the processor.

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

**toolbar:** A set of icons in the debugger application window that allows you to execute a command by clicking an icon.

**Watch window:** A window that displays the values of selected expressions, symbols, addresses, and registers.

**window:** A defined rectangular area of space on the display.
Index

? command
- description 3-3 to 10-9
- display formats 8-22 to 10-9
- examining register contents 7-3
- modifying PC 8-4 to 8-5

$SIM$$ 11-3

A

absolute addresses 7-15 to 8-7
active window
- definition D-1
- description 2-21 to 2-22
- identifying 2-21
- making a window active 2-21 to 2-22
ADDR command
- description 6-10 to 10-10
- finding current PC 7-2
addresses
- absolute addresses 7-15 to 8-7
- accessible locations 5-2 to 5-3
- contents of (indirection) 8-8 to 8-18
- hexadecimal notation 8-1
- I/O address space 5-12 to 5-13
- in Memory window 2-14 to 2-15
- invalid memory 5-3
- nonexistent memory locations 5-3
- protected areas 5-3
- symbolic addresses 8-7
- undefined areas 5-3
aggregate types
- definition D-1
- displaying 8-17 to 8-19
ALIAS command 10-11
Alias Control dialog box 4-10
alising
- ALIAS command 10-11
- defining an alias 4-11
- definition D-1
- deleting an alias 4-12
- description 4-10 to 4-12
- editing an alias 4-12
- limitations 4-12
- redefining an alias 4-12
- supplying parameters 4-11
alternate methods for performing tasks A-1 to A-4
ANSI C, definition D-1
arithmetic operators 12-2
arrays
- displaying 8-17 to 8-19
- member operators 12-2
arrow keys, scrolling A-2
- as shell option 1-6, 9-3
ASM command 10-11
assembler 1-5
assembly language code
- displaying 2-2 to 6-4 to 6-7
- displaying source code 2-9 to 6-7
assembly mode
- ASM command 10-11
- definition D-1
- description 2-2 to 5-2
- selecting 6-3
assembly optimizer 1-5
assignment operators 8-4 to 8-5
assistance from TI vii
auto mode
- C command 10-13
- definition D-1
- description 2-2 to 5-2
- selecting 6-3
autoexec.bat file, definition D-1
BA command 10-12

batch files
  controlling command execution
    conditional commands 10-22, 11-3
    looping commands 10-28, 11-3 to 11-4
  creating 11-2 to 11-5
  definition D-1
  displaying 8-9
  displaying text when executing 10-18, 11-2
  executing 10-43, 11-6
  halting execution 11-6
init.clr 10-36, B-1
init.cmd
  definition D-3
  during invocation 5-2, B-1
initialization
  init.cmd 2-2, B-1
  sample memory map 5-4
mem.map 5-11
memory maps 5-10
pausing 10-30, 11-5
sample file 11-2
siminit.cmd 8-1
TAKE command 10-43
BD command 10-12

benchmarking
  CLK pseudoregister 7-11
  constraints 7-11
  definition D-1
  description 7-11
big-endian format, selecting (–me option) 1-8
bitwise operators 12-3
BL command 10-12
BR command 10-13
Breakpoint Control dialog box 7-15
breakpoints (software)
  adding 7-14 to 7-15
  benchmarking with RUNB 7-11
  clearing 7-16
  command summary 10-5
  definition D-1
  description 7-13 to 7-18
  listing set breakpoints 7-15, 10-12
  loading breakpoint settings 7-18

breakpoints (software) (continued)
  restrictions 7-13
  saving set breakpoints 7-17
  setting 7-14 to 7-15, A-3
  setting profile stopping points 9-15 to 9-16
Breakpoints toolbar icon
  clearing a breakpoint 7-16
  loading breakpoint settings 7-18
  saving breakpoint settings 7-17
  setting a breakpoint 7-15
.bss section, clearing 1-8

C command 10-13
C compiler 1-5
–c debugger option 1-8
C expressions 8-4 to 8-5, 12-1 to 12-6
See also expressions
C source
  displaying 2-2, 2-4, 10-19
  managing memory data 8-8
C source debugger. See debugger
C Step toolbar icon 7-8
CALLS command
  description 10-13
  effect on debugging modes 2-5
Calls window
  definition D-2
  description 2-12
  displaying code for a function 6-9
casting
  definition D-2
  description 8-8, 12-4
changing the prompt 3-12
CHDIR (CD) command 10-14, A-4
c16x shell, definition D-2
clearing software breakpoints
  debugger only 7-16
  profiler only 9-16
clearing the .bss section 1-8
clearing the display area
  clicking, definition D-2
CLK pseudoregister
  definition 7-11
  restrictions in C code 7-11
Index

Index-3
customizing the display
changing the prompt 3-12 10-32
closing a window 3-6
description 3-1 to 3-16
displaying the status bar 3-7
displaying the toolbar 3-8 to 3-11
docking a window 3-4 to 3-5
floating a window 3-2 to 3-3
hiding the toolbar 3-11
loading a custom display 3-14 to 3-15 10-36
moving the toolbar 3-8 to 3-9
saving a custom display 3-13 10-40
undocking the toolbar 3-10 to 3-11

debugger
commands
alphabetical summary 10-9 to 10-48
functional summary 10-2 to 10-8
profiling 10-49 to 10-52
definition D-3
description 1-2 to 1-3
display
description 2-1 to 2-22
illustration 7-2
invocation
description 1-7 to 1-8
options 1-7 to 1-9
task ordering 8-1
key features 1-3
messages C-1 to C-17
debugging modes
assembly mode 2-2 6-2
auto mode 2-2 6-2
command summary 10-3
default mode 2-2 6-2
description 2-2 to 2-5 6-2 to 6-3
mixed mode 2-4 5-2
restrictions 2-5
selecting 6-3
decrement operator 12-3
default
data formats 8-20 to 8-22
debugging mode 2-2 6-2
display 2-2 3-15 6-2
memory map 5-4
screen configuration file 3-13
defining a memory map. See memory mapping
defining an alias 4-11
defining areas for profiling
description 9-5 to 9-12
disabling areas 9-10 to 9-12 10-49 to 10-50
enabling areas 9-11 to 9-12 10-50
marking areas 9-5 to 9-9 10-49
restrictions 9-12
unmarking areas 9-12 10-51
deleting watched values 8-19
determining type of a variable 8-2
developing code 1-4 to 1-5
dialog boxes, accessing online help 1-12
DIR command 10-16

data-display windows
CPU window 2-17 8-12 to 8-16
description D-3
definition 2-6
Memory window 2-14 to 2-16 8-6 to 8-11
overview 8-2
Watch window 2-18 to 2-19 8-17 to 8-19
data formats 8-20
data-management commands
controlling data format 8-8 8-20 to 8-22
side effects 8-4 to 8-5
summary 10-3
data types 8-21

Index-4
directories
  changing current directory 10-14
  identifying additional source directories 10-44
    See also –i debugger option
  identifying current directory A-4
  listing contents of current directory 10-16
  relative pathnames 10-14
  search algorithm 11-6 A-4 B-1

disabling areas for profiling 9-10 to 9-12

disabling memory mapping 5-8 to 5-9

disassembly
  definition D-3
  description 6-6
  displaying 6-8 to 6-7

Disassembly window
  Address field 6-6
  context menu 2-11
  definition D-3
  description 6-8 to 6-11
  modifying the display 10-16
  running code to a specific point 7-5
  scrolling through the contents 6-7
  setting a breakpoint 7-14
  setting current PC 7-2
  viewing disassembly 6-8 to 6-7

DISP command
  description 10-17
  display formats 8-22 10-17
  effect on debugging modes 2-5

display area
  clearing 4-9 10-14
  definition D-3
  description 6-2 4-6
  recording information from 10-18 11-7 to 11-8

display-customization commands 10-6

display formats
  ? command 8-22 10-9
  data types 8-21
  description 8-20 to 8-22
  DISP command 8-22 10-17
  MEM command 8-22 10-27
  resetting types 8-21
  SETF command 8-20 to 8-22 10-37
  WA command 8-22 10-46

displaying
  assembly language code 6-4
  assembly language source 2-9 6-7
  batch files 5-9

displaying (continued)
  C code 6-8 to 6-10
  C function 6-9
  code 6-1 to 6-10
  code at a specific point 6-10
  data in nondefault formats 8-20 to 8-22
    See also display formats
  debugger on a different machine (–d option) 1-8
  disassembly 5-8 to 5-9
  status bar 3-7
  text files 6-9
  text when executing a batch file 10-18 11-2
  toolbar 3-8 to 3-11
  watched values 8-17 to 8-18

DLOG command 10-18

docking a window
  definition D-3
  description 6-8 to 6-11

double-precision floating-point registers 8-16

dragging, definition D-3

E

E command 10-19
  See also EVAL command

ECHO command 10-18 11-2

edit key (F9) A-3

editing
  command line 4-7 to 4-8
  data values 8-4 A-3
  expression side effects 8-4
  function key method A-3
  overwrite method 8-4

EISA, definition D-3

ELSE command 10-22 11-3
  See also IF/ELSE/ENDIF commands

enabling areas for profiling 9-11 to 9-12

enabling memory mapping 5-8 to 5-9

dragging, definition D-3

end key, scrolling A-2

ENDIF command 10-22 11-3
  See also IF/ELSE/ENDIF commands

ENDLOOP command 10-23 11-3 to 11-4
  See also LOOP/ENDLOOP commands

entering commands
  from menus 4-2 to 4-5
  from the toolbar 4-3
  on the command line 4-6 to 4-9
Index

entering profiling environment
  menu option 5-4
    –profile option 1-8
entry point (of program) 7-2
environment variables
  definition D-3
    effects on debugger invocation B-1
error messages
  beeping 10-39, C-2
    description C-1 to C-17
escape key
  closing a menu A-3
  halting a listing in the Command window A-3
  halting execution 7-12
  undoing an edit A-3
escaping from an action (function key method) A-3
EVAL command
  description 8-3, 10-19
  modifying PC 7-3
  side effects 8-4 to 8-5
  evaluating an expression 8-3
executing code. See run commands; running programs; single-step execution
executing commands 4-1 to 4-14
exiting the debugger 1-10, 10-33
expressions
  addresses 8-7
  analysis 12-4 to 12-6
    description 12-1 to 12-6
  evaluation
    with ? command 3-3, 10-8
    with DISP command 10-17
    with EVAL command 3-3, 10-19
    with LOOP command 10-23, 11-3
  operators 12-2 to 12-3
  restrictions 12-4
  void expressions 12-4
  with expressions that have side effects 8-4 to 8-5

F

F1 key, accessing online help 1-12
F2 key, using command history 4-9
F3 key, selecting the debugging mode 6-3
F5 key
  running a profiling session 9-17
  running code 7-4
F8 key, single-stepping 7-8
F9 key
  changing the File window display 6-9
  clearing a breakpoint A-3
  displaying a function A-3
  editing data A-3
  setting a breakpoint 2-8, 2-10, A-3
F10 key, single-stepping over function calls 7-9
FILE command
  description 10-19
  effect on debugging modes 2-5
File menu
  Load Config option 3-14
  Load Program option 6-4, 8-10
  Load Symbols option 8-5
  Log File option 11-7
  Open option 6-8
  Reload Program option 6-5
  Save As Config option 3-13
  Take option 11-6
File window
  context menu 2-9
    definition D-3
    description 2-8 to 2-9, 6-8 to 6-10
  displaying assembly language source 2-9, 6-7
  running code to a specific point 7-14
  setting a breakpoint 7-5
  setting current PC 7-2
file/load commands 10-5
files
  connecting to I/O ports 5-12 to 5-13, 10-25
  disconnecting from I/O ports 5-13, 10-27
  log files 11-7, 11-8
  saving memory to a file 8-9 to 8-10, 10-29
FILL command 10-20
Fill Memory dialog box 8-10, 8-11
FILLB command 10-20
floating a window
  definition D-3
    description 3-2 to 3-3
  floating-point double-precision registers 8-16
  floating-point operations 12-4
  floating-point single-precision registers 8-15
full profile 9-17
FUNC command
  description 6-9, 10-20
  effect on debugging modes 2-5
Index

function calls
  displaying functions 10-20
  executing function only 10-34
  in expressions 8-4, 12-4
  stepping over 10-15, 10-30
  tracking in Calls window 2-12, 6-9, 10-13

G
  –g assembler option, displaying assembly language source 2-3, 6-7
  –g shell option 1-6, 9-3
  GO command 7-5, 10-21
  grouping/reference operators 12-2

H
  HALT command 10-21
  Halt toolbar icon 7-12
  halting
    batch file execution 11-6
    debugger 1-10, 10-33
    program execution 7-1, 1-10, 7-12, 10-33
  help, accessing 1-12 to 1-13
  HELP command 1-12 to 1-13, 10-21
  Help menu
    Help Topics option 4-4
  Help toolbar icon 1-12
  Help Topics toolbar icon 1-12
  hexadecimal notation
    addresses 8-7
    data formats 8-20
  hiding the toolbar 3-11
  history of commands 4-9
  home key
    scrolling A-2
    scrolling with A-2
  History of commands 4-9
  HOME key 1-12
  home key A-2
  –i debugger option 1-8, A-4

I
  interpreting profile data 9-25
  invalid memory addresses 5-25, 5-9
  invoking
    custom displays 3-15
    debugger 1-7 to 1-9
  ISA, definition D-3

K
  key sequences
    displaying functions A-3
    halting actions A-3
    making a window active 2-22
    scrolling A-2
    setting/clearing software breakpoints A-3
  used in editing
    command line A-7 to A-8
    data values A-3

L
  labels for data in Watch window 2-18 to 2-19
limits
breakpoints 7-13
customized prompt length 3-12 10-32
paths A-4
linker 1-5
Load Breakpoint File dialog box 7-18
LOAD command 10-22
Load Program dialog box 6-4
load/file commands 10-5
loading
assembly language code 6-4 to 6-7
batch files 11-6
COFF files, restrictions 5-3
custom displays
description 3-14 to 3-15
restrictions 3-15
object code
after invoking the debugger 5-4
description 5-4 to 6-7
symbol table only 1-9 5-3 10-35
while invoking the debugger 1-7 6-4
with symbol table 6-4
without symbol table 1-9 6-5 10-33
saved breakpoint settings 7-18
Log File dialog box
closing a file 11-8
opening a file 11-7
log files 11-7 to 11-8
logical operators
conditional execution 7-10
description 12-2
LOOP/ENDLOOP commands
conditions 10-23 11-4
description 10-23 11-3 to 11-4
looping commands 10-23 11-3 to 11-4

M
MA command 5-4 5-10 10-24
managing data
basic commands 8-2 to 8-3
changing data values 8-4 to 8-5
in memory 8-6 to 8-11
in registers 8-12 to 8-16
in Watch windows 8-17 to 8-19
managing windows 3-2
MAP command 10-25
mapping. See memory mapping
marking areas 9-5 to 9-9
See also unmarking areas
MC command 5-12 to 5-13 10-25
MD command 10-26
–me debugger option 1-8
MEM command
description 2-15 8-7 10-27
display formats 8-22 10-27
effect on debugging modes 2-5
memory
See also memory mapping
batch file search order 5-2 8-1
command summary 10-6
data formats 8-20 to 8-22
displaying in different numeric format 8-8
filling
byte by byte 1-1 10-20
word by word 8-10 to 8-11 10-20
invalid addresses 5-3
invalid locations 5-9
nonexistent locations 5-2
protected areas 5-3 5-9
saving 5-9 to 5-10 10-29
simulating I/O memory 5-12 to 5-13 10-25
10-27
simulating ports
MC command 5-12 to 5-13 10-25
MI command 5-13 10-27
undefined areas 5-3 5-9
valid types 5-6 5-10
Memory Map Control dialog box 5-5
memory mapping
See also memory
adding ranges 5-3 to 5-6 5-10 10-24
command summary 10-6
creating a map 5-5 to 5-7
default map 5-4
defining a map 5-2 to 5-3
defining a map in a batch file 5-10
definition (memory map) 5-4
deleting ranges 5-6 10-26
description 5-1 to 5-13
disabling 5-8 to 5-9
enabling 5-8 to 5-9
listing current map 5-5
modifying a map 5-2 5-5 5-7
potential problems 5-3
resetting 10-29

Index
memory mapping (continued)
restrictions 5-6
returning to default 5-11
sample map 5-4

Memory menu
Fill Byte option 8-11
Fill Word option 8-10
Mapping option 5-5
overview 4-3
Save option 8-9

Memory window
additional Memory windows 2-15 to 2-16
address columns 2-14
Address field 8-6
changing the range displayed 2-15
context menu 2-16
data columns 2-14
definition D-4
description 2-14 to 2-16
displaying memory contents 8-6 to 8-11
editing memory contents 8-4
modifying display 10-27
opening additional windows 2-15, 8-7
scrolling through the contents 8-6

menu bar
conventions 4-3
definition D-4
description 4-2 to 4-5

menu options
context menus 2-7
definition (pull-down menu) D-4
description 4-2 to 4-5
menu conventions 4-5
selecting 4-5
messages C-1 to C-17
–mg shell option 1-6
MI command 5-13 10-27
MIX command 10-28
mixed mode
definition D-4
description 2-4 6-2
MIX command 10-28
selecting 6-3
ML command 10-28
mode commands 10-3
Mode menu
overview 4-4
selecting the debugging mode 6-3

modes. See debugging modes
modifying
command line 4-7 to 4-8
command-line prompt 3-12
current directory 10-14
data values 8-4
memory map 5-2
5-5
mouse icon 9-6
MOVE command 10-28
moving a window 10-28
moving the toolbar 3-8 to 3-9
MR command 10-29
MS command 10-29

natural format 12-5
Next C toolbar icon 7-9
NEXT command 7-9 10-30
Next toolbar icon 7-9
nonexistent memory locations 5-2
notational conventions 3 to 4

O
–o shell option 1-6
object files
creating 6-4
loading
after invoking the debugger 5-4
LOAD command 10-22
symbol table only 1-9 10-38
while invoking the debugger 1-7 6-5
with symbol table 5-4
without symbol table 1-9 5-5 10-33

online help, accessing 1-12 to 1-13
Open Configuration File dialog box 3-14
Open File dialog box 5-8
Open Take File dialog box 11-6
Open toolbar icon 3-6 6-8
opening a window 3-6
operating system
entering commands from the debugger 4-13 to 4-14 10-42
exiting from system shell 10-42
Index

profiling (continued)
resuming a session 9-19 10-32
running a session
description 9-17 to 9-19
full 9-17 to 9-18 10-31
quick 9-17 to 9-18 10-31
saving statistics
all views 9-27 10-44
current view 9-28 10-44
stopping points
adding 9-15 10-36
deleting 9-16 10-36 10-39
description 9-15 to 9-16
listing 10-38
resetting 9-16 10-39
strategy 9-3
viewing data
associated code 9-25 to 9-26
description 9-20 to 9-26
displaying areas 9-24 to 9-25 10-52
displaying different data 9-21 to 9-22 10-52
sorting data 9-23 10-52
program
debugging 1-11
entry point
finding 7-3
resetting 10-34
halting execution 1-10 7-12 10-33
preparation for debugging 1-6
running 7-4
PROMPT command 3-12 10-32
pseudoregisters 8-15 to 8-16
pulldown menus
See also menu options
definition D-4

Q
quick profile 9-17
QUIT command 1-10 10-33

R
recording Command window displays 10-18
reentering commands 4-9
reference/grouping operators 12-2
registers
CLK pseudoregister 7-11
displaying/modifying 6-12 to 8-16
pseudoregisters 8-15 to 8-16
referencing by name 12-4
reordering in the CPU window 2-17 8-13
related documentation 1 to 2
relational operators
conditional execution
description 12-2
relative pathnames 10-14 A-4
RELOAD command 6-5 10-33
RESET command 10-33
resetting
memory map 10-29
program entry point 10-34
target system 7-6 10-33
RESTART (REST) command 10-34
Restart toolbar icon 7-2
restrictions
See also limits
breakpoints 7-13
C expressions 12-4
debugging modes 2-5
loading a custom display 3-15
memory mapping 5-6
profiling environment 9-4
RETURN (RET) command 10-34
Return toolbar icon 7-6
RUN command 7-4 10-35
run commands 10-7
Run toolbar icon 7-4
RUNB command 7-11 10-35
running programs
conditionally 7-10
continuous run 7-6
description 7-1 to 7-18
halting execution 7-12
program entry point 7-2 to 7-3
running code in a C function 7-6
running entire program 7-4
through breakpoints up to a specific point 7-5

S
–s debugger option 1-9 6-5
Index

SA command 10-36
Save Breakpoint File dialog box 7-17
Save Configuration File dialog box 3-13
Save Memory to COFF File dialog box 8-9
Save Profile File dialog box 9-28
Save Profile View File dialog box 9-27
saving a custom display 3-13
saving breakpoint settings 7-17
saving profile data 9-27 to 9-28
scalar type, definition D-4
SCONFIG command 3-15 10-36
screen-customization commands 10-6
scroll bar, definition D-4
scroll bar handle definition 8-7 8-6
description A-2
scrolling definition D-4
function key method A-2
SD command 10-36
section, definition D-4
selecting a debugging mode 5-3
selecting big-endian format (–me option) 1-8
serial ports, connecting an I/O port 5-12 to 5-13
SETF command 8-20 to 8-22 10-37
setting a software breakpoint 7-14 to 7-15 9-15
Setup menu
  Alias Commands option 4-10
  Breakpoints option
    clearing a breakpoint 7-16
    loading breakpoint settings 7-16
    saving breakpoint settings 7-17
    setting a breakpoint 7-15
    overview 4-3
    Watch Variable option 2-18 8-14 8-17
side effects
  definition D-5
  description 8-4 to 8-5 12-3
valid operators 8-5
$$SIM$$ constant 11-3
sim6x command, options 1-7
simulator
  definition D-5
  I/O memory 5-12 to 5-13 10-25 10-27
  invoking the debugger 1-7 to 1-9
  resetting 7-6
  $$SIM$$ constant 11-3
single-precision floating-point registers 8-15
single-step commands
  CNEX command 7-9 10-15
  CSTEP command 7-8 10-15
  NEXT command 7-9 10-30
  STEP command 7-8 10-41
single-step execution
  assembly language code 7-7 to 7-8 10-41
  C code 7-8 10-15
  continuous step D-5
  description 7-7 to 7-9
  over function calls 7-9 10-15 10-30
  stepping over breakpoints 7-9
SIZE command 10-38
sizeof operator 12-4
sizing a window
  description 10-38
  while moving it 10-28
SL command 10-38
SLOAD command
  description 10-38
  –s debugger option 1-9
software breakpoints. See breakpoints (software)
sorting profile data 9-23
SOUND command 10-39
SR command 10-39
SSAVE command 10-40
starting point for program execution 7-2 to 7-3
status bar
  definition D-5
  description 3-7
STEP command 7-8 10-41
Step toolbar icon 7-8
stopping points
  adding 9-15 10-36
  deleting 9-16 10-36 10-39
  listing 10-38
  resetting 9-16 10-39
structures
  direct reference operator 12-2
  indirect reference operator 12-2
symbol table definition D-5
loading without object code 1-9 6-5 10-38
symbolic addresses 8-7
SYSTEM command 4-13 to 4-14 10-42
system commands
  entering from command line 4-13
  entering several from system shell 4-14
  summary 10-4
system shells 4-13 to 4-14

T

–t debugger option
  defining a memory map 5-10
description 1-9
  during debugger invocation 5-2 8-1
TAKE command
  defining a memory map 5-10
description 10-43
  returning to the original memory map 5-11
Target menu
  C Step option 7-8
  Continuous Run option 7-6
  Halt! option 7-12
  Next C option 7-9
  Next option 7-9
  overview 4-3
  Reset Target option 7-6
  Restart option 7-2
  Return option 7-6
  Run option 7-4
  Step option 7-8

TARget system
  memory definition for debugger 5-1 to 5-13
  resetting 10-33
terminating the debugger 1-10 10-33
text files, displaying 8-9

U

UNALIAS command 10-43
undocking the toolbar 3-10 to 3-11
unmarking areas 9-12
USE command 10-44 A-4

V

–v debugger option 1-9
VAA command 10-44
VAC command 10-44
variables
  aggregate values in Watch window 8-17 to 8-19 10-17
determining type 8-2
displaying in different numeric format 8-22 10-46
displaying/modifying 8-17 to 8-19
  scalar values in Watch window 8-17 to 8-19
VERSION command 10-45
View menu
  making a window active 5-1 to 5-13
  overview 4-3
viewing code 6-2 to 6-3
viewing profile data
  description 9-20 to 9-26
displaying areas 9-24 to 9-26
  displaying different data 9-21 to 9-22 10-52
  sorting data 9-23 10-52
  viewing associated code 9-25 to 9-26

W

WA command
  description 10-46
display formats 8-22 10-46
Watch add dialog box 2-19 8-14 8-17

watch commands
  WA command 10-46
  WD command 10-47
  WR command 8-19 10-48
Index

Watch window
  adding items 8-17 to 8-18 10-46
  closing 8-19 10-48
  definition D-5
  deleting items 2-19 8-19 10-47
  description 2-18 to 2-19 8-17 to 8-19
  displaying additional data 2-19 8-19
  editing values 8-4
  effect of load commands 8-19
  labeling watched data 10-46
  opening 8-17 to 8-18 10-46
WD command 10-47
WHATIS command 8-2 10-47
WIN command 10-47
window commands 10-3
Window menu, overview 4-4
windows
  active window 2-21 to 2-22
  Calls window 2-12 8-9
  closing 3-6
  Command window 2-20 4-6
  CPU window 2-17 8-12 to 8-16
  definition D-5
  description 2-6 to 2-20
windows (continued)
  Disassembly window 2-10 to 2-11 8-6 to 8-7
  docking 3-4 to 3-5
  File window 2-8 to 2-9 6-8 to 6-10
  floating 3-2 to 3-5
  Memory window 2-14 to 2-16 8-6 to 8-11
  moving 10-28
  opening 3-6
  Profile window 2-13 9-20 to 9-26
  scrolling with function keys A-2
  sizing 10-38
  using context menus 2-7
  Watch window 2-18 to 2-19 8-17 to 8-19
WR command 8-19, 10-48

X
–x debugger option 1-9
X Window System, displaying debugger on a different machine 1-8

Z
ZOOM command 10-48
zooming a window 10-48