TMS320C27xx C Source Debugger User's Guide

Preliminary

Literature Number: SPRU214B
March 1998

Printed on Recycled Paper

IMPORTANT NOTICE

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

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

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

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

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

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

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

Preface

Read This First

About This Manual

This book tells you how to use the TMS320C27xx C source debugger with the following debugging tools to test and refine your code:

- Emulator
- Simulator

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

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

- The basic debugger environment is a general-purpose debugging environment. You can use standard data-management commands and run-type commands to test and evaluate your code.
- The profiling environment is a special environment for collecting statistics about code execution. You can use the profiling environment to identify areas in your code where you want to improve performance.

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

This book is meant to be used with the online help included with the C source debugger. The online help provides you with information about the windows, menu items, icons, and dialog boxes of the debugger interface. For information on how to access the online help, see section 1.5 on page 1-10.

Notational Conventions

This document uses the following conventions.

- Program listings, program examples, and interactive displays are shown in a `special typeface` similar to a typewriter's. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample program listing:

```
0011  0005  0001          .field    1, 2
0012  0005  0003          .field    3, 4
0013  0005  0006          .field    6, 3
0014  0006                .even
```

  Here is an example of a system prompt and a command that you might enter:

```
C:  csr –a /user/ti/simuboard/utilities
```

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

  .asect "*section name***",** *address*

  .asect is the directive. This directive has two parameters, indicated by *section name* and *address*. When you use .asect, the first parameter must be an actual section name, enclosed in double quotes; the second parameter must be an address.

- Square brackets (**[** and **]**) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you don't enter the brackets themselves. Here's an example of an instruction that has an optional parameter:

  run [*expression*]

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

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

```
{ * | *+ | *– }
```

  This provides three choices: *, *+, or *–.

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

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

  .byte *value$_1$ [, ... , value$_n$]*

  This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, separated by commas.

Related Documentation From Texas Instruments

The following books describe the TMS320C27xx 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.

TMS320C27xx Assembly Language Tools User's Guide (literature number SPRU211) describes the assembly language tools (assembler and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the TMS320C27xx device.

TMS320C27xx Optimizing C Compiler User's Guide (literature number SPRU212) describes the TMS320C27xx C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the TMS320C27xx device.

TMS320C27xx Translation Assistant User's Guide (literature number SPRU278) describes the TMS320C27xx translation utility and how it fits in with the rest of the TMS320C27xx code development tools. It tells you how to use the translation assistant to translate code you already have for TMS320C2xx devices into code that will run on TMS320C27xx devices.

TMS320C27xx DSP CPU and Instruction Set Reference Guide (literature number SPRU220) describes the central processing unit (CPU) and the assembly language instructions of the TMS320C27xx 16-bit fixed-point digital signal processors (DSPs). It also describes emulation features available on these DSPs.

T320C2700 Customizable Digital Signal Processor (cDSP) Core (literature number SPRS057) data sheet contains the electrical and timing specifications for these devices.

Related Documentation

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

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

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

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

FCC Warning

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

Trademarks

320 Hotline On-line, XDS, XDS510, XDS510PP, XDS510WS, and XDS511 are trademarks of Texas Instruments Incorporated.

PC is a trademark of International Business Machines Corporation.

Windows is a registered trademarks of Microsoft Corporation.

If You Need Assistance . . .

❑ World-Wide Web Sites

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

❑ North America, South America, Central America

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

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

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32

Email: epic@ti.com

Deutsch	+49 8161 80 33 11 or	+33 1 30 70 11 68
English		+33 1 30 70 11 65
Francais		+33 1 30 70 11 64
Italiano		+33 1 30 70 11 67
EPIC Modem BBS		+33 1 30 70 11 99
European Factory Repair		+33 4 93 22 25 40
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10

❑ Asia-Pacific

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

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

❑ Japan

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

❑ Documentation

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

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

Email: dsph@ti.com

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

Contents

Figures

Tables

Chapter 1

Overview of the Code Development and Debugging System

The C source debugger is an advanced programmer's interface that helps you to develop, test, and refine 'C27xx C programs (compiled with the 'C27xx optimizing ANSI C compiler) and assembly language programs. The debugger is the interface to the 'C27xx simulator and the scan-based emulator.

This chapter gives an overview of the C source debugger, describes the code development environment, and explains how you must prepare your program for debugging.

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

Topic **Page**

1.1 Key Features of the Debugger

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

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

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

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

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

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

- **Analysis module.** In addition to the basic debugger features, the 'C27xx has an analysis module on the chip that allows the emulator to monitor the operations of your target system. This expands your debugging capabilities beyond simple software breakpoints.

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

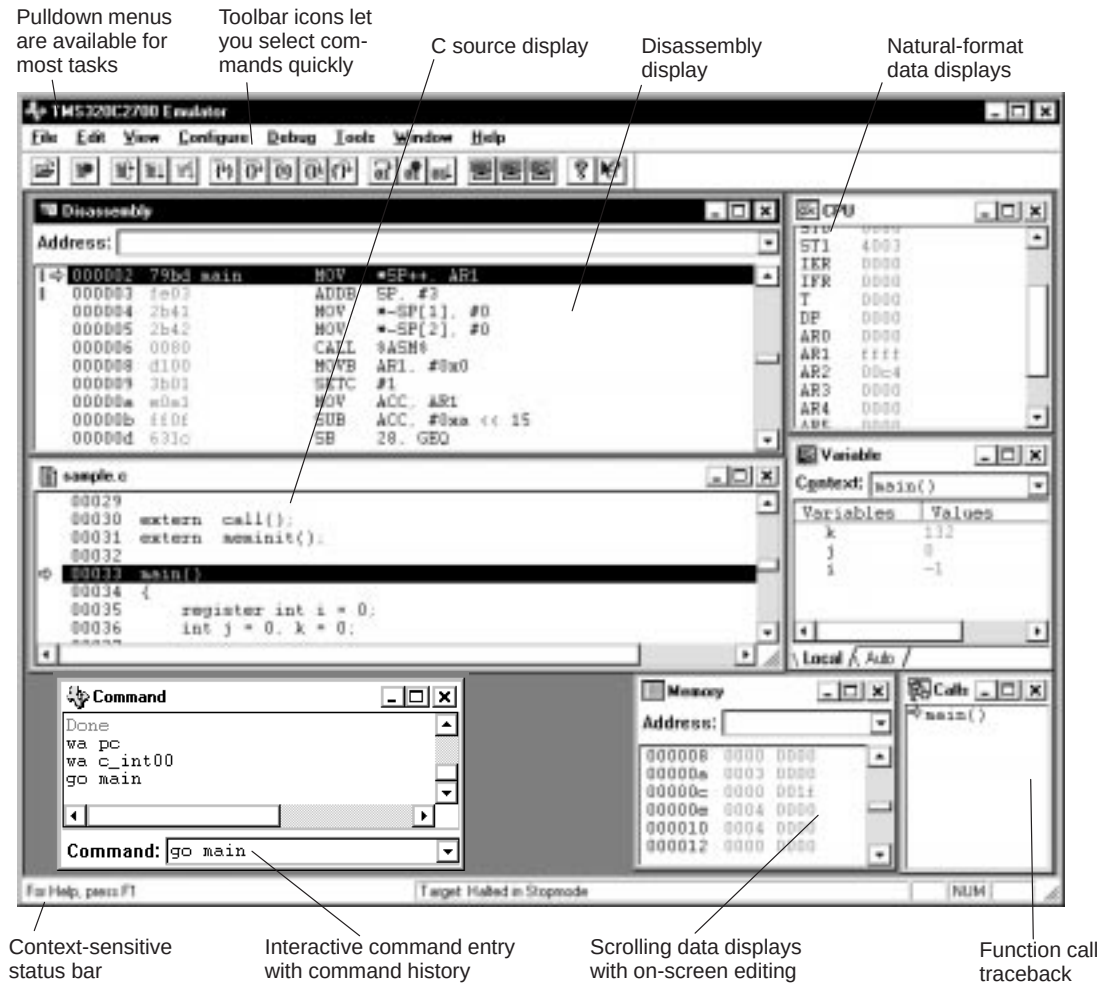
1.2 About the C Source Debugger Interface

The C source debugging interface improves productivity by allowing you to debug a program in the language 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



Descriptions of the debugger windows and their contents

The debugger can show several types of windows. Each type of window serves a specific purpose and has unique characteristics. Every window is identified by a name in its upper left corner. For the File window, the debugger displays the name of the file shown in the window instead of the word File. There are 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 if you are debugging a C program.
- The Profile window displays statistics about code execution.
- Data-display windows are for observing and modifying various types of data. There are four data-display windows:
  - A Memory window displays the contents of a range of memory. You can display multiple Memory windows to allow you to view different sections of memory at one time.
  - The CPU window displays the contents of ’C27xx registers.
  - A Watch window displays selected data such as variables, specific registers, or memory locations. You can display multiple Watch windows to allow you to view multiple variables, register, or memory locations at one time.
  - A Variable window displays all variables declared in the current procedure, or all variables in the currently executing line of C code as well as variables in the previous procedures.
- The Command window provides an area for typing in commands and reentering commands and an area for displaying various types of information, such as progress messages, error messages, or command output.

Table 1–1 summarizes the purpose of each window, how each window is created, and in which debugging mode each window is visible.

Table 1–1. Summary of Debugger Window Descriptions

Window	Purpose	Created	Mode
Calls	Lists the current function, its caller, and the caller's caller, etc. for C functions	❑ Automatically when you are displaying C code ❑ With the CALLS command if you previously closed the Calls window	❑ Auto ❑ Mixed
Command	❑ Provides a command line for entering commands ❑ Provides a display area for echoing commands and displaying command output, errors, and messages	Automatically	All
CPU	Shows the contents of the 'C27xx registers	Automatically	All
Disassembly	Displays the disassembly (or reverse assembly) of memory contents	Automatically	All
File	❑ Displays C source files ❑ Displays assembly source files ❑ Displays text files	❑ With the File→Open menu option ❑ Automatically when your program executes C code, assembly code, or serial assembly code assembled with the –g assembler option	❑ Auto ❑ Mixed
Memory	Displays the contents of memory. Reference addresses, determined by the size of the window, are listed in the first column.	❑ Automatically for the default Memory window only ❑ With the MEM command and a unique *window name* for additional Memory windows	All
Profile	Displays statistics collected during a profiling session	By entering the profiling environment: Tools→Profile→Profile Mode	Mixed
Watch	Displays the values of selected expressions, structures, arrays, or pointers	❑ With the Configure→Watch Add menu option ❑ With the WA and DISP commands	All
Variable	Displays variables and the associated values for the current procedure as well as variables on the currently executing line of C code and those variables from the previous statement.	❑ Automatically when you are displaying C code ❑ With the CALLS command if you have previously closed the Calls window ❑ With the View→Variable Window menu option	❑ Auto ❑ Mixed

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

1) Move your pointer over a debugger window.

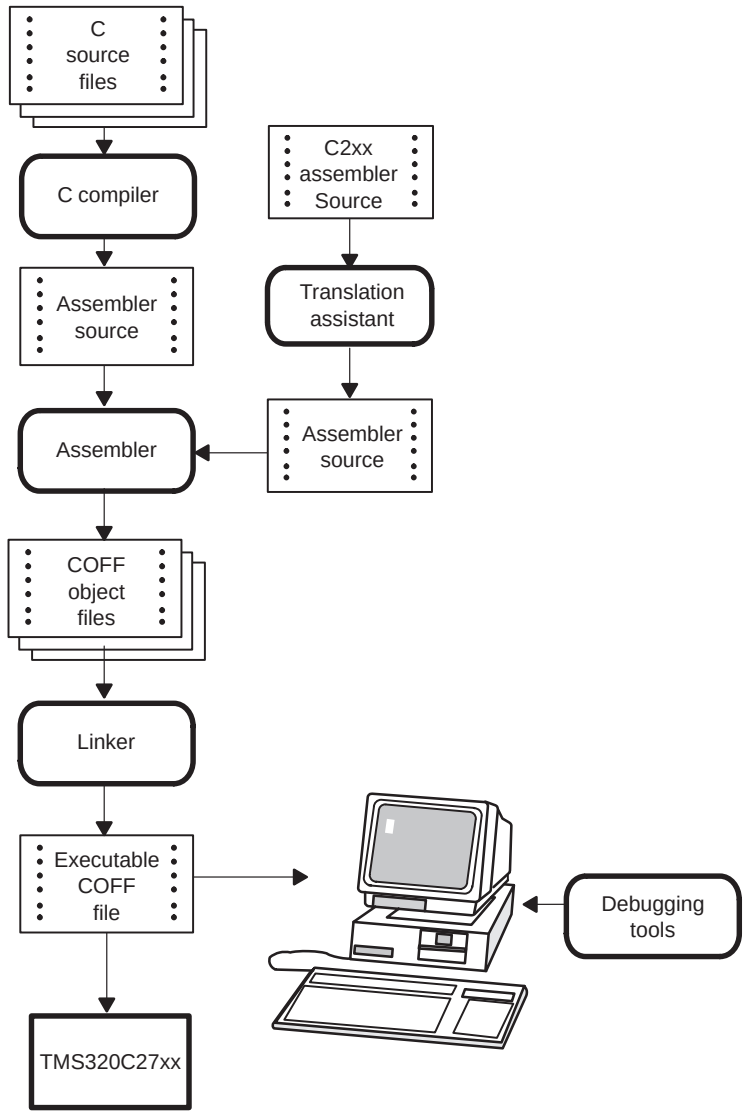
2) Click the right mouse button. This displays a context menu like the following example:

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

1.3 Developing Code for the TMS320C27xx

The ’C27xx is well supported by a complete set of hardware and software development tools, including a C compiler, an assembler, and a linker. Figure 1–2 illustrates the basic ’C27xx code development flow.

Figure 1–2. TMS320C27xx Software Development Flow



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

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

- The **C compiler** accepts C source code and produces TMS320C27xx 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 *TMS320C27xx Optimizing C Compiler User's Guide* for more information.

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

  See the *TMS320C27xx 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.

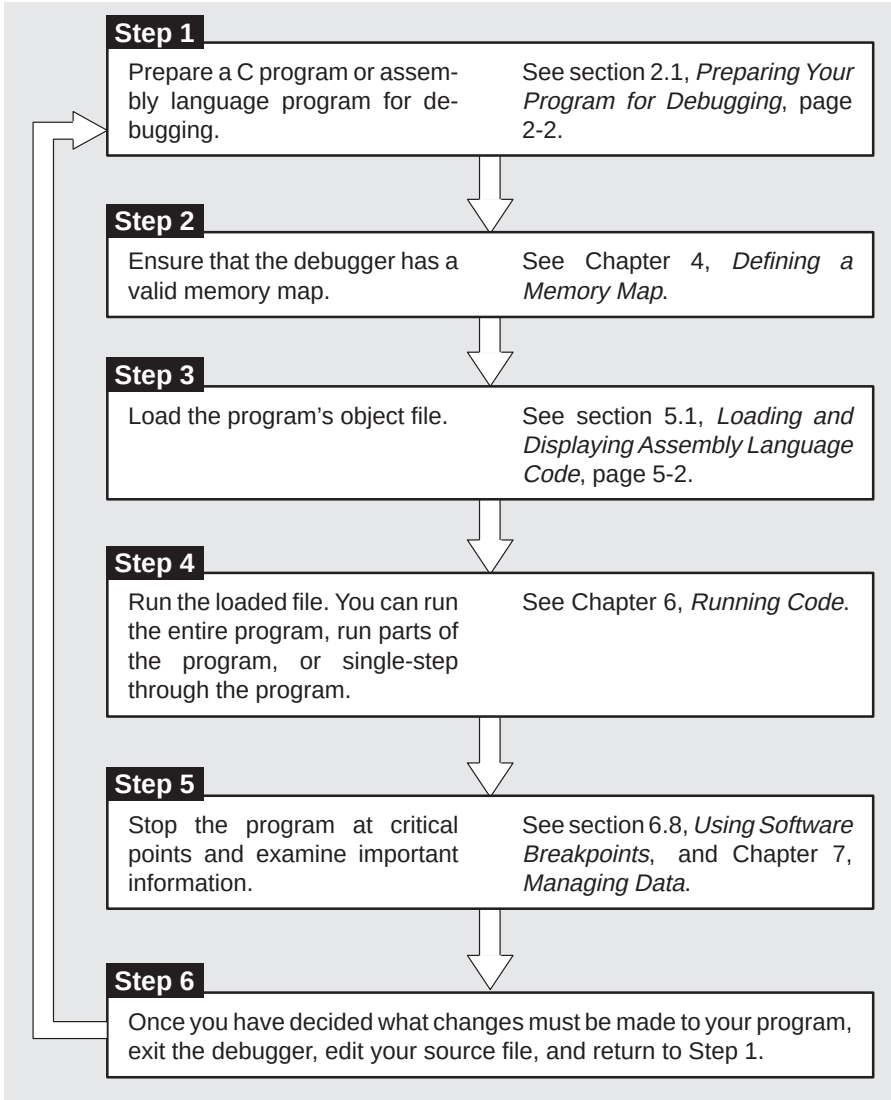
  See the *TMS320C27xx Assembly Language Tools User's Guide* for more information.

- The main product of this development process is a module that can be executed in a **TMS320C27xx target system**.

- You can use debugging tools to refine and correct your code. Available products include:
  - An instruction-accurate and clock-accurate software simulator
  - An XDS™ emulator

1.4 Overview of the Debugging Process

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



1.5 Accessing Online Help

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

Accessing a list of help topics

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

1) Open the list of help topics by using one of these methods:

   ❏ Click the Help Contents icon on the toolbar:

   ❏ From the Help menu, select Help Topics.

   ❏ From the command line, enter:

   `help` ↵

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

Accessing context-sensitive help

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

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

   1) Click the Help icon on the toolbar:

   This changes the pointer to a question mark.

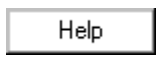
   2) Select the menu option or click on the item that you want more information about.

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

   1) Make the window or the dialog box active.

   2) Press F1.

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

Accessing help for debugger commands

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

help *debugger command*

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

Chapter 2

Getting Started With the Debugger

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

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

Topic **Page**

2.1 Preparing Your Program for Debugging

Before you use the debugger, you must create an executable object file. To do so, start with C source and/or assembly language code. You can use the cl27 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 cl27 shell program and its options and about creating an executable object file for use with the debugger, see the *TMS320C27xx Optimizing C Compiler User's Guide*.

Debugging optimized code

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

Profiling optimized code

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

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

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

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

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

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

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

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

2.3 Identifying Directories That Contain Program Source Files (D_SRC)

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

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

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

For example, if your ’C27xx programs were in a directory named source on drive C, the D_SRC setup would be:

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

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

2.4 Setting Up Default Debugger Options (D_OPTIONS)

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

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

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

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

The *filename* identifies the optional object file for the debugger to load, and *options* lists the options you want to use at invocation. Section 2.7 on page 2-8 summarizes the options that you can identify with D_OPTIONS.

2.5 Resetting the Emulator

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

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

The –x option tells the emurst utility to ignore any options specified with the D_OPTIONS environment variable. For more information about the –x option, see page 2-11.

The –p option *number* identifies the I/O port address that the debugger uses for communicating with the emulator. For more information about the –p option, see page 2-10.

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

```
CANNOT DETECT TARGET POWER
```

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

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

2.6 Invoking the Debugger

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

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

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

2.7 Summary of Debugger Options

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

Table 2–1. Summary of Debugger Options

Option	Brief Description	Debugger Tools
–@	Recognize BTT software commands	Emulator
–c	Clear the .bss section	All
–cnf *filename*	Identify a memory configuration file	Simulator
–f *filename*	Identify a new board configuration file	Emulator
–i *pathname*	Identify additional directories	All
–n *device_name*	Identify device for debugging	Emulator
–s *filename*	Load the symbol table only	All
–t *filename*	Identify a new initialization file	All
–v	Load without the symbol table	All
–x	Ignore D_OPTIONS	All

Recognizing BTT commands (–@ option)

The –@ option allows the emulator to recognize BTT software commands. Use this option when you are using a BTT device, such as the XDS522.

Clearing the .bss section (–c option)

The –c option clears the .bss section when the debugger loads code. Use this option when you have C programs that use the RAM initialization model (specified with the –cr linker option described in the *TMS320C27xx Assembly Language Tools User's Guide*).

Identifying the memory configuration file (–cnf option)

The –cnf option allows you to specify a customized memory configuration file to use instead of sim.cnf. The format for the –cnf option is:

–cnf *filename***.cnf**

Using this option is similar to loading a batch file by using the debugger's File→Load→Memory Config menu option. See section 4.9 on page 4-18 for details on the memory configuration file.

Identifying a new configuration file (–f option)

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

–f *filename***.dat**

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

Identifying additional directories (–i option)

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

–i $pathname_1$ **–i** $pathname_2$ **–i** $pathname_3$...

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

Identifying the port address (–p option)

The –p option specifies which port the debugger uses to communicate with the emulator. The –p option is valid only when you are using the emulator. The format for the –p option is:

–p *port_address*

The –p option identifies the I/O port address that the debugger uses for communicating with the emulator. If you used the default switch settings, you do not need to use the –p option. **If you use nondefault switch settings, you must use –p**. For information on switch settings, see the *XDS51x Installation Guide*; determine your switch settings, and replace *port address* with one of these values:

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

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

Loading the symbol table only (–s option)

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

–s *filename***.out**

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

Identifying a new initialization file (–t option)

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

–t *filename.cmd*

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

Loading without the symbol table (–v option)

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

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

Ignoring D_OPTIONS (–x option)

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

2.8 Debugging Modes

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

Auto mode

In *auto mode*, the debugger automatically displays whichever type of code is currently running: assembly language or C. 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 2–2. The Disassembly window displays the reverse assembly of memory contents.

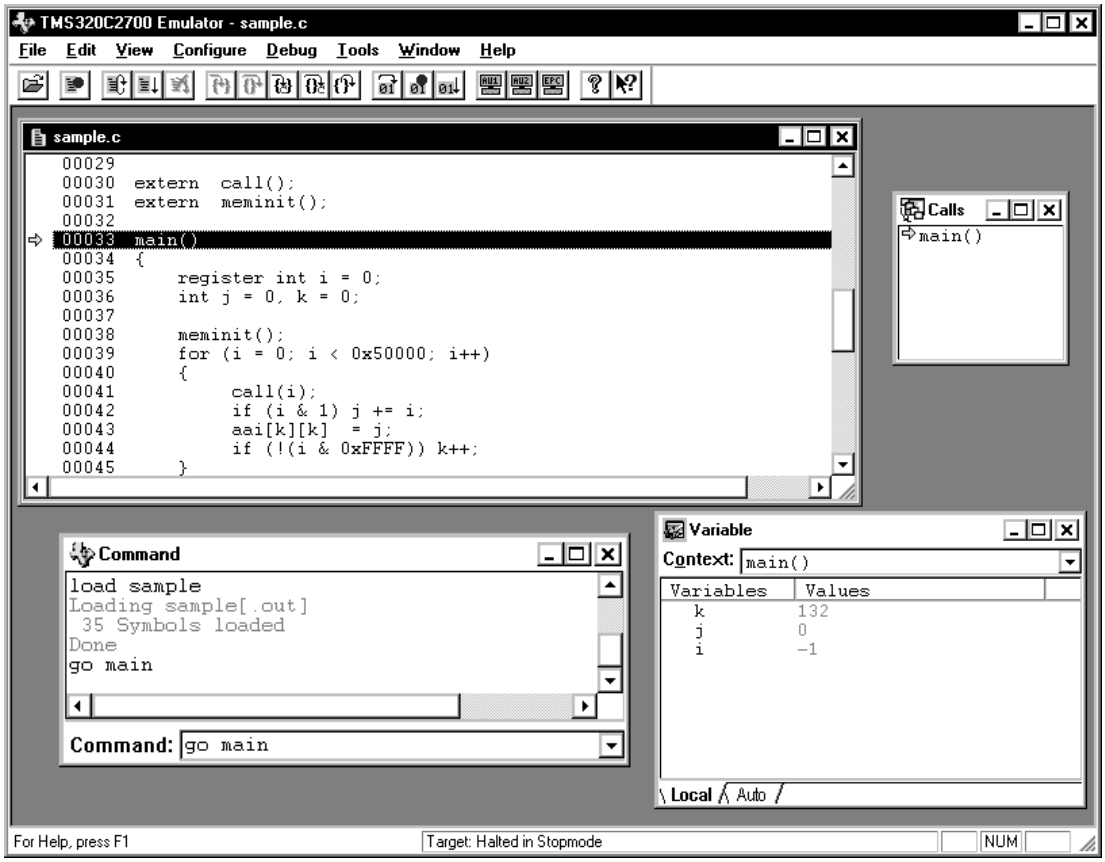
  When you first invoke the debugger, you see a display similar to this.

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

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

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

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

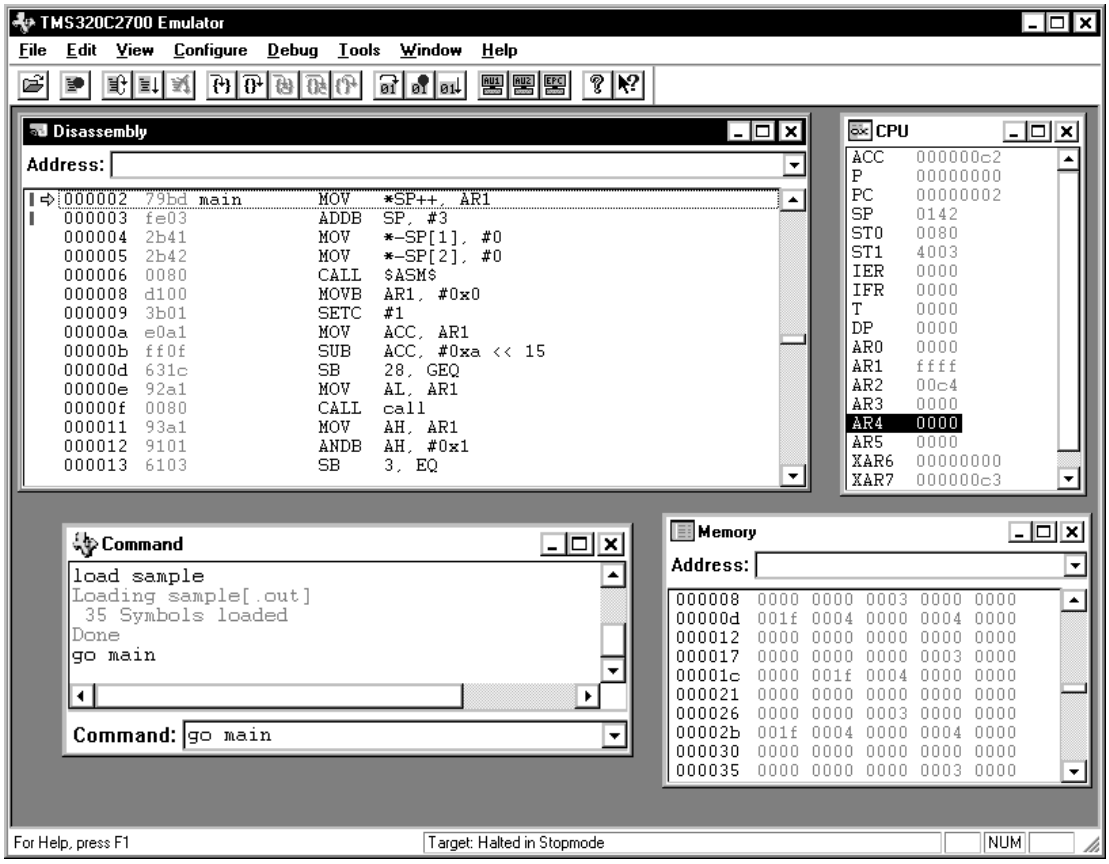


Assembly mode

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

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

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



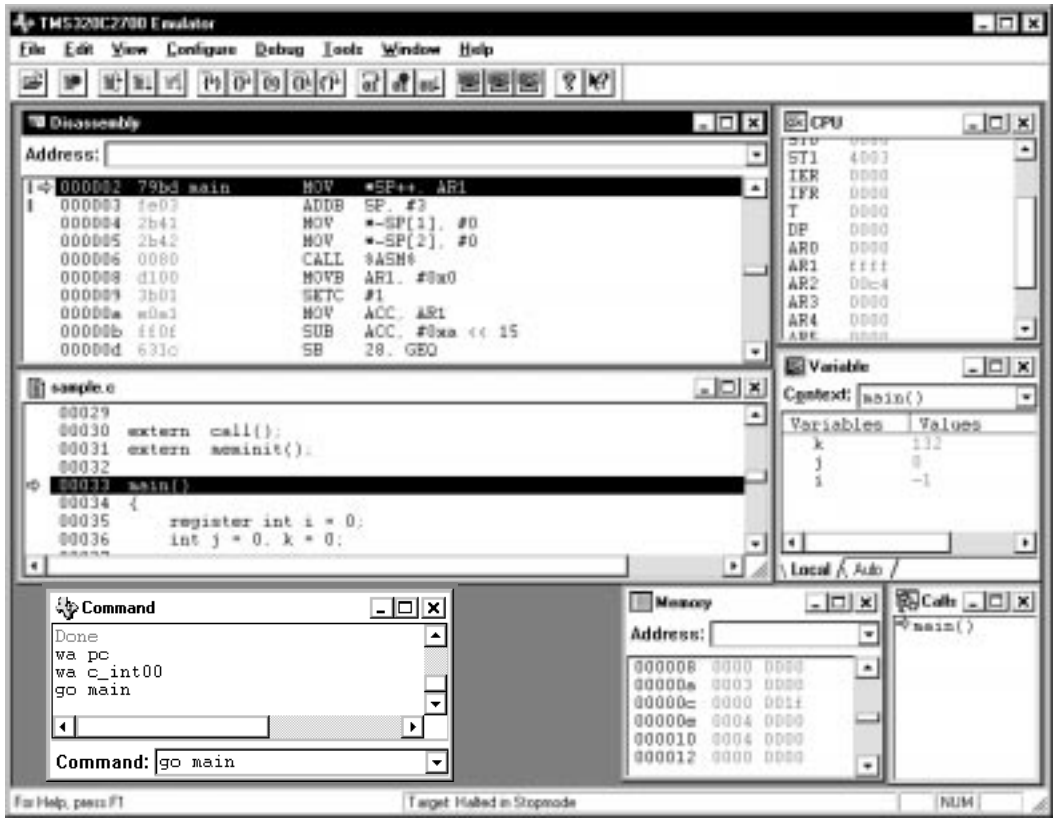
Mixed mode

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.

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



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.

2.9 Execution Modes (Simulator Only)

The 'C27xx simulator supports two pipeline execution modes: simulation and emulation. The principal difference between the modes is in the state of the pipeline when execution is halted. The simulator starts up in simulation mode by default. The EMU command switches to emulation mode. The SIM command switches to simulation mode. Emulation mode replicates the pipeline execution process of the emulator.

Pipeline differences associated with execution modes

When you halt the debugger in simulation mode, the pipeline is not flushed. Any instruction that has not completed all stages of pipeline execution may not be represented correctly by the values in the registers and other memory locations normally affected by that instruction. So, it is possible to single-step through an instruction and not see the result that instruction has on a register or memory location.

When you halt the debugger in emulation mode, the pipeline is flushed. Any instruction that is in the pipeline is pushed through the pipeline to completion, while the next instruction enters the decode 2 phase and stops. Emulation mode makes debugging easier when you single-step through code, because each instruction completes through the pipeline and updates the machine state before the next instruction processes. However, flushing the pipeline masks pipeline conflicts.

Use simulation mode to locate pipeline conflicts. Use emulation mode to verify that your instructions have the desired effect on registers and memory locations when accurate cycle counts are not critical.

To locate pipeline conflicts using simulation mode, you need to cycle-step through your code with the pipeline display enabled. See section 5.2, *Displaying Pipeline Phases With Assembly Language Code*, on page 5-6 for more information.

Debugger operation differences associated with execution modes

There are several critical differences in the behavior of the debugger in the two execution modes. Here are the differences:

- **Cycle-step**. The cycle-step command is designed for use with the pipeline display feature to help you to locate pipeline conflicts in your code. In emulation mode, cycle-step is disabled.
- **Instruction breakpoints**. In simulation mode, breakpoints are implemented by address comparison. In emulation mode, breakpoints are implemented by instruction replacement: the debugger replaces the breakpointed instruction with an ESTOP0 instruction and then runs until an ESTOP0 instruction is executed. Emulation mode affects the number of cycles accumulated due to breakpoints.
- **Repeat instructions**. In simulation mode, single-stepping executes an RPT instruction once. In emulation mode, single-stepping causes the entire RPT loop to execute and the debugger stops at the next instruction.
- **RUNF command**. When you use the RUNF (run free) command, the simulator stops every 30 instructions to poll for any events you have set up. In simulation mode this has no effect on the machine state. In emulation mode each of these stops is accompanied by a pipeline flush, which increases the cycle count artificially.
- **IC and PC registers**. In simulation mode, the instruction counter (IC) register points to decode 1 and the program counter (PC) register points to decode 2. In emulation mode, these two registers are equal.

2.10 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` ⏎

Chapter 3

Entering and Using Commands

The debugger provides you with several methods for entering commands:

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

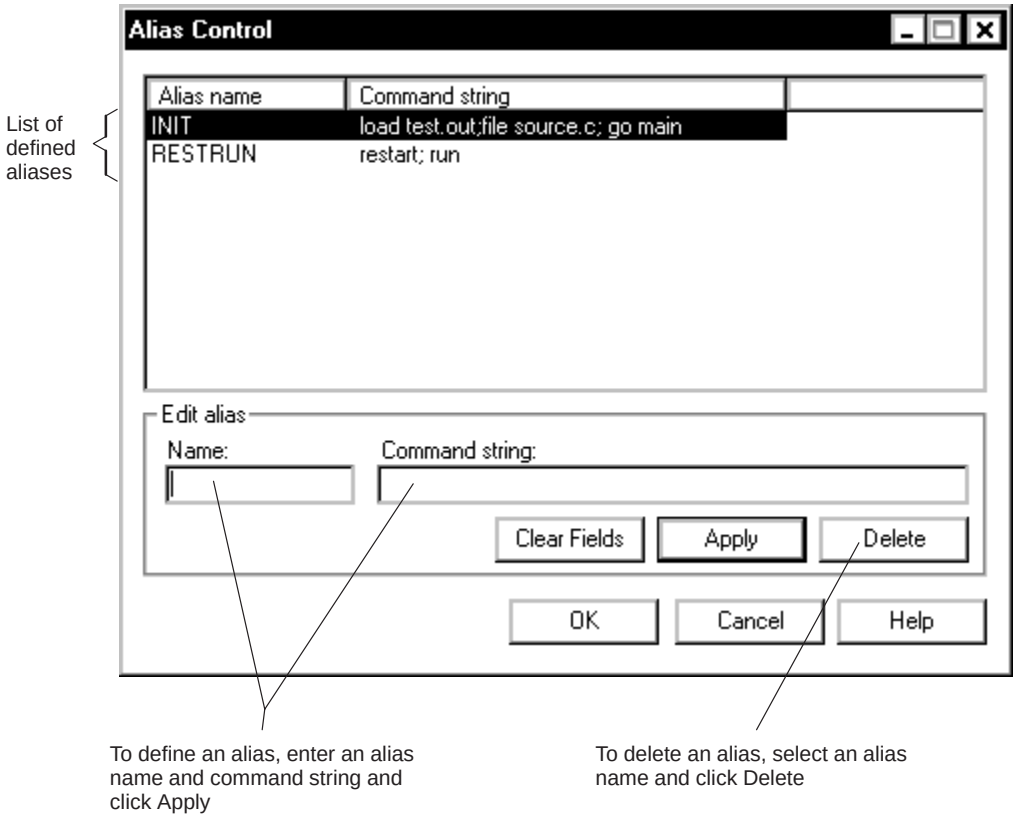
This chapter describes how you can create aliases for commands and command sequences that you enter frequently, as well as information about using a batch file or a log file for entering commands.

Topic **Page**

3.1 Defining Your Own Command Strings

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

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



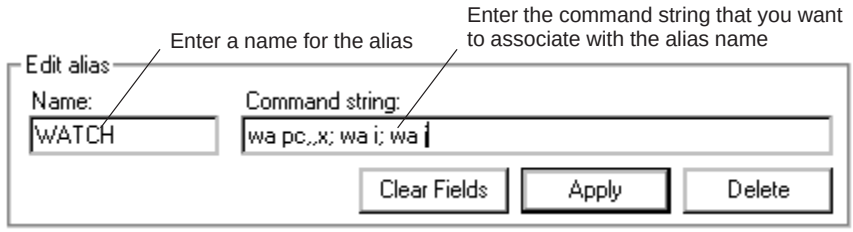
Defining an alias

To define an alias, follow these steps:

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

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

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



4) Click Apply.

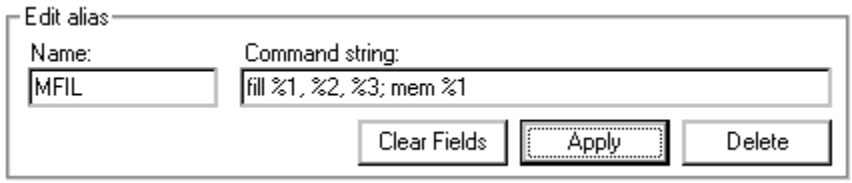
5) Click OK to close the Alias Control dialog box.

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 Configure menu, select Alias Commands. This displays the Alias Control dialog box.
2) From the list of aliases at the top of the dialog box, select the alias that you want to edit or redefine.
3) In the Name and Command string fields, make the appropriate changes.
4) Click Apply.
5) Click OK to close the Alias Control dialog box.

Deleting an alias

To delete an alias, follow these steps:

1) From the Configure menu, select Alias Commands. This displays the Alias Control dialog box.
2) From the list of aliases at the top of the dialog box, select the alias that you want to delete.
3) Click Delete.
4) Click OK to close the Alias Control dialog box.

Considerations for using alias definitions

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

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.

3.2 Entering Operating-System Commands From Within the Debugger

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

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

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

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

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

Entering a single command from the debugger command line

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

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

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

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

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

```
exit ↵.
```

(This is the default.)

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

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

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

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

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

Entering several commands from a system shell

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

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

exit ↵

3.3 Creating and Executing a Batch File

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

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

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

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

```
echo Loading object code
load testcode.out

echo Loading screen configuration
sconfig myconfig.clr

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

Echoing strings in a batch file

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

echo *string*

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

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

```
echo Creating new memory map
```

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

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

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

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

Executing commands conditionally in a batch file

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

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

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

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

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

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

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

```
if $$EMU$$
echo Invoking initialization batch file for emulator.
use .\emu27
take emuinit.cmd
.
.
.
endif

if $$SIM$$
echo Invoking initialization batch file for simulator.
use .\sim27
take siminit.cmd
.
.
.
endif
.
.
.
```

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

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

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

Looping command execution in a batch file

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

loop *expression*
debugger commands
endloop

These looping commands evaluate using the same method as the conditional RUN command expression. (See Chapter 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 code sequence:

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

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

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

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

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

```
loop !0
step
? PC
? AR0
endloop
```

The LOOP/ENDLOOP command works with the following conditions:

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

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:

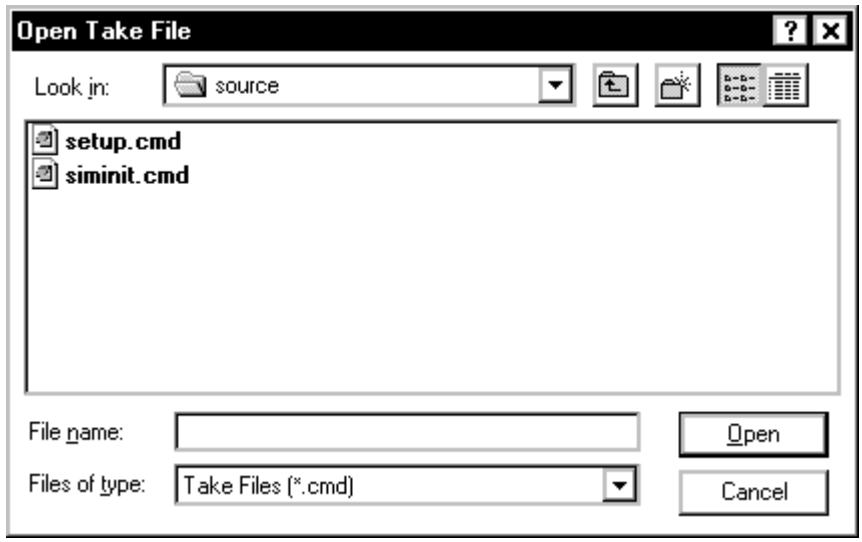
pause

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 ↵.

Executing a batch file

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

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



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

3) Click Open.

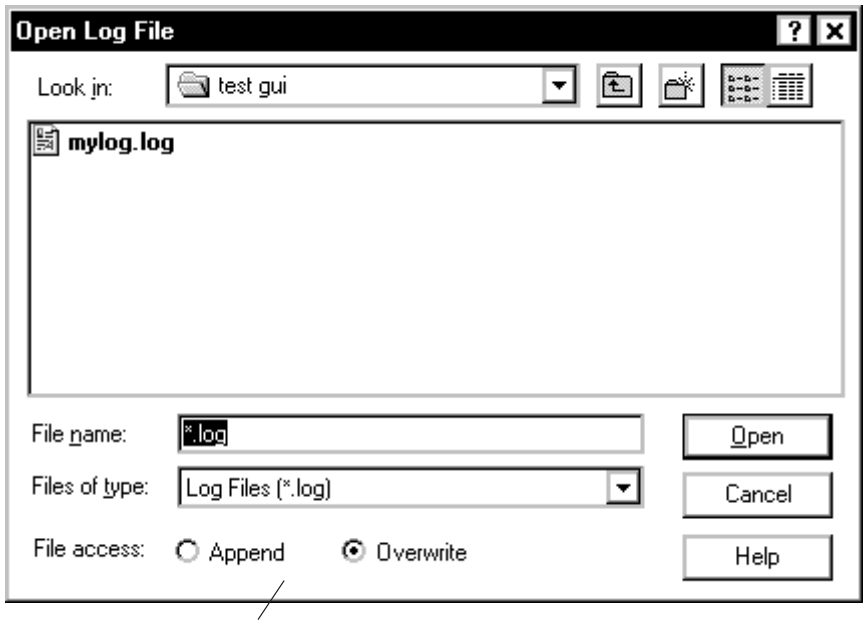
This causes the debugger to read and execute the commands in the batch file. To halt the debugger's execution of a batch file, press ESC.

3.4 Creating a Log File to Reexecute a Series of Commands

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

To begin a recording session, follow these steps:

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



Select whether to append or overwrite an existing log file.

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

3) In the File name field, enter a name for the log file. Use a .log extension to identify the file as a log file.

4) Click Open.

5) If the file that you want to use already exists, select that file, then select one of the following actions in the File access field:

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

6) Click Open.

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

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

Chapter 4

Defining a Memory Map

Before you begin a debugging session, you must supply the debugger with a memory map. The memory map tells the debugger which areas of memory it can and cannot access. The memory map is the only means for the 'C27xx emulator to distinguish between program and data memory.

Topic **Page**

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

A memory map tells the debugger which areas of memory it can and cannot access. The memory map is the only way that the 'C27xx emulator is able to distinguish between program and data memory. Memory maps vary, depending on the application being used. Typically, the memory map matches the MEMORY definition found in your linker command file.

Note:

When the 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 initially begin using the debugger, but may need to be altered at a later time. The debugger enables you to modify the default memory map or define a new memory map either interactively (as described in section 4.2 on page 4-4) or by defining the memory map in a batch file (as described in section 4.5 on page 4-12).

Potential memory map problems

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

- **Accessing invalid memory addresses.** If you do not supply a batch file containing memory-map commands, the debugger is initially unable to access any target memory locations. Invalid memory addresses and their contents are displayed in red in the data-display windows by default.
- **Accessing an undefined or protected area.** When memory mapping is enabled, the debugger checks each of its memory accesses against the memory map. If you attempt to access an undefined or protected area, the debugger displays an error message.

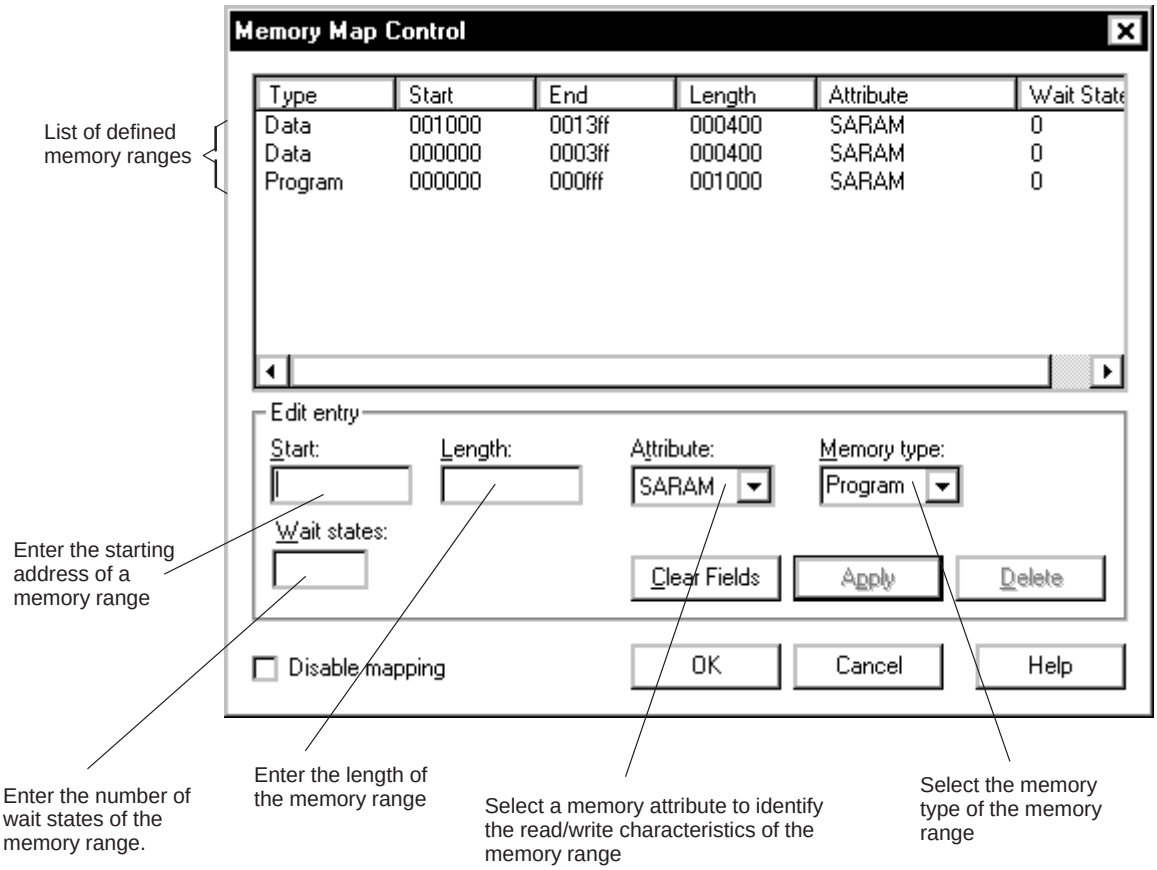
- **Loading a COFF file with sections that cross a memory range.** Be sure that the map ranges you specify in a COFF file match those that you defined in a batch file or with the Memory Map Control dialog box. Alternatively, you can turn memory mapping off during a load by disabling memory mapping (as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

Note:

If the emulator accesses an illegal or reserved memory location, it posts an error in the EMIF (external memory interface) control register and returns 0 as the data.

4.2 Creating or Modifying the Memory Map

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



Adding a range of memory

To add a range of memory, follow these steps:

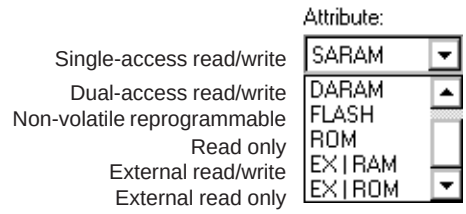
1) From the Configure menu, select Memory Maps. This displays the Memory Map Control dialog box.

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

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

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

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



6) In the Wait States field, enter the number of clock cycles to use as wait states for the memory range.

7) Click Apply.

8) 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.
- The origin and length values for a range that you define with the MEMORY directive in your linker command file must match the Start and Length values for the same range in the Memory Map Control dialog box.

Creating a customized memory type

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

To create a customized memory type, follow these steps:

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

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

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

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

13) From the Basic Types column, select the individual memory attributes that you want to apply to the memory range that you are adding.

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

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

Deleting a range of memory

To delete a range of memory, follow these steps:

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

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

3) Click Delete.

4) Click OK.

Before you can delete from the memory map a memory address used to connect to a file, you must disconnect the address. See the *Disconnecting a file* section on page 4-24 for information.

Modifying a defined range of memory

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

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

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

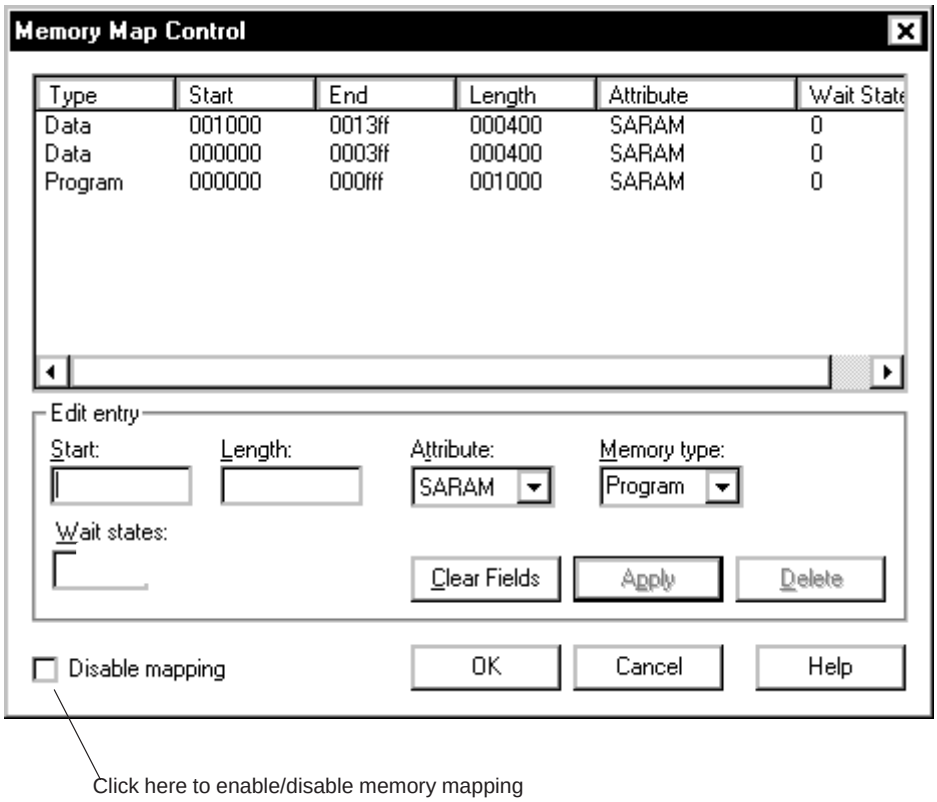
3) In the Memory Type, Start, Length, Attribute and/or Wait states fields, make the appropriate changes.

4) Click Apply.

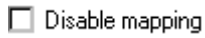
5) Click OK.

4.3 Enabling Memory Mapping

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



- Memory mapping is enabled when the box is empty:



- Memory mapping is disabled when the box is checked:

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

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

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

Note:

When memory mapping is enabled, you cannot:

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

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

```
Error in expression
```

4.4 A Sample Memory Map

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 4–1 *(a)* shows the memory map that is defined in the initialization batch file that accompanies the 'C27xx simulator. You can use the file as is, edit it, or create your own memory map batch file to match your own configuration. You can also define the memory map after you have invoked the debugger with the Memory Map Control dialog box (see section 4.2 on page 4-4).

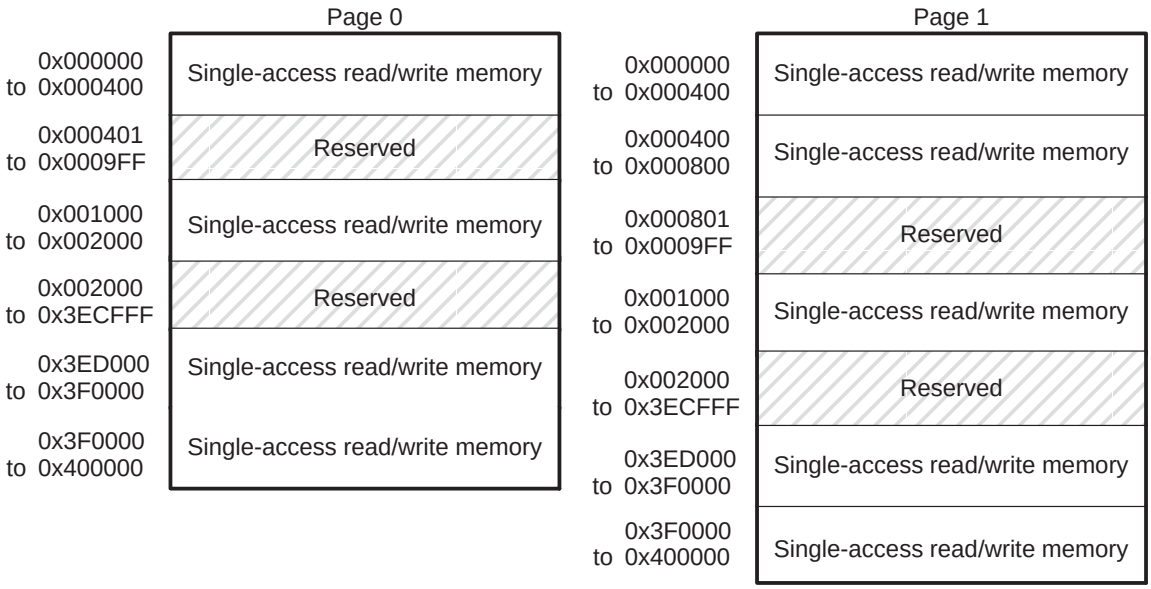
If you are defining the memory map in a batch file, you can use MA (map add) commands to define valid memory ranges and identify the read/write characteristics of the memory ranges. (For more information about the MA command, see section 4.5 on page 4-12.) By default, mapping is enabled when you invoke the debugger. Figure 4–1 illustrates the memory map defined by the MA commands in Figure 4–1 *(a)*.

Figure 4–1. Sample Memory Map for Use With a TMS320C27xx Simulator

(a) Memory map commands

```
ma 0x0 , 0, 0x400, SARAM
ma 0x0 , 1, 0x400, SARAM
ma 0x400, 1, 0x400, SARAM
ma 0x1000, 1, 0x1000, SARAM
ma 0x1000, 0, 0x1000, SARAM
ma 0x3ED000, 1, 0x3000, SARAM
ma 0x3ED000, 0, 0x3000, SARAM
ma 0x3F0000, 1, 0x10000, SARAM
ma 0x3F0000, 0, 0x10000, SARAM
```

(b) Memory map for TMS320C27xx local memory



4.5 Defining and Executing a Memory Map in a Batch File

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

Defining a memory map in a batch file

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

ma *address, page*, *length, type*

- The *address* parameter defines the starting address of a range. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.
- The *page* parameter is a one-digit number that identifies the type of memory (program or data) that a range occupies:

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

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

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

To identify this kind of memory . . .	Use this keyword as the *type* parameter . . .
Read-only memory	**R** or **ROM**
Write-only memory	**W** or **WOM**
Read/write memory	**R|W** or **RAM**
External read/write memory	**EX|RAM**
External read-only memory	**EX|ROM**
Single-access read/write memory	**SARAM**
Dual-access read/write memory	**DARAM**
Nonvolatile reprogrammable memory	**FLASH**

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

Executing a memory map batch file

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

- Use the File→Execute Take File... menu option from within the debugger environment.
- Use the –t debugger option to specify the batch file when you invoke the debugger. For more information, see page 2-11.
- Use the TAKE command. For more information, see section 3.3, *Creating and Executing a Batch File*, on page 3-7.

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

1) It checks to see whether you have used the –t debugger option. The –t option allows you to specify a batch file other than the initialization batch file shipped with the debugger. If it finds the –t option, the debugger reads and executes the specified file.
2) If you do not use the –t option, the debugger looks for the default initialization batch file. The batch filename for the simulator is called siminit.cmd. The batch filename for the emulator is called emuinit.cmd. If the debugger finds the proper initialization batch file, it reads and executes the file.

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

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

4.6 Returning to the Original Memory Map

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

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

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

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

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

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

4.7 Using Multiple Memory Maps for Multiple Target Systems

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

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

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

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

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

3) Invoke the debugger as usual.

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

   This redefines the memory map for the current debugging session.

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

4.8 The Memory Configuration: What It Is and Why You Must Define It (Simulator Only)

A memory configuration tells the simulator what types of memory (such as RAM or SARAM) are used in your system. A memory map used in conjunction with a memory configuration tells the debugger which ranges of memory of each type are actually being used.

A special default configuration file included with the debugger package defines a memory configuration for your version of the debugger. This configuration may be sufficient when you first begin using the debugger. However, the debugger enables you to modify the default configuration (sim.cnf) or define a configuration in a batch file (see section 4.9 on page 4-18).

Potential memory configuration problems

If the memory configuration file is not correctly defined or cannot be found at startup, the simulator activates only the main window and the Command window. Only the File→Load→Memory Config and File→Take menu options are enabled. The debugger waits in this state until it receives a valid configuration file via one of these two menu options.

The File→Take option assumes that the batch file specified contains a MEMFILE command that identifies the memory configuration filename. The MEMFILE command must be placed before any MA commands, or the memory map commands are ignored.

4.9 Defining and Executing a Memory Configuration Batch File

You can create a batch file that contains memory configuration commands. This is a convenient way to define what types of memory are available in your system. You can define the memory in the default memory configuration batch file (sim.cnf), which executes when you invoke the debugger, or you can define the memory in a separate batch file of your own that you can execute using the File→Load→Memory Config menu option or the –cnf debugger option.

Defining a memory configuration batch file

To define a memory configuration, you must set up a memory configuration batch file that lists the memory modules and blocks. Statement names and parameters are not case sensitive; to emphasize this, they are shown in both uppercase and lowercase throughout this chapter. Your file must be in the following format:

module *name*
  memory *name*
  first *address*
  last *address*
  space {**PROG** | **DATA**}
  type {**SARAM** | **DARAM** | **RAM attributes EX**| **ROM** [**attributes EX**]| **FLASH**}
  [**waitstates** *number*]
  [**constraints shared=***name*]
  end *name*
end *name*

- The **module** *name* statement begins the module description and identifies the module. The *name* can be any combination of letters, numbers or underscores that does not match a memory configuration statement name or keyword. Use unique names; the debugger does not check for duplicate names. The keywords read and write are reserved but not used; do not name any memory blocks by these names.
- The **memory** *name* statement begins the memory block description and identifies the memory block. You can have multiple memory blocks within a single module. The *name* can be any combination of letters, numbers, or underscores that does not match a memory configuration statement name or keyword. Use unique names; the debugger does not check for duplicate names. The keywords read and write are reserved but not used; do not name any memory blocks by these names.
- The **first** *address* statement identifies the starting address of a range. The *address* can be a decimal or hexadecimal number. Use the **0x** prefix for hexadecimal addresses; otherwise, the debugger treats the number as a decimal address.

- The **last** *address* statement identifies the ending address of a range. The *address* can be a decimal or hexadecimal number. Use the **0x** prefix for hexadecimal addresses; otherwise, the debugger treats the number as a decimal address. Memory block lengths must be even.
- The **space** {**PROG** | **DATA**} statement identifies the page space that contains the memory block. Use the prog parameter to identify program memory; use the data parameter to identify data memory. You can set up shared memory blocks when each block is in a different page space.
- The **type** {**SARAM** | **DARAM** | **RAM** | **ROM** | **FLASH**} statement identifies the read/write characteristics of the memory range. The parameter must be one of these keywords:

To identify this kind of memory . . .	Use this keyword as the *type* statement . . .
Read-only memory	**ROM** [**attributes EX**]
Read/write memory	**RAM attributes EX**
Single-access read/write memory	**SARAM**
Dual-access read/write memory	**DARAM**
Nonvolatile reprogrammable memory	**FLASH**

  The **attributes EX** parameter must be used with type RAM because the debugger supports accessing RAM only through the external interface. This statement is valid only with the RAM and ROM memory types.
- The optional **waitstates** *number* statement specifies the number of wait states for a memory block. If you do not use the waitstates statement, the memory has 0 wait states by default.
- The optional **constraints shared=***name* statement specifies that this memory block is shared with *name* memory block. Sharing means the specified memory ranges correspond to the same physical block of memory. A write to one memory appears in the other memory. The shared constraint can be specified in either of the memory blocks. Shared memory blocks must have wait states of 0. Sharing of memory blocks is not allowed when each block is on the same page (PROG or DATA).
- The **end** *name* statement ends a memory description. The *name* must be the same as the name used with the corresponding memory keyword.
- The **end** *name* statement ends a module description. The *name* must be the same as the name used with the corresponding module keyword.

Example 4–1 illustrates the memory configuration file format. You can insert comments in your file by preceding the comment with ; or //. Any text that appears after these delimiters is ignored.

Example 4–1. Sample Memory Configuration File

```
module mymem1              // Module for mymem1

    memory m1              // Description for memory block m1
              first      0x0      ; Starting address of m1
              last       0xFFFF   ; Last address of m1
              space      prog     ; m1 maps to the program space
              type       saram    ; m1 is single–access RAM
              waitstates 2        ; Waitstates supported by m1
    end m1

    memory m2              // Description for memory block m2
              first      0x2      ; Starting address of m2
              last       0xFFFD   ; Last address of m2
              space      data     ; m1 maps to the data space
              type       daram    ; m1 is dual–access RAM
    end m2
end mymem
```

Interaction with the memory map commands

While the simulator memory configuration file specifies what kinds of memory are available in the system, the MA command tells the debugger which ranges of memory you actually use. Any memory address accessed within the debugger must be included in a memory range specified by MA. Memory ranges specified with the MA command cannot span block boundaries defined in the configuration file.

The following code accesses memory blocks set up in Example 4–1.

```
ma 0x0, 0, 0xFFFF, saram, 2
ma 0x2, 1, 0xFFFD, daram
```

You can specify MA commands in your memory configuration file or in a memory initialization file. See the *Defining a memory map in a batch file* section on page 4-12 for more information.

Connecting memory blocks to the external interface

The debugger supports connecting to memory blocks in external RAM. In the simulator configuration file (.cnf extension), the memory attribute for the block must be specified as *ex*. For example:

```
memory   m1
            waitstates 2
            first 0x300
            last 0x5ff
            space prog
            type  RAM attributes  ex

end m1
```

This example allocates a block of RAM that is connected via the external interface. To view this memory within the simulator, you need the following map add command in your siminit.cmd file:

```
ma 0x300, 0, 0x300, EX|RAM, 2
```

When connecting to external memory, be aware of the following limitations:

- ❏ You cannot configure external memory in the same range as any internal memory. The simulator issues an error message when this occurs.
- ❏ The memory you configure must be large enough to accommodate the changes you make to the PSTRT, PEND, DSTRT, and DEND external interface configuration registers.
- ❏ Even though you can specify wait states in the memory configuration file, if you have not set the xready bit in the timing register, the external interface acts as if the memory has no wait states.

  If the timing register fields Setup, Active, and Hold are all 0, then the xready bit cannot be set. For slow memories, either Setup or Active fields must be nonzero.

Executing a memory configuration batch file

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

- ❏ Use the File→Load→Memory Config menu option from within the debugger environment.
- ❏ Use the –cnf debugger option to specify the batch file when you invoke the debugger. For more information, see page 2-9.

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

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

2) If you do not use the –cnf option, the debugger looks for the default configuration batch file (sim.cnf). If the debugger finds the sim.cnf file, it reads and executes the file.

3) If the default configuration batch file (sim.cnf) is not found, the debugger activates only the main window and the Command window. Only the File→Load→Memory Config and File→Take menu options are enabled. The debugger waits in this state until it receives a valid configuration file via one of these two menu options.

   The debugger assumes that the take filename specified with the File→Take menu option contains a MEMFILE command that identifies the memory configuration filename. The MEMFILE command must be placed before any MA commands, or the memory map commands are ignored.

If the debugger finds the memory configuration file, it reads and executes the file. You can load a new configuration file at any time during a debugging session, but the debugger state is completely reinitialized.

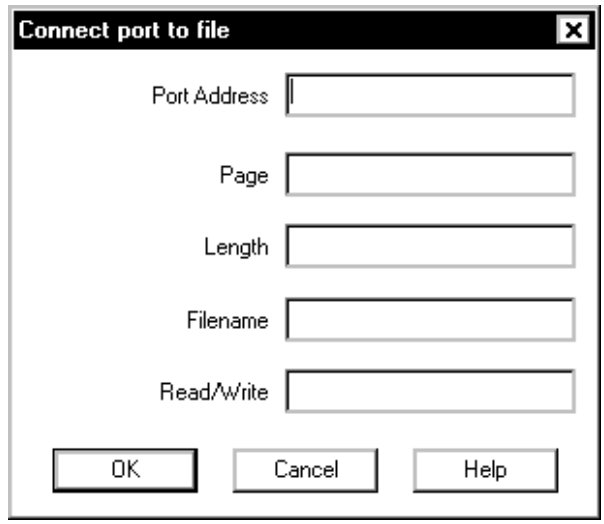
4.10 Connecting a File to a Memory Address (Simulator Only)

In addition to adding memory ranges to the memory map, you can use the Configure→Memory Maps menu option to access a file through a memory address. Then, by connecting to the memory (port) address, the debugger allows you to read data in from a file and/or write data out to a file.

Connecting a file

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

1) Enter the command. This displays the Connecting port to a file dialog box:



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

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

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

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

5) In the Filename field, enter the name of the file to be connected to the memory address. If you connect the memory address to read from a file, the file must exist, or the connect memory to file action fails.

6) In the Read/Write field, enter how the file will be used (for input or output, respectively).

7) Click Apply.

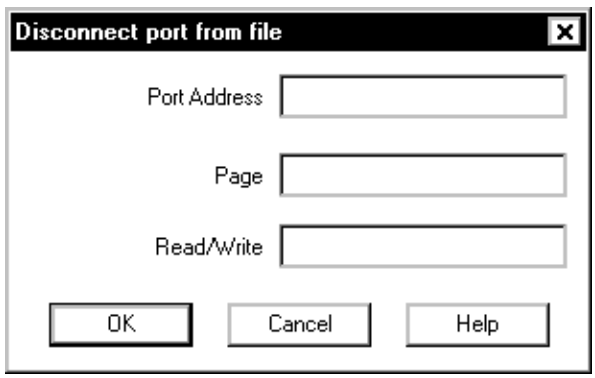
8) Click OK.

The file is accessed during an assembly language read or write of the associated memory address. Any memory address can be connected to a file. A maximum of one input and one output file can be connected to a single memory address; multiple addresses can be connected to a single file.

Disconnecting a file

Before you can delete from the memory map a memory address that you have connected to a file, you must disconnect the address. To disconnect a memory address from an input or output file, follow these steps:

1) Enter the command. This displays the Disconnecting port dialog box:



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

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

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

4.11 Simulating External Interrupts (Simulator Only)

The 'C27xx simulator allows you to simulate and monitor external interrupt signals and to specify at what clock cycle you want an interrupt to occur. To do this, you create a data file and connect it to one of the following external interrupts. The format of the data in the file depends on the signal simulated.

- ❑ $\overline{\text{INT}}$1–$\overline{\text{INT}}$14
- ❑ $\overline{\text{DLOGINT}}$
- ❑ $\overline{\text{RTOSINT}}$
- ❑ $\overline{\text{EMUINT}}$
- ❑ $\overline{\text{NMI}}$

Note:

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

Setting up your input file

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

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

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

  You can have two types of CPU clock cycles:

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

    ```
    12 34 56
    ```

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

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

    ```
    12 +34 55
    ```

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

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

  You can have two forms of repetition to simulate interrupts:

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

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

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

    The n is a positive integer value.

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

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

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

Connecting your input file to the interrupt pin

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

pinc *pinname, filename*

- The *pinname* identifies the input pin and must be one of the following pins:
  - $\overline{\text{INT}}1$–$\overline{\text{INT}}14$
  - $\overline{\text{DLOGINT}}$
  - $\overline{\text{RTOSINT}}$
  - $\overline{\text{EMUINT}}$
  - $\overline{\text{NMI}}$
- The *filename* is the name of your input file.

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

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

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

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

```
12 34 56 89
```

To connect the input file to the pin, enter:

```
pinc int2,  myfile
```

Connects your data file to the specific interrupt pin

This command connects myfile to theINT2 pin. As a result, the simulator generates an external interrupt onINT2 at the 12th, 34th, 56th, and 89th clock cycles.

Disconnecting your input file from the interrupt pin

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

pind *pinname*

The *pinname* parameter identifies the interrupt pin and must be one of the following pins:

- ❏ $\overline{\text{INT}}$1–$\overline{\text{INT}}$14
- ❏ $\overline{\text{DLOGINT}}$
- ❏ $\overline{\text{RTOSINT}}$
- ❏ $\overline{\text{EMUINT}}$
- ❏ $\overline{\text{NMI}}$

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

Listing the interrupt pins and connecting input files

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

pinl

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

Chapter 5

Loading and Displaying Code

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

Topic **Page**

5.1 Loading and Displaying Assembly Language Code

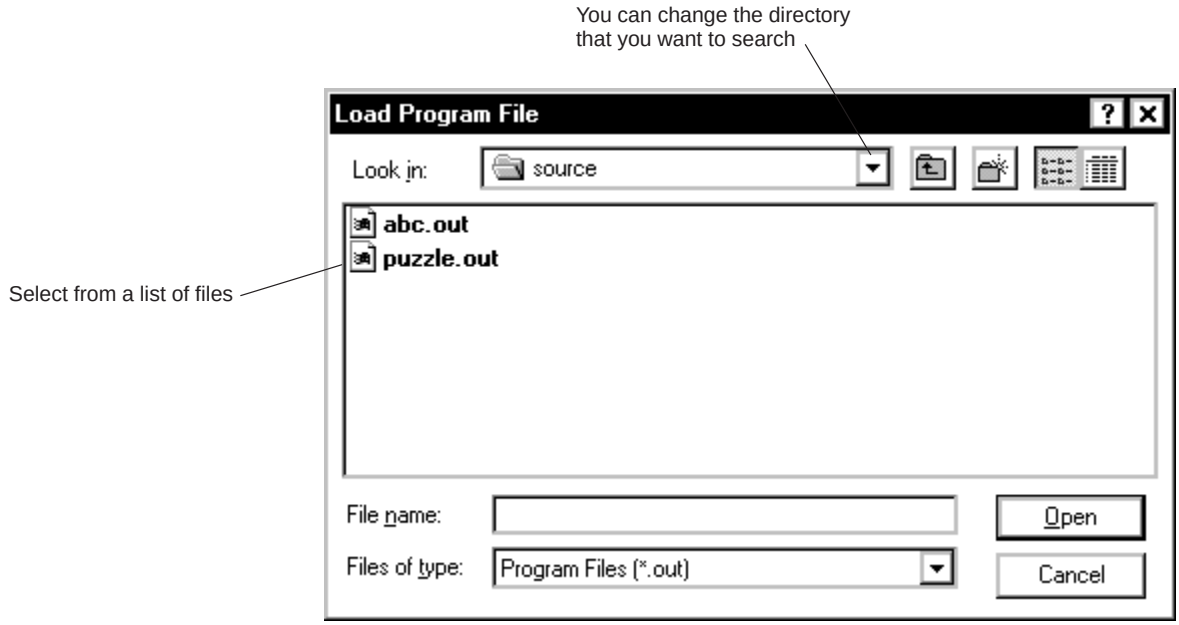
To debug a program, you must load the program's object code into memory. You create an object file by compiling, assembling, and linking your source files; see section 2.1, *Preparing Your Program for Debugging*, on page 2-2.

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

Loading an object file and its symbol table

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

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



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

3) Click Open.

Loading an object file without its symbol table

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

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

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

reload *object filename*

Loading a symbol table only

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

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

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

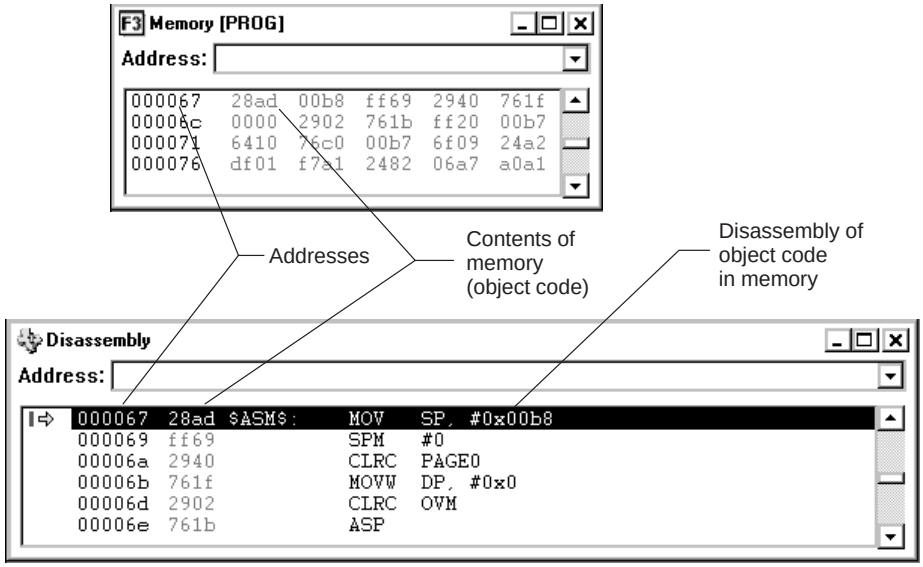
Loading code while invoking the debugger

You can load an object file when you invoke the debugger. (This has the same effect as using the File→Load→Load Program menu option described on page 5-2.) To do this, enter the appropriate debugger-invocation command along with the name of the object file.

If you want to load only a file's symbol table when you invoke the debugger, use the –s option. (This option has the same effect as using the File→Load→Program Symbols menu option.) To do this, enter the appropriate debugger-invocation command along with the name of the object file and specify –s (see page 2-10 for more information).

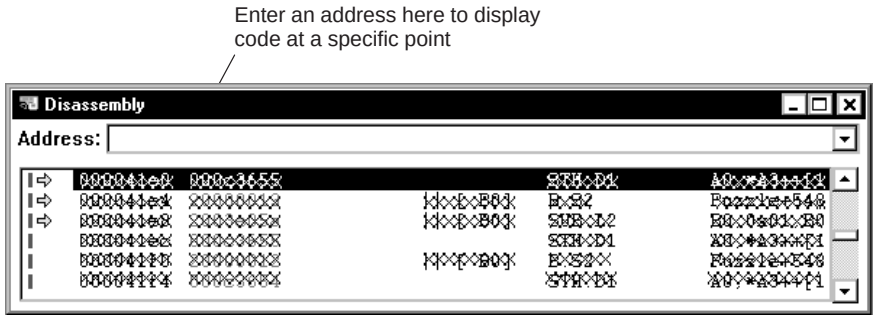
Displaying portions of disassembly

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.



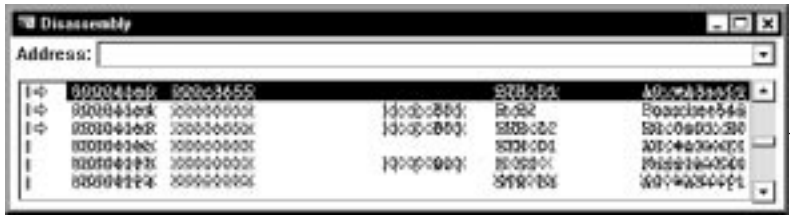
When you invoke the debugger, it comes up in auto mode. If you load an object file when you invoke the debugger, the Disassembly window displays the reverse assembly of the object file that is loaded into memory. If you do not load an object file, the Disassembly window shows the reverse assembly of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

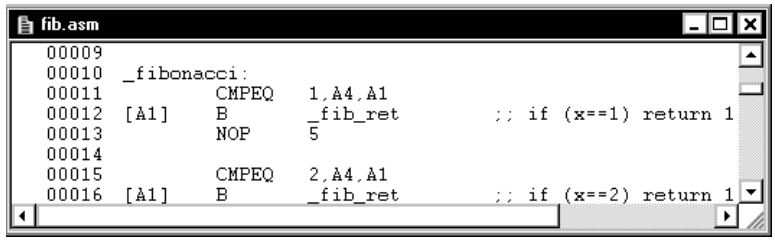
You can also move through the contents of the Disassembly window by using the scroll bar. Because the Disassembly window shows the reverse assembly of memory contents, the scroll bar handle is displayed in the middle of the scroll bar. The middle of the reverse assembly is defined as the most recent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



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

Displaying assembly source code

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



5.2 Displaying Pipeline Phases With Assembly Language Code

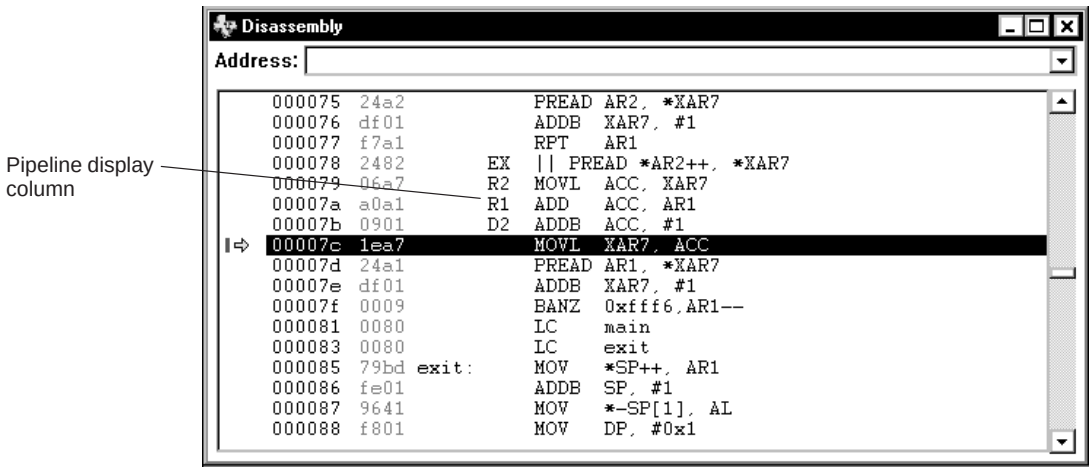
As you first write your code, you do not need to consider pipeline conflicts. Just get the code to work correctly. Then, to find the pipeline conflicts in the critical regions of your code, use the simulator to cycle-step with the pipeline display enabled. Locating the second instruction in a pipeline conflict helps you to identify where you can rewrite your code to avoid or minimize cycles caused by pipeline conflicts.

The simulator enables you to view the current pipeline phase corresponding to a disassembled instruction through the pipeline display feature in the Disassembly window. You can view the pipeline display while single-stepping or cycle-stepping through your code. Cycle-stepping is recommended because if a pipeline stall is detected during a cycle-step operation, the reason for the stall is displayed. Pipeline stall messages do not appear while single-stepping.

The pipeline phase determines the contents of two critical registers. The instruction counter (IC) register contains the address of the instruction in the decode 1 phase. The program counter (PC) register contains the address of the instruction in the decode 2 phase. The current instruction is defined to be the instruction indicated by IC.

Enabling the pipeline display

To display the pipeline information, select Pipeline Display from the Disassembly window context menu. This adds an extra column to the Disassembly window, to the left of the disassembled instructions. Here is an example of the pipeline display:



The pipeline information column indicates the phase of the pipeline that instructions are currently in. The pipeline phases that can be viewed are:

- ❑ Decode 2 D2
- ❑ Read 1 R1
- ❑ Read 2 R2
- ❑ Execute EX
- ❑ Write W

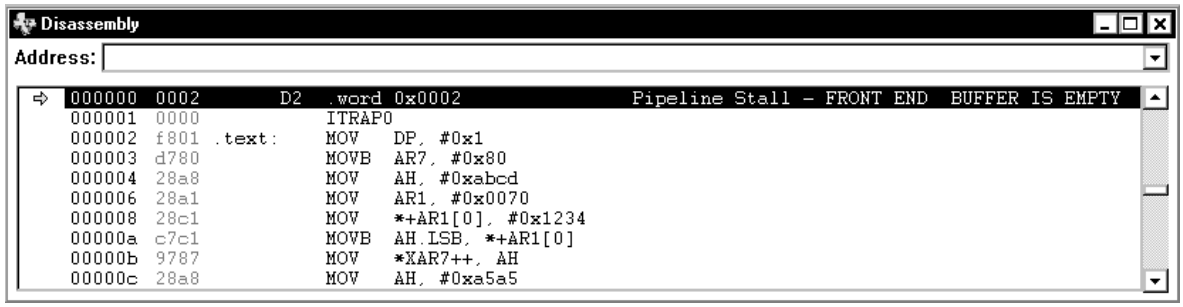
The Fetch 1, Fetch 2, and Decode 1 phases of the pipeline are not viewable in the pipeline display. These phases are difficult for the simulator to display meaningfully. Pipeline conflicts cannot be caused by these phases, which means that displaying them does not help you to identify pipeline conflicts. Only instructions that reach Decode 2 complete.

See the *TMS320C27xx DSP CPU and Instruction Set Reference Guide* for more information about the pipeline.

Cycle-stepping with the pipeline phases displayed

When the simulator is running in the simulation execution mode, the cycle-step command is provided for use with the pipeline display feature. Cycle-stepping your assembly code steps and redisplays information cycle by cycle. This enables you to watch the code step through certain phases of the pipeline and to locate any pipeline conflicts.

If a stall is detected during cycle-stepping, a statement that describes the reason for the stall is shown immediately following the instruction that is in the Decode 2 phase of the pipeline. Here is an example of a pipeline stall message:



Due to the stall, the stalled instruction cannot execute the Decode 2 phase of the pipeline. The simulator waits for the resource to free up in order to execute the instruction through Decode 2.

For information on cycle-stepping, see section 6.4, *Cycle-Stepping Through Assembly Language Code*, on page 6-11. For information on simulation mode, see section 2.9, *Execution Modes*, on page 2-17.

5.3 Displaying C Code

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

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

Displaying the contents of a text file

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

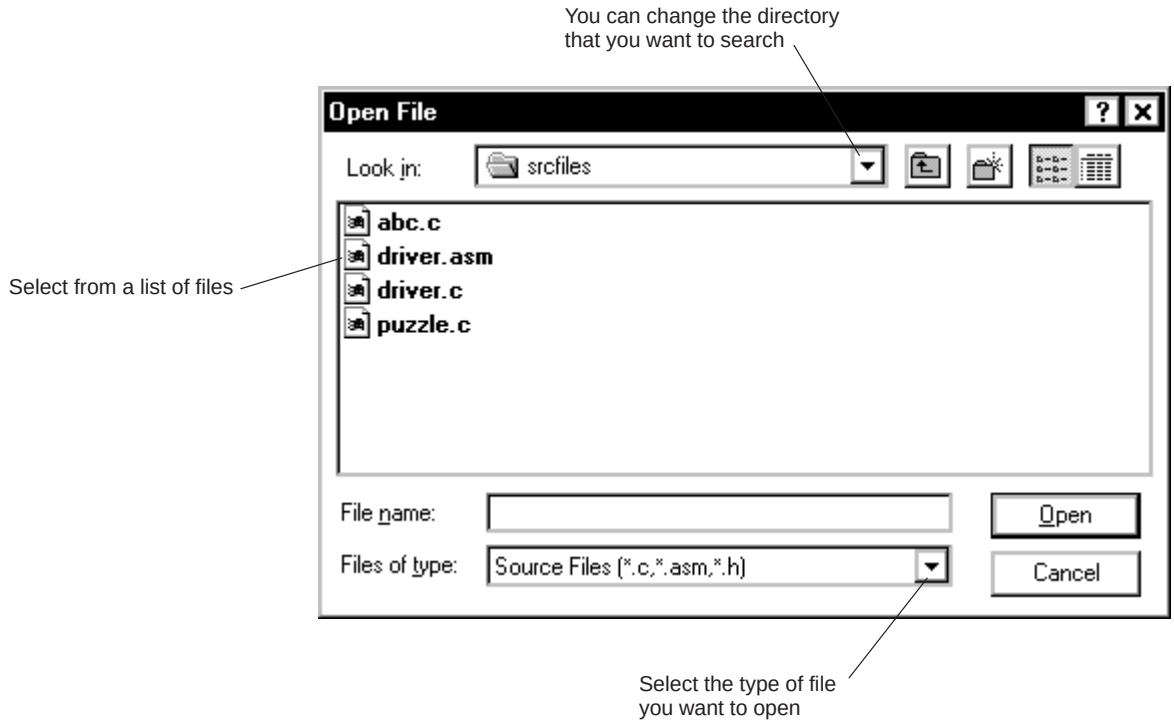
1) Use one of these methods to open the Open File dialog box:

   ❏ Click the Open icon on the toolbar:

   ❏ From the File menu, select Open→Source File.

   This displays the Open File dialog box:



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

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

3) Click Open.

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

Displaying a C file *does not* load that file's object code. If you want to be able to run the program, you must load the file's associated object code as described in section 5.1, *Loading and Displaying Assembly Language Code*, on page 5-2.

Displaying a specific C function

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

func {*function name* | *address*}

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

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

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

Displaying code beginning at a specific point

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

addr {*address* | *function name*}

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

Chapter 6

Running Code

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

- Basic *run commands* run your code without updating the display until you explicitly halt execution.
- *Single-step* commands execute assembly language or C code one statement at a time and update the display after each execution.

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

Topic **Page**

6.1 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

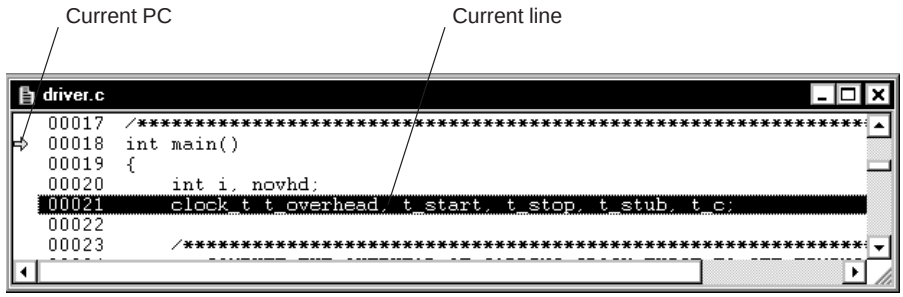
Sometimes you may want to modify the PC to point to a different position in your program. Choose one of these methods:

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

  Alternatively, you can select Restart from the Debug menu.

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



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

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

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

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

- ❑ Modify the value of the PC in the CPU window. (For more information about changing values the displayed in the CPU window, see section 7.3, *Basic Methods for Changing Data Values*, on page 7-5.)

6.2 Using the Basic Run Commands

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

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

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

Running an entire program

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

- ❏ Click the Run icon on the toolbar:

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

  run [*expression*]

  If you supply a logical or relational *expression*, the RUN command becomes a conditional run (see section 6.5 on page 6-12).

  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 6.5 on page 6-12.)
- ❏ You click the Halt icon on the toolbar:

- ❏ You select Halt! from the Debug menu.
- ❏ You press ESC.

Running code up to a specific point in a program

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

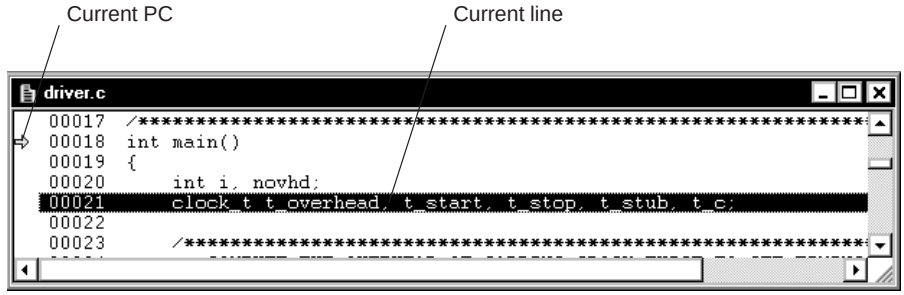
go [*address*]

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

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

- ❏ You select Halt! from the Debug menu.
- ❏ You press ESC.

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



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

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

Running the code in the current C function

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

- ❏ Click the Return icon on the toolbar:

- ❏ From the Debug menu, select Return.

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

- ❏ Click the Halt icon on the toolbar:

- ❏ From the Debug menu, select Halt!.
- ❏ Press ESC.

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

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

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

Running code through breakpoints

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

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

To halt a continuous run, use one of the methods described in section 6.7 on page 6-14.

Resetting the simulator

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

Resetting the emulator

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

6.3 Single-Stepping Through Code

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

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

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

- If you do not supply an *expression*, the program executes a single statement, then halts.
- If you supply a logical or relational *expression*, this becomes a conditional single-step execution (see section 6.5 on page 6-12).
- If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger single-steps *count* assembly language statements unless you are currently in C code. If you are currently in C code, the debugger single-steps *count* C statements.

Single-stepping through assembly language or C code

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

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

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

- ❑ Click the Step icon on the toolbar:

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

  step [*expression*]

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

For more information about the compiler's –g option, see the *TMS320C27xx Optimizing C Compiler User's Guide*.

Single-stepping through C code

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

- ❑ Click the Single Step C icon on the toolbar:

- ❑ From the Debug menu, select Step C.
- ❑ Press CONTROL F8.
- ❑ From the command line, enter the CSTEP command. The format for this command is:

  cstep [*expression*]

Continuously stepping through code

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

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

If no breakpoints are set, you can halt a continuous step by using one of the methods described in section 6.7 on page 6-14.

Single-stepping through code and stepping over C functions

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

- To use the next single-step command and single-step in terms of assembly language or C statements (similar to the basic single-step command), choose one of these methods:
  - Click the Next Statement icon on the toolbar:

  - From the Debug menu, select Next.
  - Press F10 .
  - From the command line, enter the NEXT command. The format for this command is:

    next [*expression*]

- To use the next single-step command and single-step in terms of C statements (similar to the C single-step command), choose one of these methods:
  - Click the Next C Statement icon on the toolbar:

  - From the Debug menu, select Next C.
  - Press CONTROL F10 .
  - From the command line, enter the CNEXT command. The format for this command is:

    cnext [*expression*]

6.4 Cycle-Stepping Through Assembly Language Code

For cycle-step execution, the simulator executes one cycle, updates the display, and halts execution. (You can supply a parameter that tells the simulator to cycle-step continuously; the simulator updates the display after each statement is executed.) You can cycle-step only through assembly language code.

The cycle-step command has an optional *expression* parameter that works like this:

- ❑ If you do not supply an *expression*, the program executes a single cycle, then halts.
- ❑ If you supply a logical or relational *expression*, this becomes a conditional single-step execution (see section 6.5 on page 6-12).
- ❑ If you supply any other type of *expression*, the simulator treats the expression as a *count* parameter. The simulator cycle-steps *count* assembly language statements.

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

- ❑ Click the Single Clock (Cycle) Step icon on the toolbar:

- ❑ From the Debug menu, select Clock Step.
- ❑ Press F7.
- ❑ From the command line, enter the STEPCYCLE command. The format for this command is:

  stepcycle [*expression*]

Cycle-stepping through code is most useful with the pipeline display enabled. See section 5.2, *Displaying Pipeline Phases With Assembly Language Code*, on page 5-6, for more information.

Note:

Cycle-step is available only in simulation mode. If you are using the emulation execution mode, cycle-step is disabled. See section 2.9, *Execution Modes*, on page 2-17 for information on how the simulator behaves in the two execution modes.

6.5 Running Code Conditionally

The RUN, STEP, CSTEP, STEPCYCLE, 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 6.8 on page 6-15.) 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.

6.6 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 6.8 on page 6-15.)

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

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

Step 4: From the Debug menu, select Run Benchmark.

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

Notes:

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

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

3) When programming in C, avoid using a variable named CLK.

4) The RUNB command accesses the analysis module to count CPU clock cycles. If you have set up an instruction breakpoint, the debugger halts on that breakpoint in addition to your software breakpoints.

6.7 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 Debug menu, select Halt!.
- Press ESC.

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

What happens when you halt the emulator

If you are using the emulator version of the debugger, any of the above methods halts the target system after you have commanded the debugger to run code while disconnected from the target (run free).

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

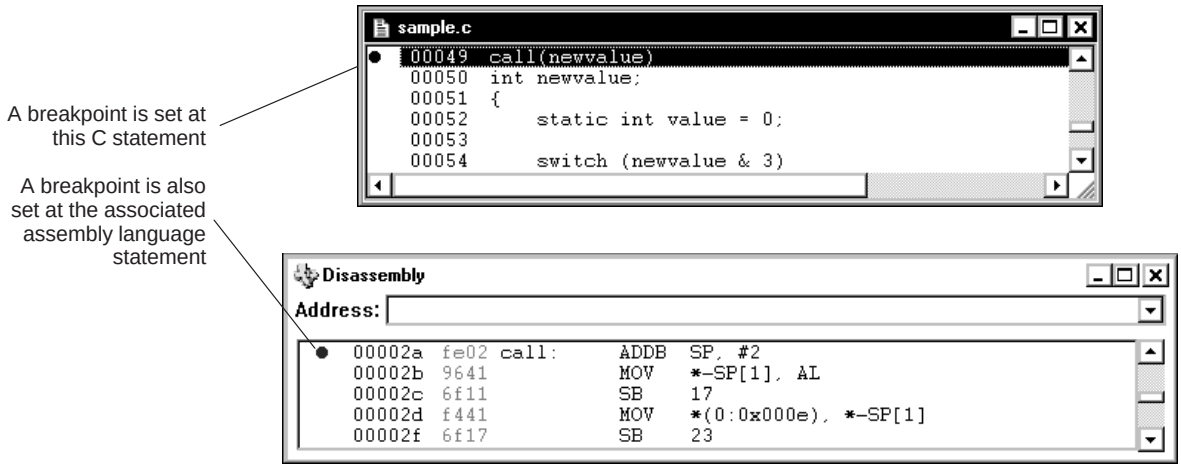
6.8 Using Software Breakpoints

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

Software breakpoints are especially useful in combination with conditional execution (described in section 6.5 on page 6-12).

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:

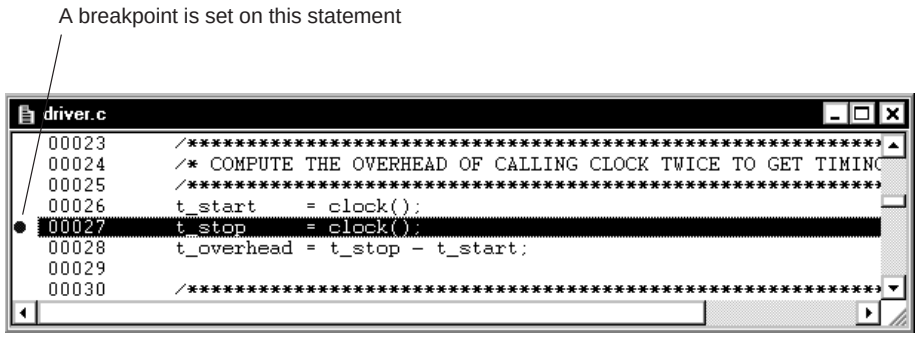
1) After execution is halted by a breakpoint, you can continue program execution by reissuing any of the run or single-step commands.

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



Another way to set a breakpoint is to use the context menu for the File or Disassembly window. You can set a breakpoint on the current line in the File or Disassembly window. The current line is highlighted in the display.

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

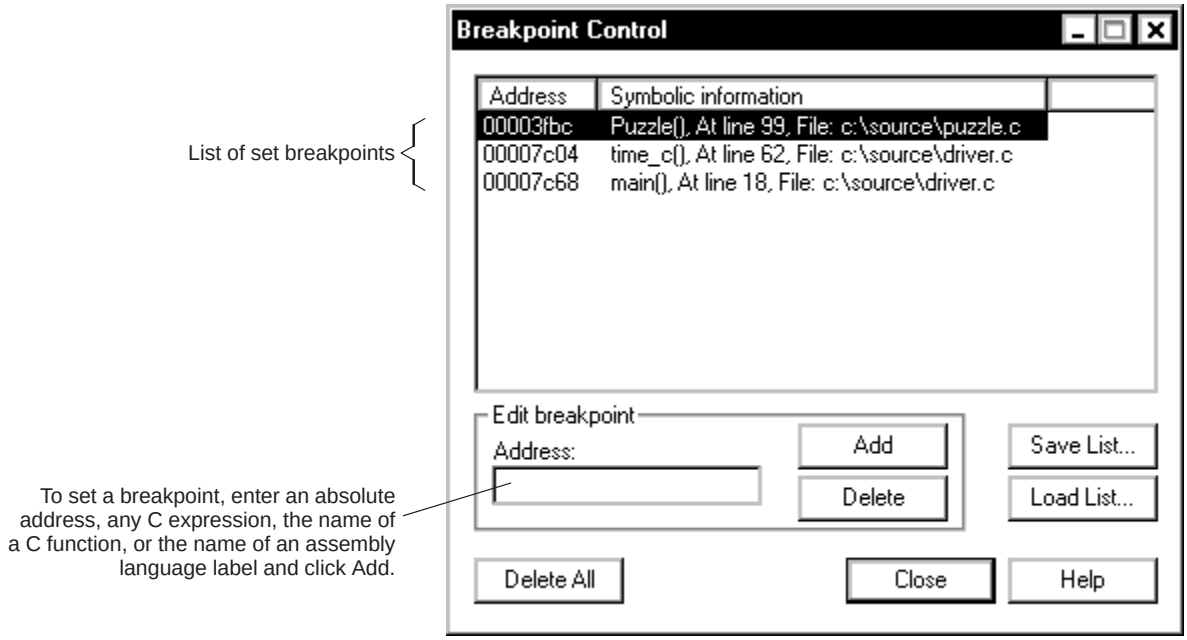
- ❏ Open the context menu for the window. (For more information, see page 1-6.)
- ❏ Select Toggle Breakpoint from the context menu.

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

- ❏ Click the Breakpoint Dialog icon on the toolbar:

- ❏ From the Configure menu, select Breakpoints.

This displays the Breakpoint Control dialog box:



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 from which you want to remove the breakpoint.
2) From the context menu for the window, select Toggle Breakpoint.

You can also clear a breakpoint by using the Breakpoint Control dialog box (see the illustration on page 6-17):

1) Open the Breakpoint Control dialog box by using one of these methods:
   - ❑ Click the Breakpoint Dialog icon on the toolbar:

   - ❑ From the Configure menu, select Breakpoints.
2) Select the address of the breakpoint that you want to clear.
3) Click Delete. The breakpoint is removed from the breakpoint list.
4) Click Close to close the Breakpoint Control dialog box.

Clearing all software breakpoints

To clear all software breakpoints, follow these steps:

1) Open the Breakpoint Control dialog box by using one of these methods:
   - ❑ Click the Breakpoint Dialog icon on the toolbar:

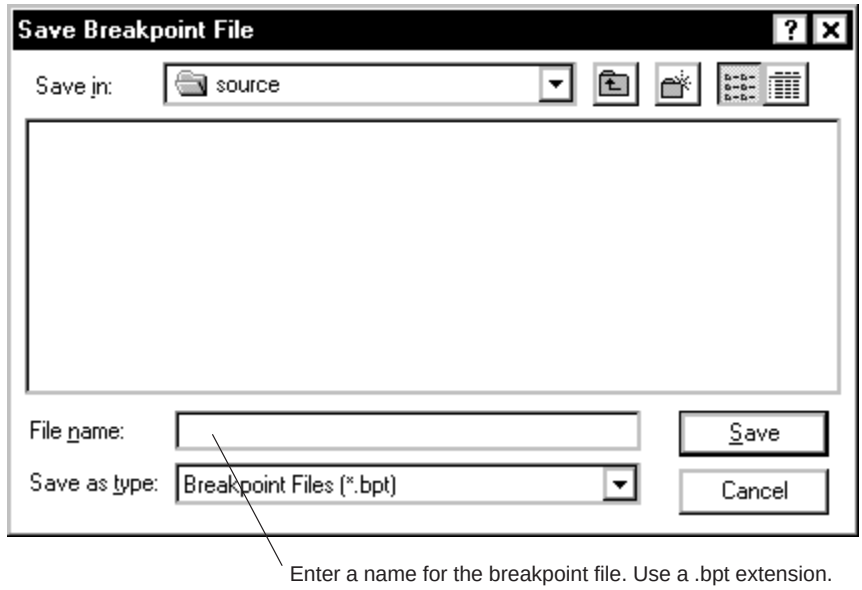
   - ❑ From the Configure menu, select Breakpoints.
2) Click Delete All.
3) Click Close to close the Breakpoint Control dialog box.

Saving breakpoint settings

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

1) Open the Breakpoint Control dialog box by using one of these methods:
   - ❑ Click the Breakpoint Dialog icon on the toolbar:

     
   - ❑ From the Configure menu, select Breakpoints.
2) Click Save List. This displays the Save Breakpoint File dialog box:



3) Select the directory where you want the file to be saved.
4) In the File name field, enter a name for the breakpoint file. You can use a .bpt extension to identify the file as a breakpoint file.
5) Click Save.
6) In the Breakpoint Control dialog box, click Close.

Notes:

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

Loading saved breakpoint settings

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

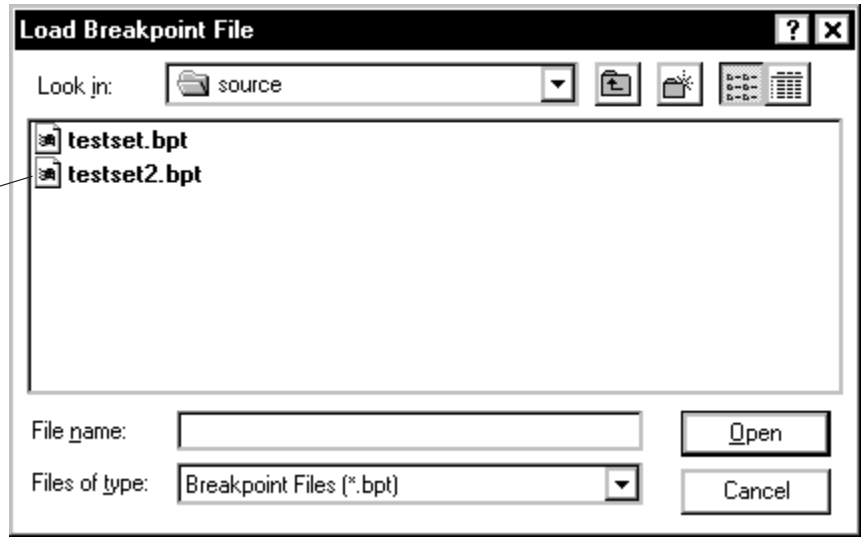
1) Open the Breakpoint Control dialog box by using one of these methods:

   ❑ Click the Breakpoint Dialog icon on the toolbar:

   ❑ From the Configure menu, select Breakpoints.

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



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.

Chapter 7

Managing Data

The debugger allows you to examine and modify many types of data related to the ’C27xx 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 ’C27xx registers
- Variables, including scalar types (ints, chars, etc.) and aggregate types (arrays, structures, etc.)

Topic **Page**

7.1 Where Data Is Displayed

Various types of data are displayed in one of four dedicated windows.

Type of Data	Window Name	Purpose
Memory locations	Memory window	Displays the contents of a range of memory
Register values	CPU window	Displays the contents of 'C27xx registers
Pointer data, variables, aggregate types, and specific memory locations or registers	Watch window	Displays selected data
Variable values	Variable window	Displays values for variables in the current or selected function

The four dedicated windows are referred to as *data-display windows*.

7.2 Basic Commands for Managing Data

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

Determining the type of a variable

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

whatis *symbol*

The *symbol*'s data type is then listed in the display area of the Command window. The *symbol* can be any variable (local, global, or static), a function name, a structure tag, a typedef name, or an enumeration constant.

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

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

Evaluating an expression

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

? *expression*

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

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

Here are some examples that use the ? command.

Command	Result Displayed in the Command Window

`? aai`	`aai[0][0] 1`
	`aai[0][1] 23`
	`aai[0][2] 45`
	`.`
	`.`
	`.`

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

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

eval *expression*
or
e *expression*

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

7.3 Basic Methods for Changing Data Values

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

Editing data displayed in a window

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

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

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

1) Select the data item that you want to modify. Choose one of these methods:
   - ❏ Double-click the data item that you want to modify.
   - ❏ Select the data item that you want to modify and press F9.
2) Type the new information. If you make a mistake or change your mind, press ESC; this resets the field to its original value.
3) When you finish typing the new information, press ↵ or click on another data value. This replaces the original value with the new value.

Editing data using expressions that have side effects

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

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

For example, if you want to see what is in register AR3, you can enter:

```
? AR3 ↵
```

You can also use this type of command to modify AR3's contents. Here are some examples of how you might do this:

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

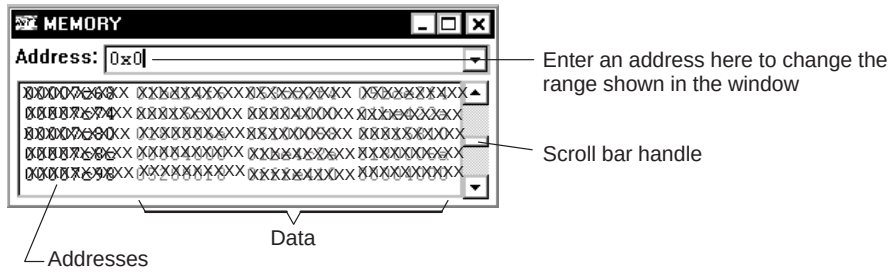
Not all expressions have side effects. For example, if you enter **? AR3+4**, the debugger displays the result of adding 4 to the contents of AR3 but does not modify AR3's contents. Expressions that have side effects must contain an assignment operator or an operator that implies an assignment. Operators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

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

7.4 Managing Data in Memory

The most common way to observe memory contents is to view the display in a Memory window. In mixed and assembly modes, the debugger displays the default Memory window automatically (labeled Memory). You can open any number of additional Memory windows to display different memory ranges. Figure 7–1 shows the default Memory window.

Figure 7–1. 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 7.3, *Basic Methods for Changing Data Values*, on page 7-5.

Changing the memory range displayed in a Memory window

To change the memory range displayed in a Memory window, enter a new starting address in the Address field of the Memory window, as shown in Figure 7–1. 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 7–1). 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 11-28). You can scroll up or down to see 1K bytes of memory on either side of the current starting address.

Opening an additional Memory window

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

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

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

  The *expression* can be an absolute address, a symbolic address, or any C expression. Here are some examples:

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

    ```
    mem 0x0
    ```

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

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

    ```
    mem &SYM
    ```

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

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

    ```
    mem SP – AR0 + label
    ```

- The *display format* parameter is optional. When used, the data is displayed in the selected format, as shown in Table 7–2 on page 7-22.

- Use the *window name* parameter to name the additional Memory window. The debugger appends the *window name* to the Memory window label. If you do not supply a name, the debugger does not open a new window; it simply updates the default Memory window to reflect the changes.

Displaying memory contents while you are debugging C

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

Note:

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

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

  `? *0x26` ↵

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

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

  1) In the Expression field, enter **`*0x26`**.

  2) In the Format combo box, enter x-Hexadecimal.

  3) Click OK.

  The debugger displays the memory value in the Watch window.

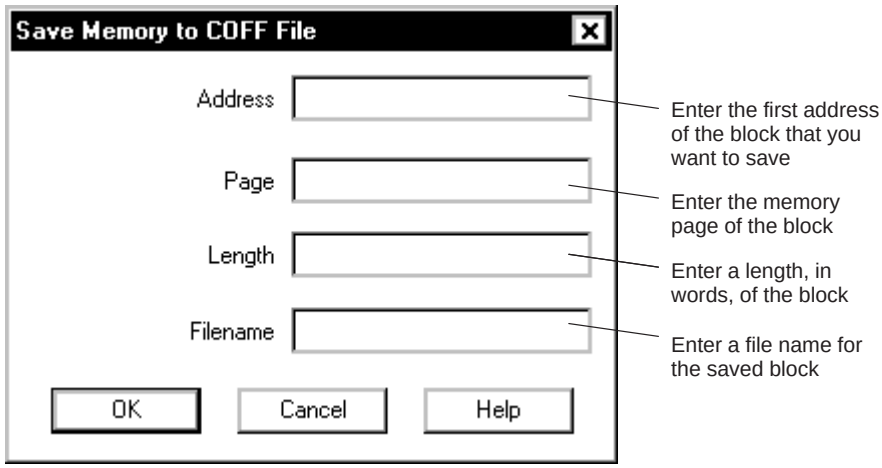
- You can also use the DISP command to display memory contents in a Watch window. The Watch window shows memory in an array format with the specified address as member [0]. In this situation, you can also use casting to display memory contents in a different numeric format:

  `disp *(float *)0x26` ↵

Saving memory values to a file

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

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



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

3) In the Page field, enter a 1-digit number that identifies the type of memory (program or data) that the address occupies:

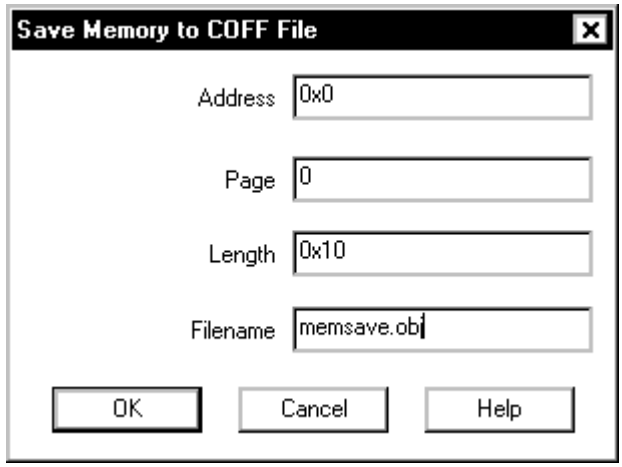
To identify this page . . .	Use this value as the *page* parameter . . .
Program memory	0
Data memory	1

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

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

6) Click OK.

For example, to save the values in data memory locations 0x0000–0x003F to a file named memsave.obj, you could enter:



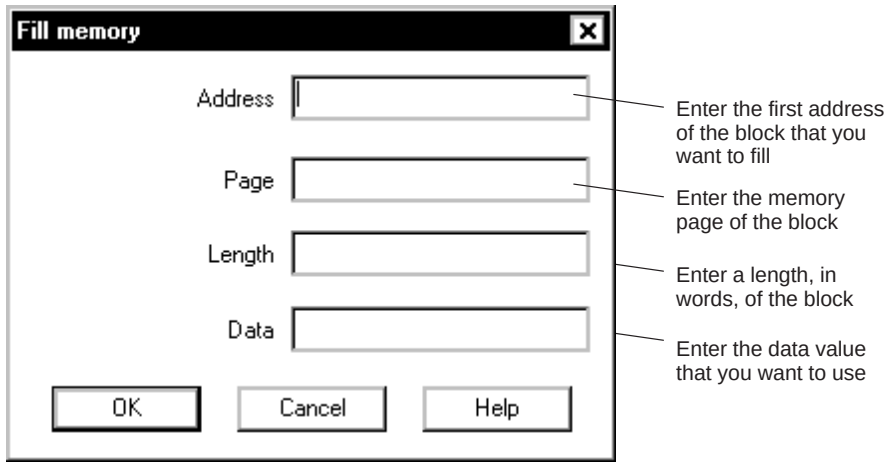
The value of the Length field is measured in words and data memory locations are measured in bytes. For this example, you must enter a length of 10 words to equal 0x40 bytes (0x0000–0x0040).

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

Filling a block of memory

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

1) From the Configure menu, select Memory Fill→Fill Word. This displays the Fill Memory 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 Page field, enter a 1-digit number that identifies the type of memory (program or data) that the address occupies:

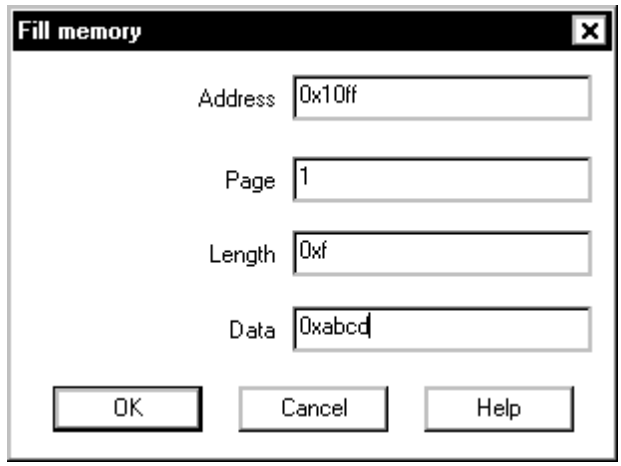
To identify this page . . .	Use this value as the *page* parameter . . .
Program memory	0
Data memory	1

4) In the Length field, enter a length, in words, of the block.

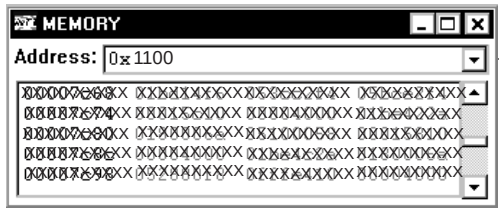
5) In the Data field, enter a value that you want placed in each word in the block.

6) Click OK.

For example, to fill memory locations 0x1100–0x113B with the value 0xABCD, you could enter:



If you want to check whether memory has been filled correctly, you can change the Memory window display to show the block of memory beginning at memory address 0x1100:



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

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

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

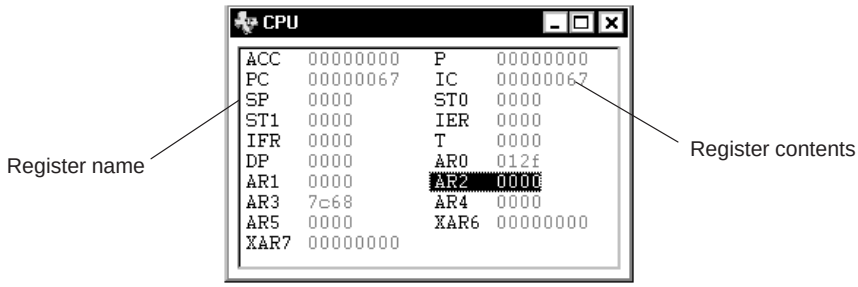
3) In the Length field, enter a length, in bytes, of the block.

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

5) Click OK.

7.5 Managing Register Data

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



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 7.3, *Basic Methods for Changing Data Values*, on page 7-5.

Displaying register contents

The main way to observe register contents is to view the display in the CPU window. You can rearrange the registers in the CPU window to display the ones that you are most interested in at the top of the CPU window. To do so, drag and drop the registers to the desired location in the CPU window.

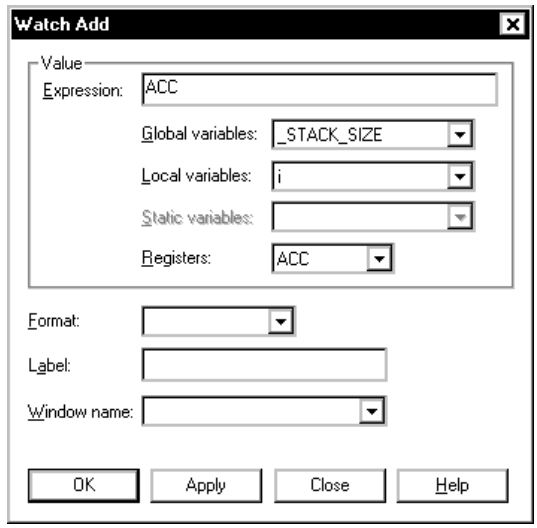
In addition to the CPU window, you can observe the contents of selected registers by using the ? (evaluate expression) command or Configure→Watch Add menu option:

- If you have only a temporary interest in the contents of a register, you can use the ? command to display the register's contents. For example, if you want to know the contents of AR3, you could enter:

  ? AR3 ↵

  The debugger displays AR3'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 Configure→Watch Add to display the register in a Watch window. For example, if you want to observe the accumulator register (ACC), you could enter:



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

When you are debugging C in auto mode, the ? command and the Configure→Watch Add menu option are useful because the debugger does not show the CPU window in the C-only display.

- spsr_fiq

Viewing the bit fields in 'C27xx machine registers

The debugger allows you to view individual sections or bit fields of certain important registers in a Watch window through predetermined psuedoregisters. The psuedoregisters let you view the contents of the specific register section or bit field directly rather than through a lengthy manual process. Table 7–1 lists the pseudoregister names for each of these registers.

Table 7–1. Pseudoregister Names for TMS320C27xx Registers

'C27xx Register Name	Pseudoregister Name
ACC	ACCH ACCL
P	PH PL
ST0	OVC PM V N Z C TC OVM SXM
ST1	ARP LOOP SPA VMAP PAGE0 DBGM INTM
XAR6	XAR6H AR6
XAR7	XAR7H AR7

You can display the contents of these psuedoregisters with the WA (watch) command or Configure→Watch Add menu option.

See the *TMS320C27xx DSP CPU and Instruction Set Reference Guide* for information on the specific registers, register sections, and bit fields.

External interface psuedoregisters

You can program the external interface peripheral and timing registers with either C code or assembly code or by modifying the simulator psuedoregisters. You can use these psuedoregisters:

- Peripheral psuedoregisters
  - PSTRT_XINTF
  - DSTRT_XINTF
  - PEND_XINTF
  - DEND_XINTF
  - MCTL_XINTF
- Timing psuedodregisters
  - XPTIMING0
  - XPTIMING1
  - XDTIMING0
  - XDTIMING1
  - XDTIMING2
  - XDTIMING3
  - XDTIMING4
  - XINTFCNF0
  - XINTFCNF1
  - XINTFCNF2

To view the external interface psuedoregisters, you must add a line to your siminit.cmd file specifying the WA command. See page 11-49 for information on the WA command.

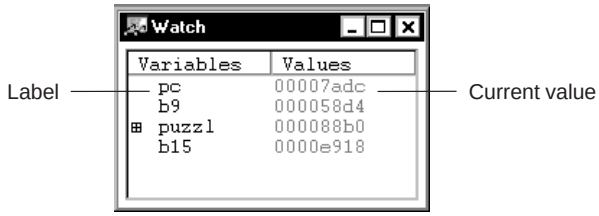
An additional psuedoregister is EMU_ENABLE. This psuedoregister is used to enable changes to the peripheral psuedoregisters during reset. The timing psuedoregisters can be changed without modifying EMU_ENABLE. This example sets the value of PSTRT_XINF to 0x300:

```
load sample.out
reset
? EMU_ENABLE = 1
? PSTRT_XINTF = 0x300
```

The behavior of the ’C27xx device is not defined in the case of changing the buffer depth while a write is pending from the external interface. If you issue a write to external ROM, you do not see an error message.

7.6 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.



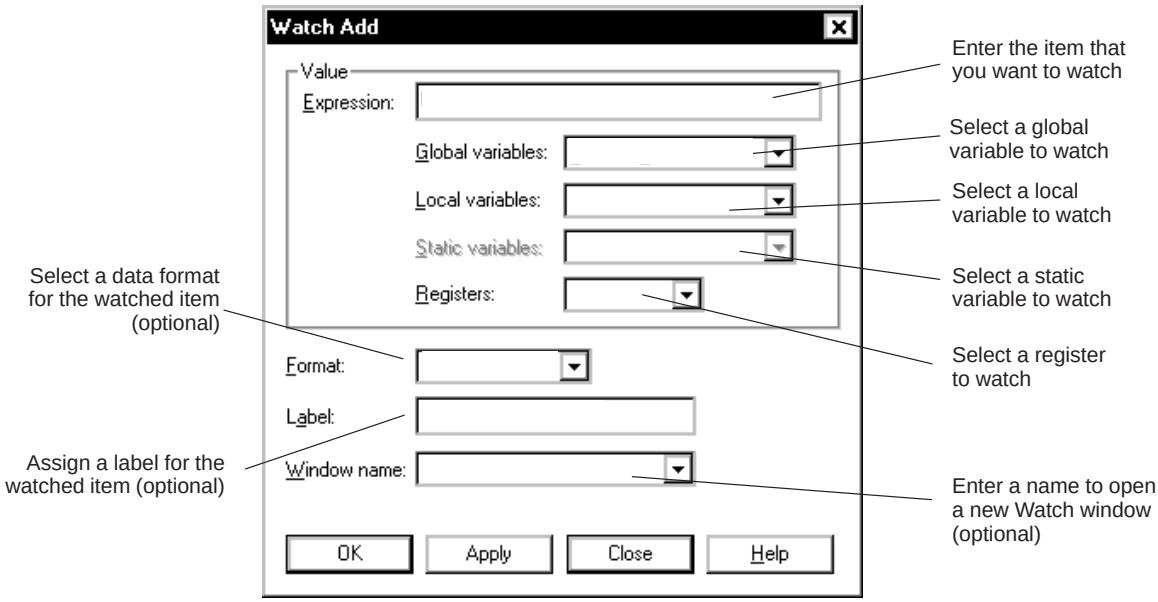
The debugger displays a Watch window *only when you specifically request a Watch window*.

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 7.3, *Basic Methods for Changing Data Values*, on page 7-5.

Displaying data in a Watch window

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

1) From the Configure menu, select Watch Add. This displays the Watch Add dialog box:



2) In the Expression field, enter the item that you want to watch. The expression can be any C expression, including an expression that has side effects. Also, you can select a global variable, local variable, static variable, or register to watch.

   If you want to use the *contents* of an address as a parameter, be sure to prefix the address with the C indirection operator (*****). For example, you could enter this value in the Expression field:

   ```
   *0x26
   ```

3) If you want to change the data format for the watched item, select the format you want to use from the Format drop list. The format field is optional.

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

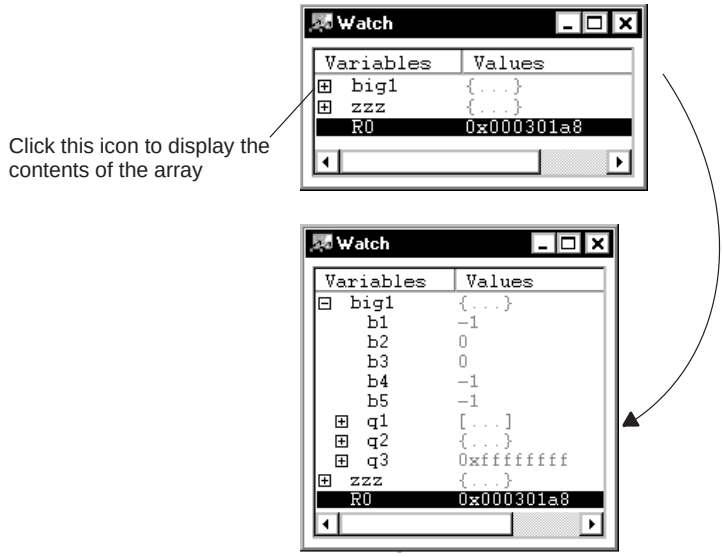
5) If you want to open a new Watch window, enter a name for the new Watch window in the Window name field. This field is optional. When you enter a window name, the debugger appends the window name to the Watch window label. If you do not supply a name, the debugger adds the item to the default Watch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

6) Click Apply. When you have entered the last expression, variable, or register that you want to watch, click OK.

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

Displaying additional data

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



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

Deleting watched values

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

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

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

wr [{***** | *window name*}]

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

Note:

The debugger automatically closes any Watch windows when you execute File→Load→Load Program, File→Load→Program Symbols the LOAD command, or the SLOAD command.

7.7 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 7–2 lists the available formats and the corresponding characters that can be used as the *display format* parameter.

Table 7–2. Display Formats for Debugger Data

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

Table 7–3 lists the C data types that can be used for the *data type* parameter. Only a subset of the display formats applies to each data type, so Table 7–3 also shows valid combinations of data types and display formats.

Table 7–3. Data Types for Displaying Debugger Data

	Valid Display Formats									
Data Type	**c**	**d**	**o**	**x**	**e**	**f**	**p**	**s**	**u**	**Default Display Format**
char	√	√	√	√					√	ASCII (c)
uchar	√	√	√	√					√	Decimal (d)
short	√	√	√	√					√	Decimal (d)
int	√	√	√	√					√	Decimal (d)
uint	√	√	√	√					√	Decimal (d)
long	√	√	√	√					√	Decimal (d)
ulong	√	√	√	√					√	Decimal (d)
float			√	√	√	√				Exponential floating point (e)
double			√	√	√	√				Exponential floating point (e)
ptr			√	√			√	√		Address (p)

Here are some examples:

- To display all data of type short as an unsigned decimal, enter:

  `setf short, u` ↵

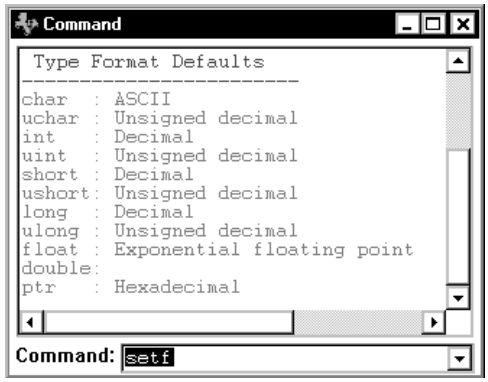
- To return all data of type short to its default display format, enter:

  `setf short, *` ↵

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

  `setf` ↵

The display should look something like this:



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

```
setf * ↵
```

Changing the default format with data-management commands

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

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

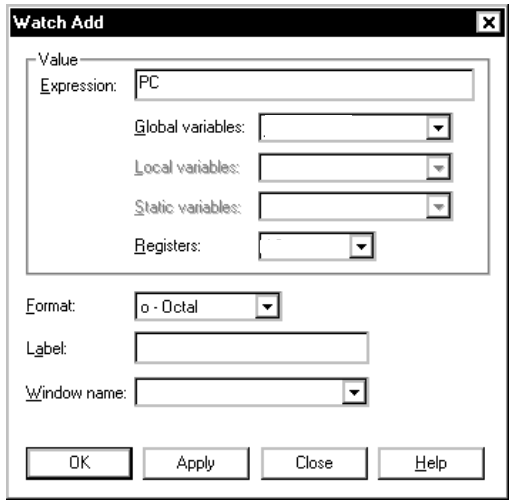
When you do not use a display format parameter, data is shown in its natural format (unless you have changed the format for the data type with SETF).

Here are some examples:

❑ To display memory contents in octal, enter:

```
mem 0x0,o ↵
```

❑ To watch the PC in octal, from the Format drop list, select o-Octal:



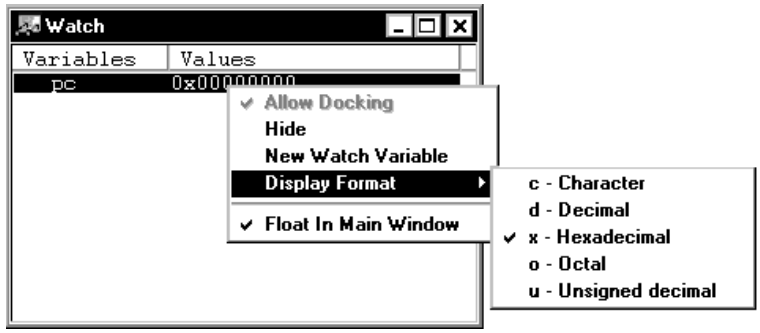
❑ To change the format of the PC in the Watch window, follow these steps:

1) In the Watch window, select PC.

2) Right click the mouse to bring up the Watch window context menu.

3) From the context menu, select Display Format. A submenu of data formats appears.

4) From the submenu, select the format in which you want the PC to display.



The valid combinations of data types and display formats listed for SETF also apply to the data displayed with ?, MEM, Configure→Watch Add, WA, and DISP. For example, if you want to use display format **e** or **f**, the data that you are displaying must be of type float or type double. Additionally, you cannot use the **s** display format parameter with the MEM command.

Chapter 8

Profiling Code Execution

The profiling environment is a special debugger environment that provides a method for collecting execution statistics about specific areas in your code. These statistics give you immediate feedback on your application's performance.

Topic **Page**

8.1 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.

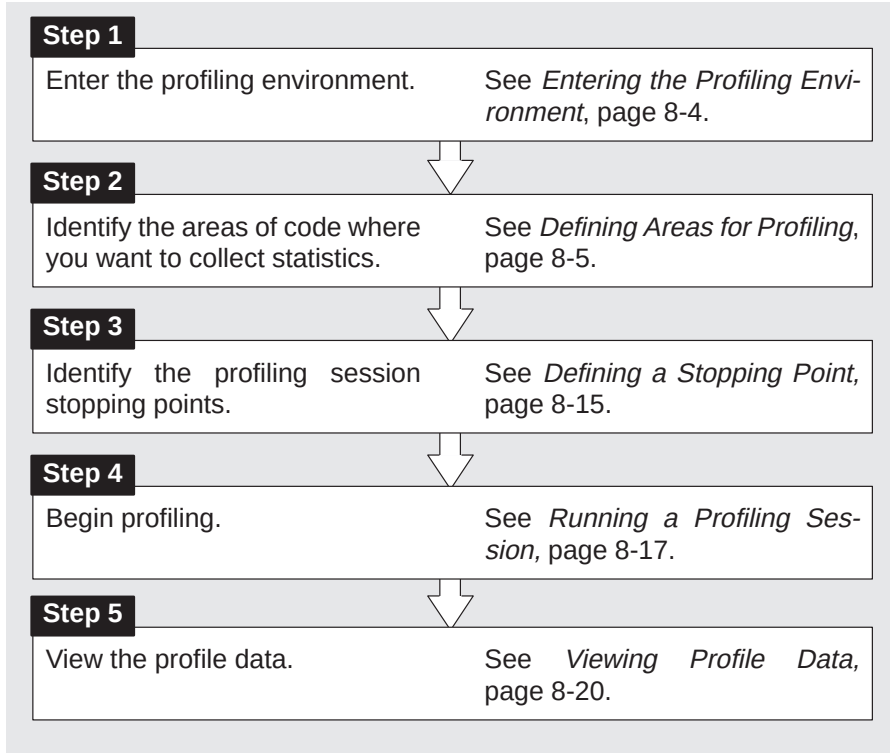
- **Configurable display of statistics.** Display the entire set of data, or display one type of data at a time. Display all the areas you are profiling, or display a selected subset of the areas.

- **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.

8.2 Overview of the Profiling Process

Profiling consists of five simple steps:



Note:

When you compile a program that will be profiled, you must use the –g and the –as compiler shell options. The –g option includes symbolic debugging information; the –as option ensures that you will be able to include ranges as profile areas. For more information on these options, see the TMS320C27xx C Compiler User's Guide.

A profiling strategy

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

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

8.3 Entering the Profiling Environment

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

Some restrictions apply to the profiling environment:

- ❏ The debugger is always in mixed mode.
- ❏ Command, Disassembly, File, and Profile are the only windows available; additional windows, such as a Watch window, cannot be opened.
- ❏ The profiling environment supports only a subset of the debugger commands. Table 8–1 lists the debugger commands that can and cannot be used in the profiling environment.

Table 8–1. Debugger Commands That Can/Cannot Be Used in the Profiling Environment

Can be used	Cannot be used
Data-evaluation commands (such as ? and EVAL) Breakpoint commands Memory-mapping commands System commands (such as SYSTEM, TAKE, and ALIAS) Windowing commands (such as SIZE, MOVE, and ZOOM)	All run commands Debugging mode commands (such as ASM, C, and MIX) Commands related to the Watch, Memory, or Calls window

Chapter 11, *Summary of Commands*, summarizes all of the debugger commands and tells you whether a command is valid in the profiling environment.

8.4 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 8-12.

Marking an area with a mouse

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

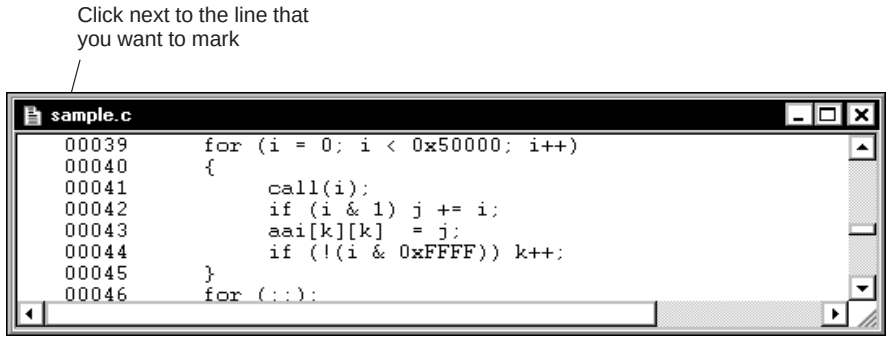
Remember, to display C code, use the File→Open menu option or the FUNC command; to display disassembly, use the DASM command.

Notes:

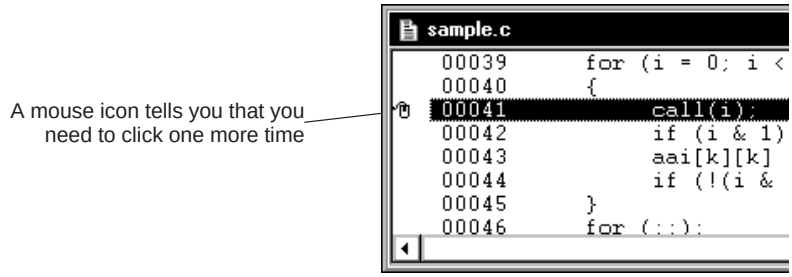
1) Marking an area in C *does not* mark the associated code in disassembly.
2) Areas can be nested; for example, you can mark a line within a marked range. The debugger reports statistics for both the line and the function.
3) Ranges cannot overlap, and they cannot span function boundaries.

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

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

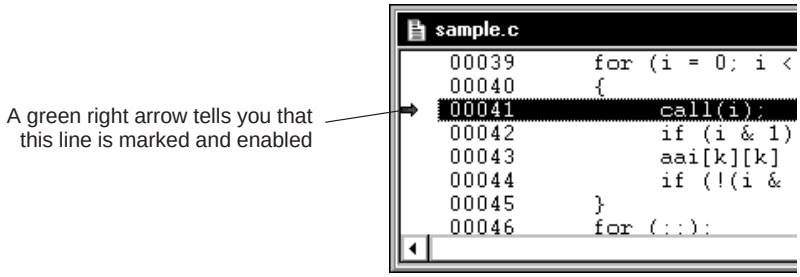


When you click once next to a line, a mouse icon appears, telling you that you need to click one more time:

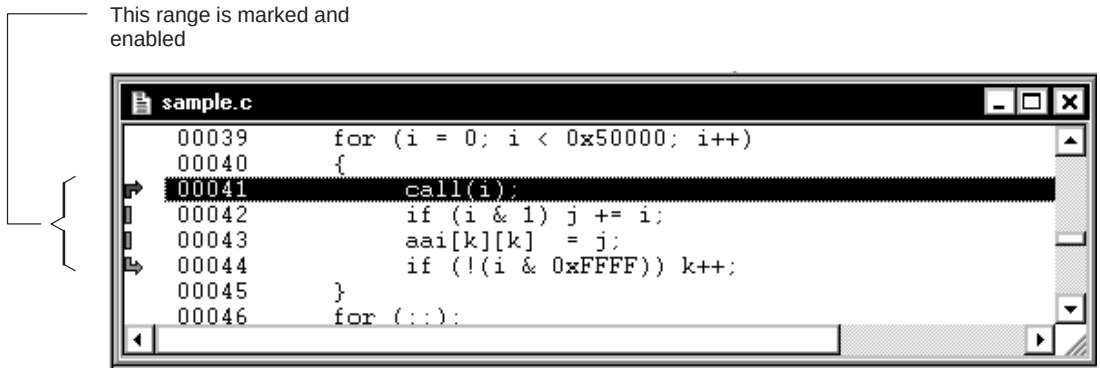


2) Choose to mark a single line or a range:

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



- To mark a range, click the last line of the range that you want to mark. This changes the mouse icon on the first line of the range into a green arrow. The entire range is marked with two green right arrows that are connected:



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

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

2) When you see the mouse icon, click again to mark and enable the C function. A green arrow appears, indicating that the function is marked.

Note:

In Profile mode, if you try to mark a line or function by double-clicking next to the statement that you want to mark, the debugger sets a software breakpoint instead of marking the line or function. To mark a function, click once. If you are marking a line and you see the mouse icon, click again.

If you are not in profile mode, single-clicking next to a line or function sets a software breakpoint.

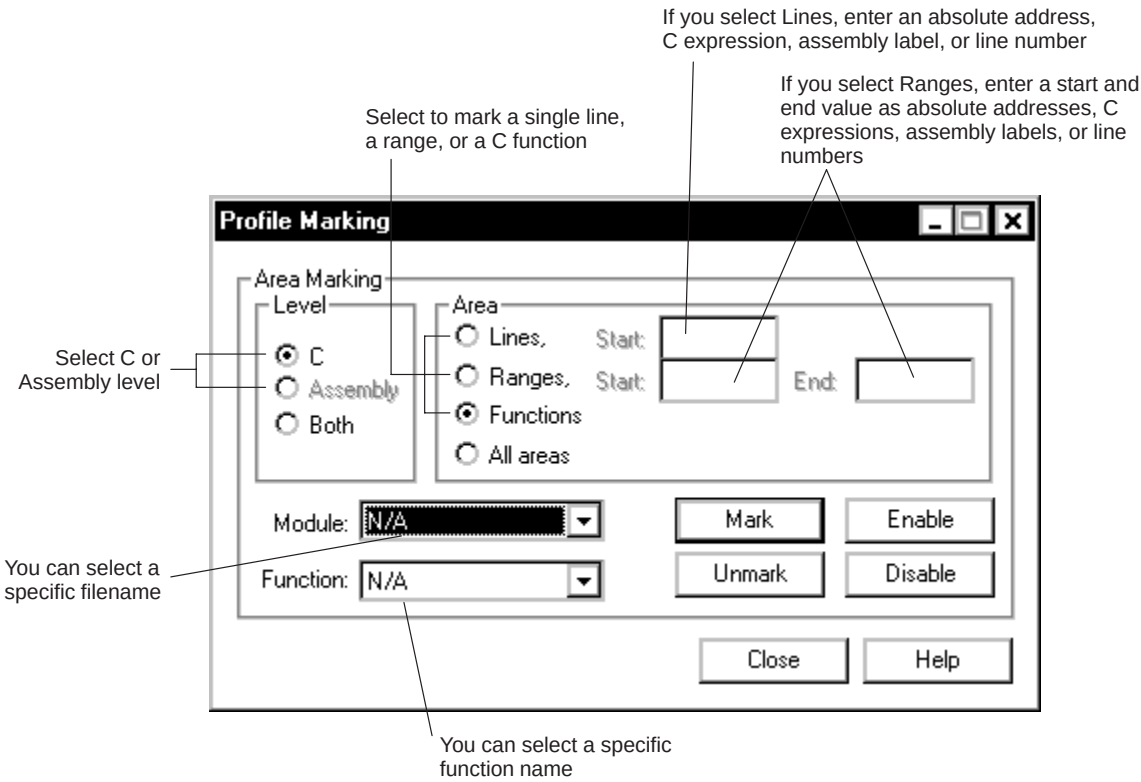
Marking an area with a dialog box

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

1) Open the Profile Marking dialog box by using one of these methods:

   ❏ From the Tools→Profile menu, select Select Areas.
   ❏ From the context menu for the Profile window, select Select Areas.

   This displays the Profile Marking dialog box:



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

3) In the Area box, select Lines, Ranges, or Functions. See Table 8–2 for a list of valid combinations.

4) Depending on what you select in step 3, do one or more of the following:

   ❏ Next to Lines, enter an absolute address, C expression, assembly label, or line number. If you are entering an absolute address, be sure to prefix it with 0x.

   ❏ Next to Ranges, enter a Start and an End value as absolute addresses, C expressions, assembly labels, or line numbers.

   ❏ From the Module combo box, select a specific filename.

   ❏ From the Function combo box, select a specific function name.

   See Table 8–2 for a list of valid combinations.

5) Click Mark.

6) Click Close to close the dialog box.

Table 8–2. Using the Profile Marking Dialog Box to Mark Areas

(a) Marking lines

To mark this area...	If C level is selected...	If Assembly level is selected...
By line number, address	❏ Select a module name. ❏ In the Area box, select Lines. ❏ Next to Lines, specify a line number.	❏ In the Area box, select Lines. ❏ Next to Lines, specify an absolute address, a C expression, or an assembly label
All lines in a function	❏ Select a function name. ❏ In the Area box, select Lines.	❏ Select a function name. ❏ In the Area box, select Lines.

(b) Marking ranges

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

(c) Marking functions

To mark this area...	If C level is selected...	If Assembly level is selected...
By function name	❏ Select a function name. ❏ In the Area box, select Functions.	Not applicable
All functions in a module	❏ Select a module name. ❏ In the Area box, select Functions.	Not applicable
All functions everywhere	❏ In the Area box, select Functions. ❏ Be sure that Function and Module are set to N/A.	Not applicable

Disabling an area

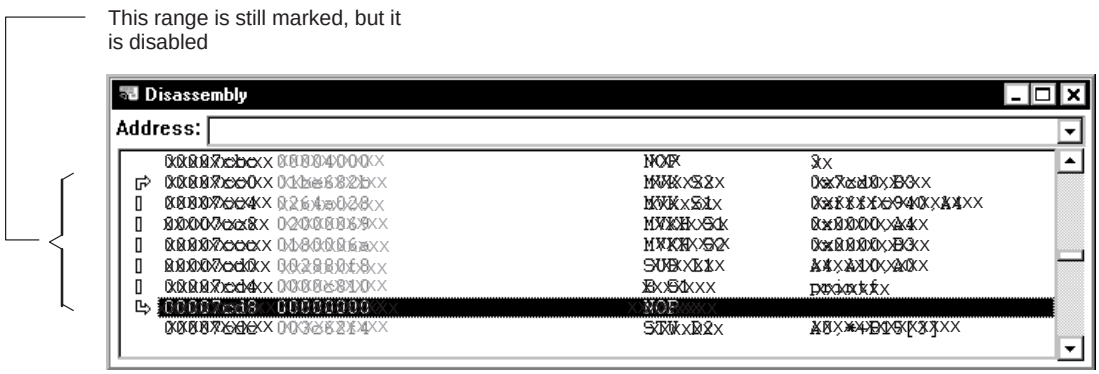
At times, it is useful to identify areas that you do not want affecting profile statistics. To do this, *disable* the appropriate area. Disabling effectively subtracts the timing information of the disabled area from all profile areas that include or call the disabled area. Areas must be marked before 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 a 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 Tools→Profile menu, select Select Areas.
   - From the context menu for the Profile window, select Select Areas.

   This displays the Profile Marking dialog box.

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

3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 8–3 on page 8-13 for a list of valid combinations.

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

   See Table 8–3 for a list of valid combinations.
5) Click Disable.
6) Click Close to close the dialog box.

Reenabling a disabled area

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

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

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

   This displays the Profile Marking dialog box.
2) In the Level box, select C, Assembly, or Both.
3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 8–3 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 8–3 for a list of valid combinations.
5) Click Enable.
6) Click Close to close the dialog box.

Unmarking an area

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

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

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

1) Open the Profile Marking dialog box by using one of these methods:
   - ❏ From the Tools→Profile menu, select Select Areas.
   - ❏ From the context menu for the Profile window, select Select Areas.
2) In the Level box, select C, Assembly, or Both.
3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 8–3 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 8–3 for a list of valid combinations.
5) Click Unmark.
6) Click Close to close the dialog box.

Restrictions on profiling areas

The following restrictions apply to profiling areas:

- ❏ An area cannot begin or end in the delay slot of a load instruction (emulator only).
- ❏ An area cannot begin in the delay slot of a branch instruction.
- ❏ An area can end in the last delay slot of a branch instruction but cannot end in any other delay slot of a branch instruction.

Table 8–3. Disabling, Enabling, Unmarking, or Viewing Areas

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

To identify this area...	If the C level is selected...	If the Assembly level is selected...	If the Both level is selected...
By line number, address†	❑ Select a module name. ❑ In the Area box, select Lines. ❑ Next to Lines, specify a line number.	❑ In the Area box, select Lines. ❑ Next to Lines, specify an absolute address, a C expression, or an assembly label.	Not applicable
All lines in a function	❑ Select a function name. ❑ In the Area box, select Lines.	❑ Select a function name. ❑ In the Area box, select Lines.	❑ Select a function name. ❑ In the Area box, select Lines.
All lines in a module	❑ Select a module name. ❑ In the Area box, select Lines.	❑ Select a module name. ❑ In the Area box, select Lines.	❑ Select a module name. ❑ In the Area box, select Lines.
All lines everywhere	❑ In the Area box, select Lines. ❑ Be sure that Function and Module are set to N/A.	❑ In the Area box, select Lines. ❑ Be sure that Function and Module are set to N/A.	❑ In the Area box, select Lines. ❑ Be sure that Function and Module are set to N/A.

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

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

To identify this area...	If the C level is selected...	If the Assembly level is selected...	If the Both level is selected...
By line numbers, addresses†	❑ Select a module name. ❑ In the Area box, select Ranges. ❑ Next to Ranges, specify a Start line number and an End line number.	❑ In the Area box, select Ranges. ❑ Next to Ranges, specify a Start and an End value as absolute addresses, C expressions, or assembly labels.	Not applicable
All ranges in a function	❑ Select a function name. ❑ In the Area box, select Ranges.	❑ Select a function name. ❑ In the Area box, select Ranges.	❑ Select a function name. ❑ In the Area box, select Ranges.
All ranges in a module	❑ Select a module name. ❑ In the Area box, select Ranges.	❑ Select a module name. ❑ In the Area box, select Ranges.	❑ Select a module name. ❑ In the Area box, select Ranges.
All ranges everywhere	❑ In the Area box, select Ranges. ❑ Be sure that Function and Module are set to N/A.	❑ In the Area box, select Ranges. ❑ Be sure that Function and Module are set to N/A.	❑ In the Area box, select Ranges. ❑ Be sure that Function and Module are set to N/A.

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

Table 8–3. Disabling, Enabling, Unmarking, or Viewing Areas (Continued)

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

To identify this area...	If the C level is selected...	If the Assembly level is selected...	If the Both level is selected...
By function name	❑ Select a function name. ❑ In the Area box, select Functions.	Not applicable	Not applicable
All functions in a module	❑ Select a module name. ❑ In the Area box, select Functions.	Not applicable	❑ Select a module name. ❑ In the Area box, select Functions.
All functions everywhere	❑ In the Area box, select Functions. ❑ Be sure that Function and Module are set to N/A.	Not applicable	❑ In the Area box, select Functions. ❑ Be sure that Function and Module are set to N/A.

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

To identify this area...	If the C level is selected	If the Assembly level is selected	If the Both level is selected
All areas in a function	❑ Select a function name. ❑ In the Area box, select All areas.	❑ Select a function name. ❑ In the Area box, select All areas.	❑ Select a function name. ❑ In the Area box, select All areas.
All areas in a module	❑ Select a module name. ❑ In the Area box, select All areas.	❑ Select a module name. ❑ In the Area box, select All areas.	❑ Select a module name. ❑ In the Area box, select All areas.
All areas everywhere	❑ In the Area box, select All areas. ❑ Be sure that Function and Module are set to N/A.	❑ In the Area box, select All areas. ❑ Be sure that Function and Module are set to N/A.	❑ In the Area box, select All areas. ❑ Be sure that Function and Module are set to N/A.

8.5 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 6.8 on page 6-15.

Setting a software breakpoint

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

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

1) Open the Breakpoint Control dialog box by using one of these methods:

   ❑ Click the Breakpoint Dialog icon on the toolbar:

   ❑ From the Configure menu, select Breakpoints.

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

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

4) Click Close to close the Breakpoint Control dialog box.

Clearing a software breakpoint

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

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

1) Open the Breakpoint Control dialog box by using one of these methods:
   - ❑ Click the Breakpoint Dialog icon on the toolbar:

     
   - ❑ From the Configure menu, select Breakpoints.
2) Select the address of the breakpoint that you want to clear.
3) Click Delete. The breakpoint is removed from the breakpoint list.
4) Click Close to close the Breakpoint Control dialog box.

8.6 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 8.7 on page 8-20). This reduces overhead because the debugger does not have to track entering/exiting subroutines within an area.

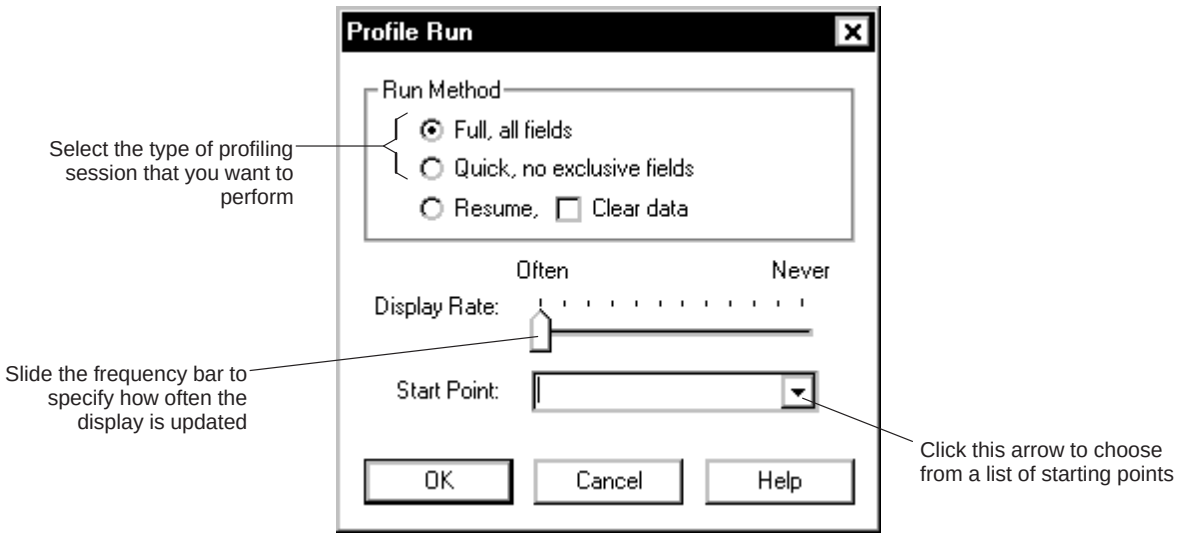
Running a full or a quick profiling session

To run a profiling session, follow these steps:

1) Open the Profile Run dialog box by using one of these methods:
   - Click the Run icon on the toolbar:

     
   - From the Debug menu, select Run.
   - Press F5.

   This displays the Profile Run dialog box:



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

3) Slide the Display Rate frequency bar to specify how often the display is updated.

   You can choose a Display Rate from Often to Never. A Display Rate of Never causes the profiler to display profiling information only when the profiling session is complete.

4) In the Start Point field, enter the starting point for the profiling session. The starting point can be a label, a function name, or a memory address. If you specify a memory address, be sure to prefix the address with **0x**.

   You can choose from a list of starting points by clicking on the arrow at the end of the Start Point field.

5) Click OK.

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

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

- ❑ Click the Halt icon on the toolbar:

- ❑ From the Debug menu, select Halt!.
- ❑ Press ESC.

Resuming a profiling session that has halted

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

1) Open the Profile Run dialog box by using one of these methods:

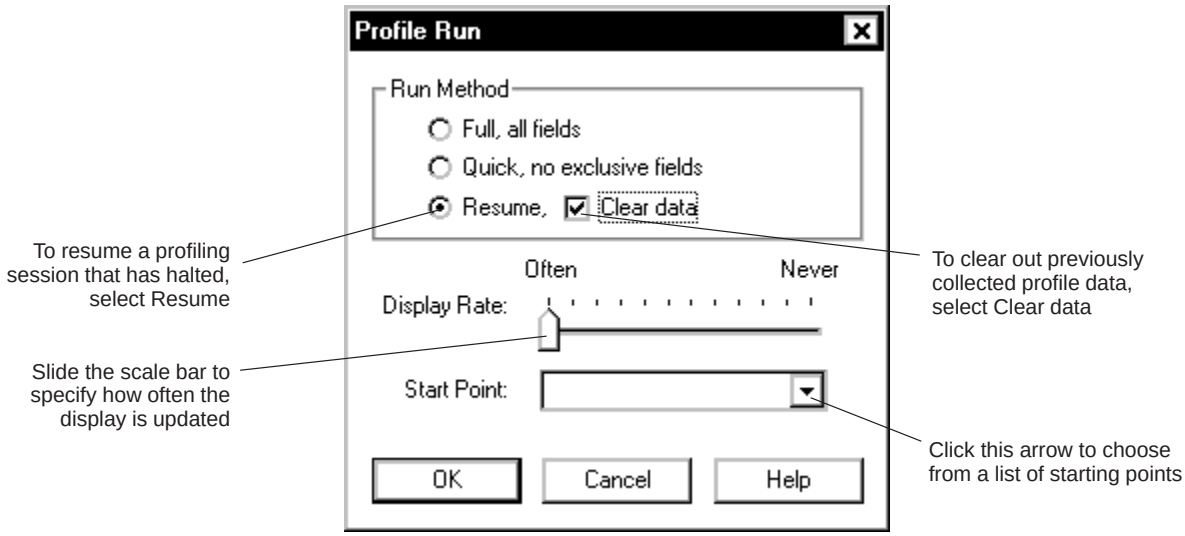
   ❑ Click the Run icon on the toolbar:

   ❑ From the Debug menu, select Run.

   ❑ Press F5.

   This displays the Profile Run dialog box:



2) In the Run Method box, select Resume.

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

4) Slide the Display Rate scale to specify how often the display is updated.

   You can choose a Display Rate from Often to Never. A Display Rate of Never causes the profiler to display profiling information only when the profiling session is complete.

5) In the Start Point field, enter the starting point for the profiling session. The starting point can be a label, a function name, or a memory address. If you specify a memory address, be sure to prefix the address with **0x**.

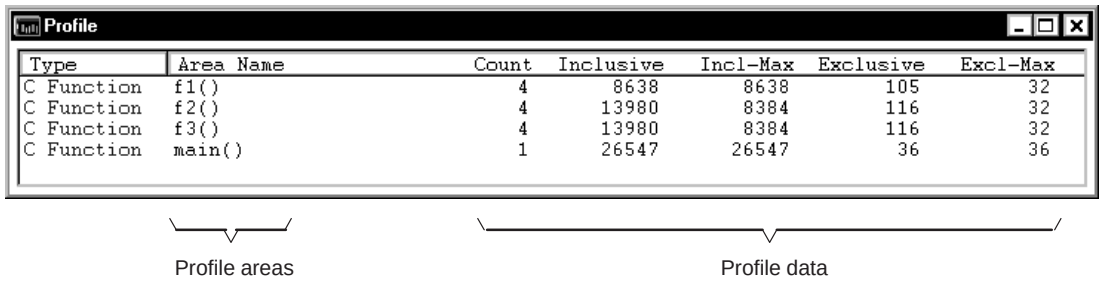
   You can choose from a list of starting points by clicking on the arrow at the end of the Start Point field.

6) Click OK.

8.7 Viewing Profile Data

The statistics collected during a profiling session are displayed in the Profile window. Figure 8–1 shows an example of this window.

Figure 8–1. An Example of the Profile Window



The example in Figure 8–1 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

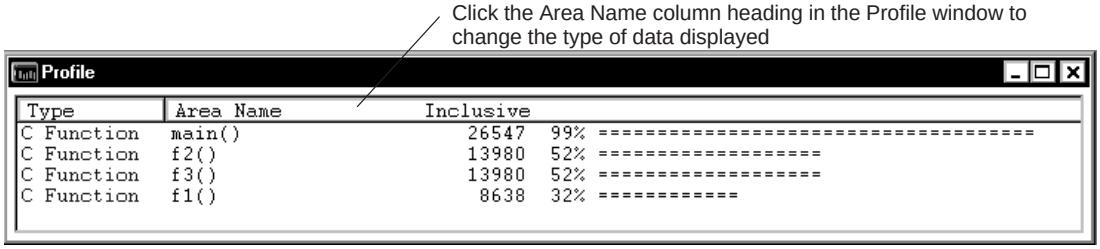
By default, the Profile window shows a set of statistics labeled as Count, Inclusive, Incl-Max, Exclusive, and Excl-Max. The Address field, which is not part of the default statistics, can also be displayed. Table 8–4 describes the statistic that each field represents.

Table 8–4. Types of Data Shown in the Profile Window

Label	Profile Data
Count	The number of times a profile area is entered during a session
Inclusive	The total execution time (cycle count) of a profile area, including the execution time of any subroutines called from within the profile area
Incl-Max (inclusive maximum)	The maximum inclusive time for one iteration of a profile area
Exclusive	The total execution time (cycle count) of a profile area, excluding the execution time of any subroutines called from within the profile area In general, the exclusive data provides the best statistics for comparing the execution time of one profile area to another area.
Excl-Max (exclusive maximum)	The maximum exclusive time for one iteration of a profile area
Address	The memory address of the line. If the area is a function or range, the Address field shows the memory address of the first line in the area.

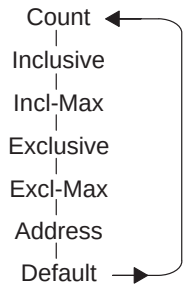
In addition to viewing this data in the default manner, you can view each of these statistics individually. The benefit of viewing them individually is that in addition to a cycle count, you are also supplied with a percentage indication and a histogram.

To view the fields individually, click the Area Name column heading in the Profile window.



When you click the Area Name column heading in the Profile window, fields are displayed individually in the order shown in Figure 8–2.

Figure 8–2. Cycling Through the Profile Window Fields



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

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

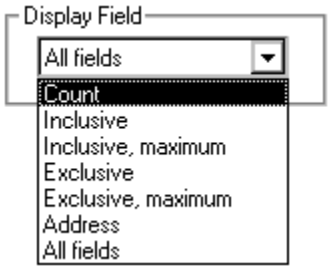
You can also use the Profile View dialog box to select the field you want to display. To do so, follow these steps:

1) Open the Profile View dialog box by using one of these methods:

   - ❑ From the Tools→Profile menu, select Change View.
   - ❑ From the context menu for the Profile window, select Change View.

   This displays the Profile View dialog box.

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



3) Click OK.

Sorting profile data

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

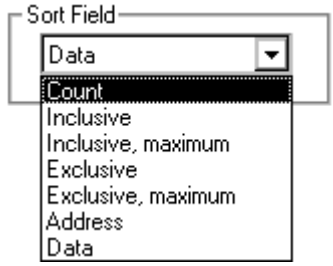
To sort the data on any of the data fields, follow these steps:

1) Open the Profile View dialog box by using one of these methods:

   - ❏ From the Tools→Profile menu, select Change View.
   - ❏ From the context menu for the Profile window, select Change View.

   This displays the Profile View dialog box.

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



3) Click OK.

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

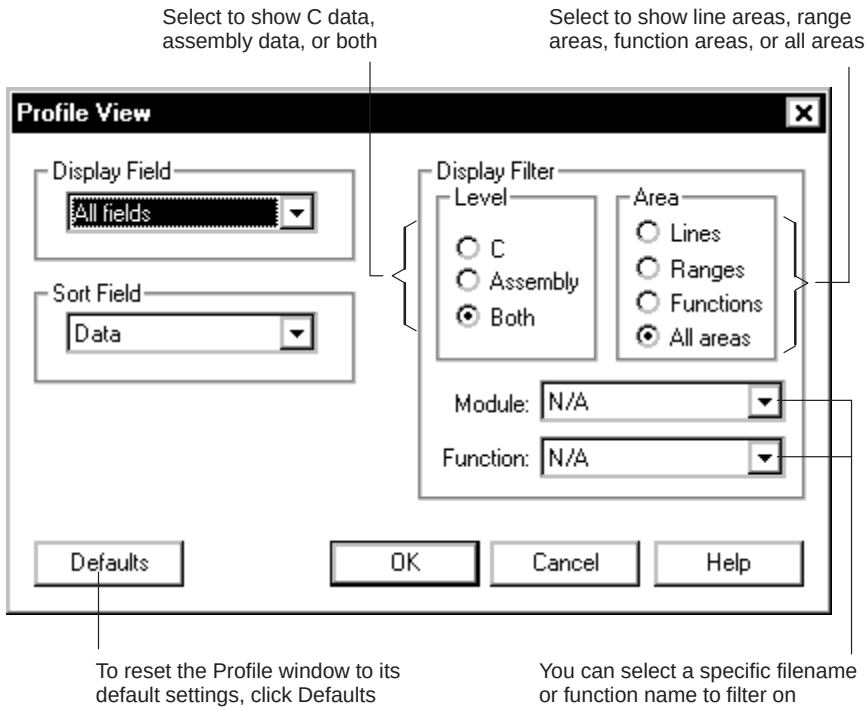
Viewing different profile areas

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

1) Open the Profile View dialog box by using one of these methods:

   - ❏ From the Tools→Profile menu, select Change View.
   - ❏ From the context menu for the Profile window, select Change View.

   This displays the Profile View dialog box.



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

3) In the Area box, select Lines, Ranges, Functions, or All areas. See Table 8–3 on page 8-13 on for a list of valid combinations.

4) If you want to view areas within a specific file or function, do one of the following:

   - ❏ From the Module combo box, select a specific filename.
   - ❏ From the Function combo box, select a specific function name.

   See Table 8–3 on page 8-13 for a list of valid combinations.

5) Click OK.

If you want to reset the Profile window to its default characteristics, use the Profile View dialog box (Profile→Change View). Click the Defaults button, then click OK.

Interpreting session data

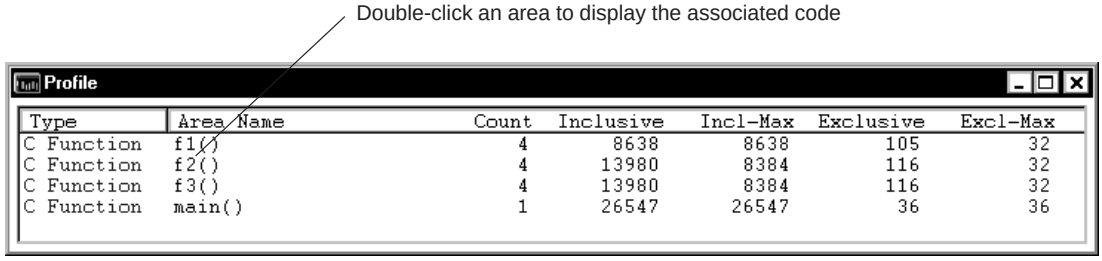
General information about a profiling session is displayed in the Command window during and after the session. This information identifies the starting and stopping points. It also lists statistics for three important areas:

- *Run cycles* shows the number of execution cycles consumed by the program from the starting point to the stopping point.
- *Profile cycles* equals the run cycles minus the cycles consumed by disabled areas.
- *Hits* shows the number of internal breakpoints encountered during the profiling session.

Viewing code associated with a profile area

You can view the code associated with a displayed profile area. The debugger updates the display so that the associated C or disassembly statements are shown in the File or Disassembly window.

To select the profile area in the Profile window and display the associated code, double-click the area that you want to display:



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.

8.8 Saving Profile Data to a File

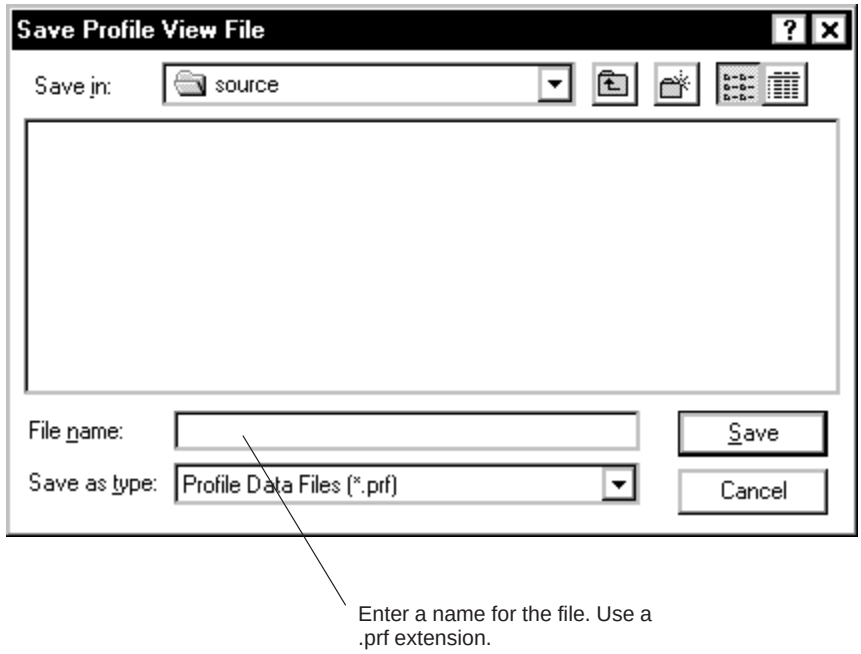
You may want to run several profiling sessions during a debugging session. Whenever you start a new profiling session, the results of the previous session are lost. However, you can save the results of the current profiling session to a system file.

The saved file contents are in ASCII and are formatted in exactly the same manner as they are displayed (or would be displayed) in the Profile window. The general profiling-session information that is displayed in the Command window is also written to the file.

Saving the contents of the Profile window

To save the contents of the Profile window to a system file, follow these steps:

1) From the Tools→Profile menu, select Save View. This displays the Save Profile View File dialog box:



2) In the File name field, enter a name for the file. You can use a .prf extension to identify the file as a profile data file.

3) Click Save.

This saves only the current view; if, for example, you are viewing only the Count field, then only that information is saved. If the file already exists, debugger overwrites the file with the new data.

Saving all data for currently displayed areas

To save all data for the currently displayed areas, follow these steps:

1) From the Tools→Profile menu, select Save All. This displays the Save Profile File dialog box.

2) In the File name field, enter a name for the file. You can use a .prf extension to identify the file as a profile data file.

3) Click Save.

This saves all views of the data—including the individual count, inclusive, etc.—with the percentage indications and histograms. If the file already exists, debugger overwrites the file with the new data.

Chapter 9

Monitoring Hardware Functions With the Emulator Analysis Module

The ’C27xx has an analysis module on the chip that allows the emulator to monitor hardware functions. Using the analysis module, you can count occurrences of certain hardware functions or set hardware breakpoints on these occurrences.

You access the analysis features through dialog boxes described in this chapter. These dialog boxes provide a transparent means of loading the special set of pseudoregisters that the debugger uses to access the on-chip analysis module.

Topic **Page**

9.1 Major Functions of the Analysis Module

The ’C27xx analysis module provides a detailed look into events occurring in hardware, expanding your debugging capabilities beyond software breakpoints. The analysis module examines ’C27xx bus cycle information in real time and reacts to this information through actions such as hardware breakpoints and event counting. The analysis module allows you to:

- **Count events.** The analysis module has two 16-bit and one 32-bit *internal counters* that can count several types of *events*. You can count the number of times a defined bus event or other internal event occurs during execution of your program.

  Events that can be counted include:

  - CPU clock cycles
  - Instructions
  - Interrupts taken
  - Data accesses
  - Program accesses

  You can use only one counter, the 32-bit counter or the 16-bit counters, at a time. For each counter, you can count only one event at a time.

- **Set hardware breakpoints.** You can also set up the analysis module to halt the processor during execution of your program. The events that cause the processor to stop are called *break events*. You can define a break event as one or more of the following conditions:

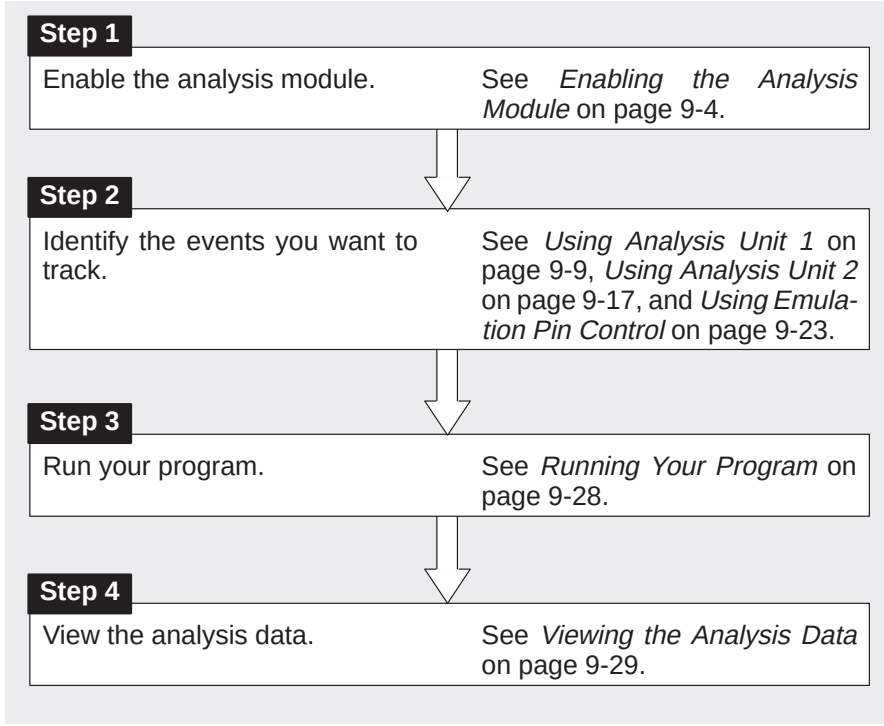
  - Any program address
  - A low level on the EMU0 pin (EMU0 driven low)
  - A low level on the EMU1 pin (EMU1 driven low)
  - A low level on the EXTTRIG pin (EXTTRIG driven low)

  Hardware break events allow you to set breakpoints in ROM and program memory. In addition, any of the debugger’s basic features available with software breakpoints can also be used with hardware breakpoints. As a result, you can take advantage of all the step and most of the run commands.

- **Set global breakpoints with EMU0/1 pins.** In a system of multiple ’C27xx processors connected by EMU0/1 (emulation event) pins, setting up the EMU0/1 pins allows you to create global breakpoints. Whenever one processor in your system reaches a breakpoint (software or hardware), *all* processors in the system can be halted.

9.2 Overview of the Analysis Process

Completing an analysis session consists of four simple steps:



9.3 Enabling the Analysis Module

When the debugger comes up, analysis is disabled by default. To begin tracking system events, you must explicitly enable analysis by pointing to Enable on the Tools→Analysis menu and selecting an option. To disable analysis, select Disable All from the Tools→Analysis menu.

When analysis is disabled, all events you previously enabled are now inactive, but remain unchanged. You can simply reenable analysis and use the events already defined. However, the settings for the action to take when the debugger halts are not saved.

You can save your events and action settings in a configuration file. For information, see section 9.10, *Using an Analysis Configuration File*, on page 9-30.

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

For information about the Enable→Advanced feature, see the *Enabling the advanced analysis features* section on page 9-7.

Note:

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

Enabling C stepping in ROM

The C stepping in ROM feature single-steps through code that is loaded into ROM or flash memory one C statement at a time, updating the display after executing each statement. If you C step through assembly language code, the debugger does not update the display until is has executed all assembly language statements associated with a single C statement. To C step in ROM, you must set up a hardware breakpoint first. Use one of these methods to enable C single-stepping in ROM:

- ❑ Click the C-Step in Rom icon on the toolbar:

- ❑ From the Analysis menu, point to Enable, then point to Enable CStepping in ROM.

Enabling the RUNB command

The RUNB command executes a specific section of code and counts the number of CPU clock cycles consumed by the execution. For RUNB to operate correctly, *execution must be halted by a software breakpoint*. After RUNB execution halts, the debugger stores the number of cycles into the CLK pseudoregister.

Use one of these methods to enable RUNB:

- Click the Enable RunB icon on the toolbar:

- From the Analysis menu, point to Enable, then point to Enable RunB.

Next, run your code with the RUNB command. For information on running your code after you set up the analysis module, see Section 9.8, *Running Your Program*, on page 9-28. For a complete explanation of the RUNB command and the benchmarking process, see section 6.6, *Benchmarking*, on page 6-13.

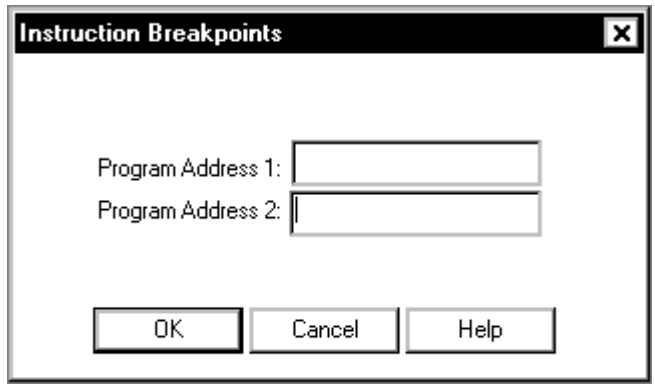
Enabling instruction breakpoints

You can set up two instruction (or hardware) breakpoints. The debugger halts when it reaches a hardware breakpoint. Use one of these methods to enable instruction breakpoints:

- ❑ Click the Setup Instruction Breakpoints icon on the toolbar:

- ❑ From the Tools menu, point to Enable, then point to Instruction Breakpoints.

Both methods display the Instruction Breakpoints dialog box:



- ❑ In the Program Address 1 or Program Address 2 field, enter the program address or the expression where you want the hardware breakpoint set. 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.
- ❑ Click OK to accept your changes and to dismiss the dialog box.

For information on setting up hardware breakpoints with Analysis Unit 1, see the *Setting up instruction breakpoints* section, on page 9-9. For information on setting up hardware breakpoints with Analysis Unit 2, see the *Setting up instruction breakpoints* section, on page 9-18.

Enabling the advanced analysis features

You must define the conditions the analysis module must meet to track a particular event. To do this, select the events you want to track by enabling the appropriate conditions in the Analysis Unit 1, Analysis Unit 2, or Emulation Pin Control dialog boxes.

You can display these dialog boxes by pointing to Analysis on the Tools menu, pointing to Enable and then Advanced, and selecting an option.

You can also display these dialog boxes through toolbar icons. The following sections detail the Analysis Unit 1, Analysis Unit 2, and Emulation Pin Control dialog boxes and discuss using the toolbar icons.

9.4 How the TMS320C27xx Buses Work

The analysis module allows you to control the ’C27xx address and data buses. These buses are affected by the combination of Analysis Unit 1 and Analysis Unit 2 settings.To understand some of the analysis module settings, it is important to understand how the buses work together.

The ’C27xx CPU has six address and data buses. The address buses are 32 bits and the data buses are 16 bits. Table 9–1 lists the buses.

Table 9–1. TMS320C27xx Address and Data Buses

Data Buses	Description	Address Buses	Description
PRDB	Program memory read data bus	PAB	Program memory read and write address bus
DRDB	Data memory read data bus	DRAB	Data memory read address bus
DWDB	Program memory and data memory write data bus	DWAB	Data memory write address bus

The address and data buses work in pairs in the following manner:

- To read from program memory, use PAB and PRDB.
- To write from program memory, use PAB and DWRB.
- To read from data memory, use DRAB and DRDB.
- To write from data memory, use DWAB and DWDB.

You can use two pairs of buses (or four buses) at the same time. For example, you can do the following actions:

- Read from program memory (PAB and PRDB) and read from data memory (DRAB and DRDB).
- Read from program memory (PAB and PRDB) and write to data memory (DWAB and DWDB).

You cannot use the same bus to do two different things at the same time. For example, you cannot write to program and data memory at the same time, because you would need to use the same bus (DWDB) for both writes.

9.5 Using Analysis Unit 1

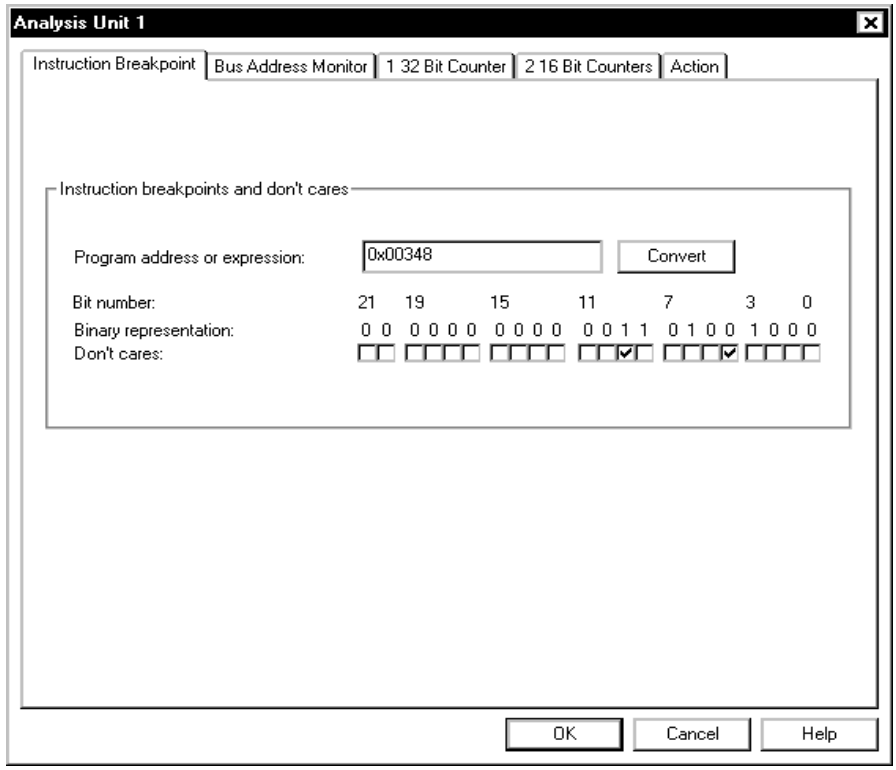
The analysis module detects hardware events and monitors the internal signals of the processor according to the parameters you define that count events or halt the processor.

Analysis Unit 1 is used to count events or monitor address buses. You must define the conditions the analysis module must meet to track a particular event with Analysis Unit 1 through tab dialog boxes. To display these tab dialog boxes, click the Analysis Unit 1 icon on the tool bar:

Setting up instruction breakpoints

The Instruction Breakpoint tab dialog box allows you to set up instruction (or hardware) breakpoints on Analysis Unit 1. You can set hardware breakpoints by entering program addresses or expressions, and you can specify don't care bits. With the Analysis Unit 1 tab dialog boxes displayed, follow these steps to set up a hardware breakpoint:

1) Click the Instruction Breakpoint tab. The Instruction Breakpoint tab dialog box appears:



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

3) To specify the don't care bits, follow these steps:

   a) Click Convert. The binary representation of the address you entered appears in the Binary Representation field.

   b) Click the checkboxes of the bits that you want to ignore in the Don't cares field.

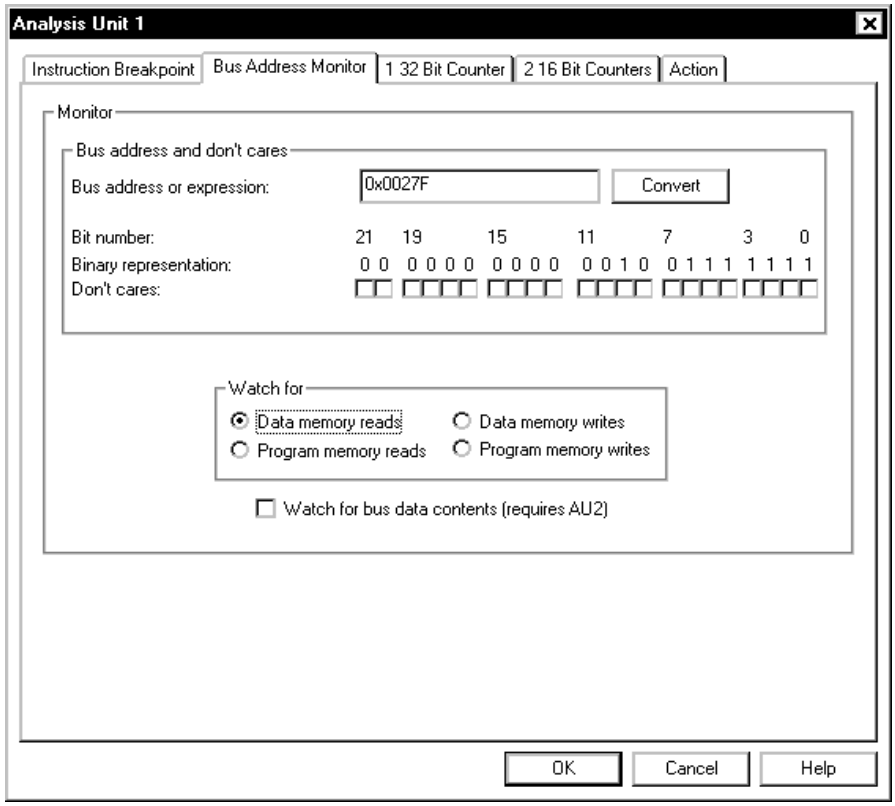
4) Click on the Action tab. From the displayed Action dialog box, select the Break radio button. (The *Choosing an action for Analysis Unit 1* section, on page 9-16, discusses the Action tab in more detail.)

5) Click OK to accept your changes and to dismiss the dialog box.

Setting up the bus address monitor

The Bus Address Monitor tab dialog box allows you monitor address bus activity on Analysis Unit 1. With the Analysis Unit 1 tab dialog boxes displayed, follow these steps to set up the bus address monitor:

1) Click the Bus Address Monitor tab. The Bus Address Monitor tab dialog box appears:



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

3) To specify the don't care bits, follow these steps:

   a) Click Convert. The binary representation of the address you entered appears in the Binary Representation field.

   b) Click the checkboxes of the bits that you want to ignore in the Don't cares field.

4) To specify the memory type, in the Watch for field, select the appropriate radio button for the type of memory reads or writes you want to watch for.

5) To watch for data values, click the Watch for bus data contents (requires AU2) checkbox. You must specify the data that you want to watch for in Analysis Unit 2. See the *Setting up the bus data monitor* section, on page 9-20, for information on setting up the data bus monitor on Analysis Unit 2.

6) Click on the Action tab. From the displayed Action dialog box, select the desired action radio button. (The *Choosing an action for Analysis Unit 1* section, on page 9-16, discusses the Action tab in more detail.)

7) Click OK to accept your changes and to dismiss the dialog box.

Counting events

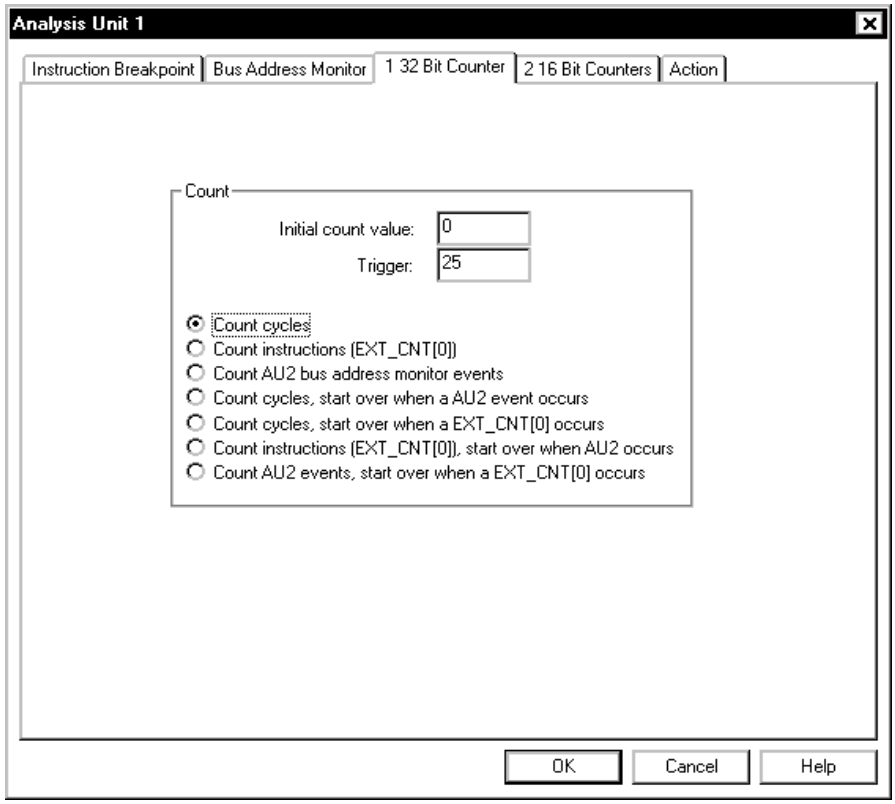
The analysis module has internal counters that count bus events and detect other internal events. The counters keep track of how many times an event occurs. The 1 32 Bit Counter and 2 16 Bit Counters dialog boxes allow you to count one of several types of events until the processor halts.

You can use either the 32-bit counter or the 16-bit counter(s) at one time. You can count a single event on any of the three counters, or you can count two events by setting up one on each of the two 16-bit counters. You cannot use the 32-bit counter while using either 16-bit counter. You can count events that occur on Analysis Unit 1 or on Analysis Unit 2.

Setting up the 32-bit counter

You can set up a 32-bit counter to count events that occur on Analysis Unit 1 or Analysis Unit 2. With the Analysis Unit 1 dialog boxes displayed, follow these steps to set up the 32-bit counter:

1) Click the 1 32 Bit Counter tab. The 1 32 Bit Counter tab dialog box appears:



2) In the Initial count value field, enter the start value that you want the counter to use. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.

3) In the Trigger field, enter the end value that you want the counter to use. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.

4) Click on the radio button that corresponds to the event you want to count. If you are counting an AU2 event, you must set up the event in the Analysis Unit 2 dialog boxes. (See section 9.6, *Using Analysis Unit 2*, on page 9-17, for information on setting up AU2 events.)

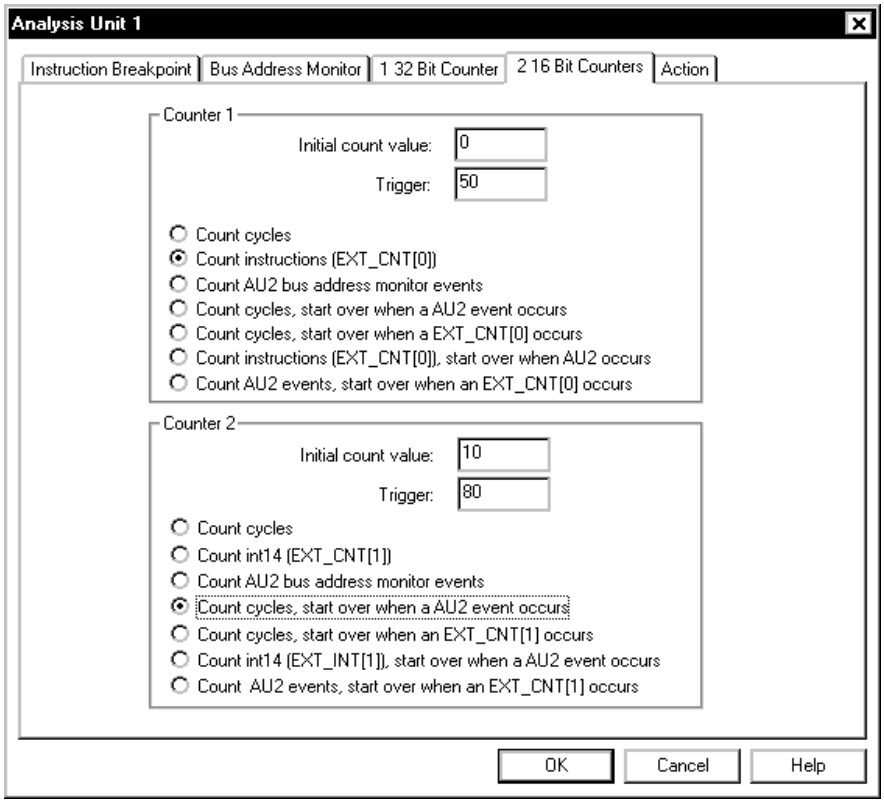
5) Click on the Action tab. From the displayed Action dialog box, select the desired action radio button. (The *Choosing an action for Analysis Unit 1* section, on page 9-16, discusses the Action tab in more detail.)

6) Click OK to accept your changes and to dismiss the dialog box.

Setting up the 16-bit counters

You can set up either or both of the 16-bit counters to count events that occur on Analysis Unit 1 or Analysis Unit 2. The counters can have different start and end values. With the Analysis Unit 1 dialog boxes displayed, follow these steps to set up the 16-bit counters:

1) Click the 2 16 Bit Counter tab. The 2 16 Bit Counter tab dialog box appears:



2) In the Initial count value field, enter the start value that you want the counter to use. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.

3) In the Trigger field, enter the end value that you want the counter to use. If you want to specify a hex address, be sure to prefix the address number with **0x**; otherwise, the debugger treats the number as a decimal address.

4) Click on the radio button that corresponds to the event you want to count. If you are counting an AU2 event, you must set up the event in the Analysis Unit 2 dialog boxes. (See section 9.6, *Using Analysis Unit 2*, on page 9-17, for information on setting up AU2 events.)

5) Click on the Action tab. From the displayed Action dialog box, select the desired action radio button. (The *Choosing an action for Analysis Unit 1* section, on page 9-16, discusses the Action tab in more detail.)

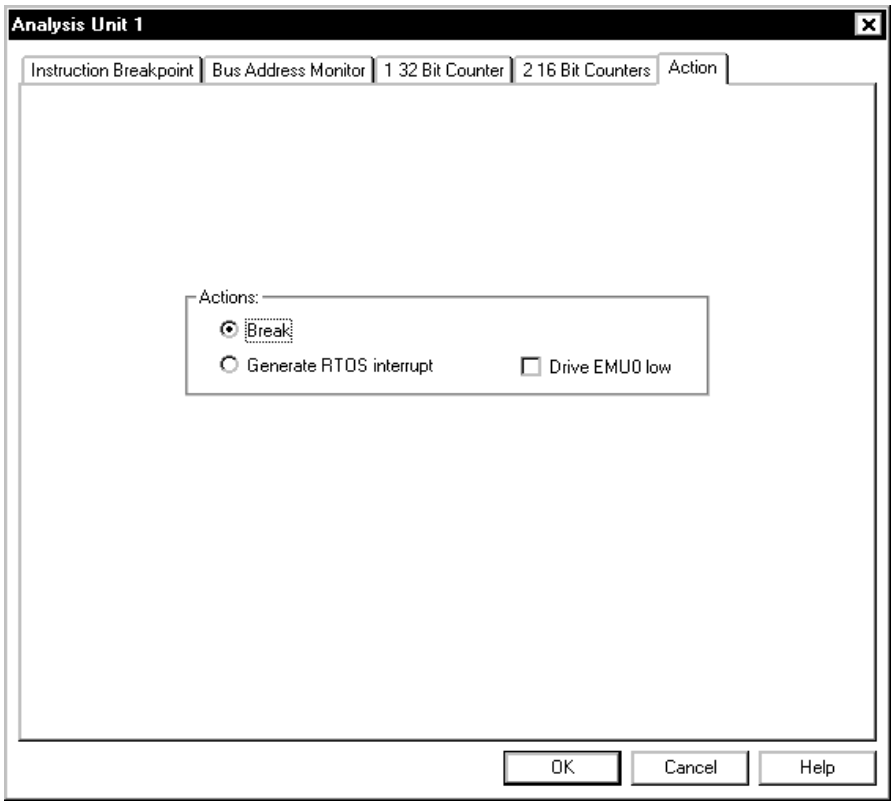
6) Click OK to accept your changes and to dismiss the dialog box.

Choosing an Action for Analysis Unit 1

The Action tab for Analysis Unit 1 allows you to specify the action that you want to occur when the Analysis Unit 1 event that you set up occurs.

Before the requested action can take place, you must set up your event in the Instruction Breakpoint, Bus Address Monitor, 1 32 Bit Counter, or 2 16 bit Counters tab dialog boxes. You can specify actions on only one Analysis Unit 1 tab dialog box at a time. To specify an action:

1) Click on Action. This displays the Action tab dialog box:



2) Select whether you want a hardware breakpoint or an RTOS interrupt to occur by selecting the appropriate radio button.

3) If you want EMU0 to drive low when the counter stops also, click the Drive EMU0 low checkbox.

4) Click OK to accept your changes and to dismiss the dialog box.

9.6 Using Analysis Unit 2

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

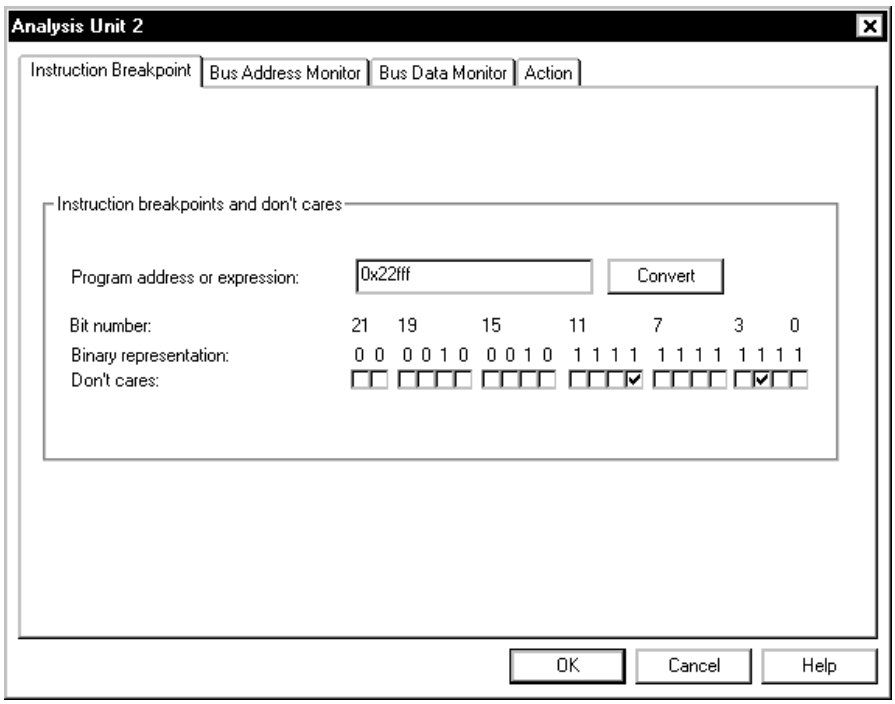
Analysis Unit 2 is used to monitor address and data buses. You must define the conditions the analysis module must meet to track a particular event with Analysis Unit 2 through dialog boxes. To display these tab dialog boxes, click the Analysis Unit 2 icon on the tool bar:

You can count certain Analysis Unit 2 events with the Analysis Unit 1 counters. You must set up the event in the Analysis Unit 2 dialog boxes and the counter in the Analysis Unit 1 counter dialog boxes. See the *Setting up the 32-bit counter* section, on page 9-13, or the *Setting up the 16-bit counters* section, on page 9-14, for more information on the counter dialog boxes.

Setting up instruction breakpoints

The Instruction Breakpoint tab dialog box allows you to set up instruction (or hardware) breakpoints on Analysis Unit 2. You can set hardware breakpoints by entering program addresses or expressions, and you can specify don't care bits. With the Analysis Unit 2 tab dialog boxes displayed, follow these steps to set up a hardware breakpoint:

1) Click the Instruction Breakpoint tab. The Instruction Breakpoint tab dialog box appears:



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

3) To specify the don't care bits, follow these steps:

   a) Click Convert. The binary representation of the address you entered appears in the Binary Representation field.

   b) Click the checkboxes of the bits that you want to ignore in the Don't cares field.

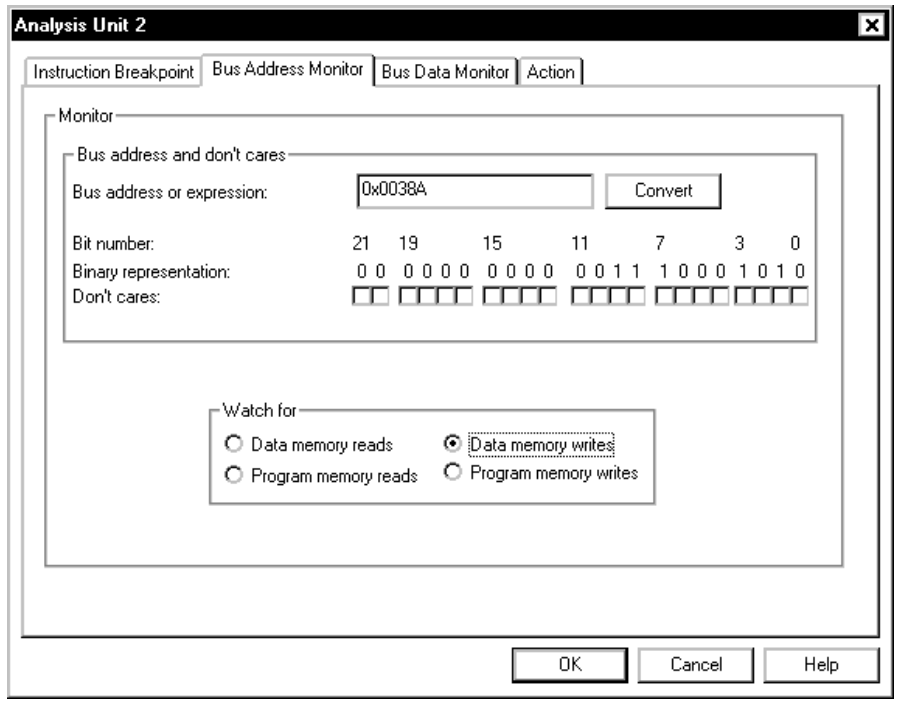
4) Click on the Action tab. From the displayed Action dialog box, select the Break radio button. (The *Choosing an action for Analysis Unit 2* section, on page 9-22, discusses the Action tab in more detail.)

5) Click OK to accept your changes and to dismiss the dialog box.

Setting up the bus address monitor

The Bus Address Monitor tab dialog box allows you monitor address bus activity on Analysis Unit 2. With the Analysis Unit 2 tab dialog boxes displayed, follow these steps to set up the bus address monitor:

1) Click the Bus Address Monitor tab. The Bus Address Monitor tab dialog box appears:



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

3) To specify the don't care bits, follow these steps:

   a) Click Convert. The binary representation of the address you entered appears in the Binary Representation field.

   b) Click the checkboxes of the bits that you want to ignore in the Don't cares field.

4) To specify the memory type, in the Watch for field, select the appropriate radio button for the type of memory reads or writes you want to watch for.

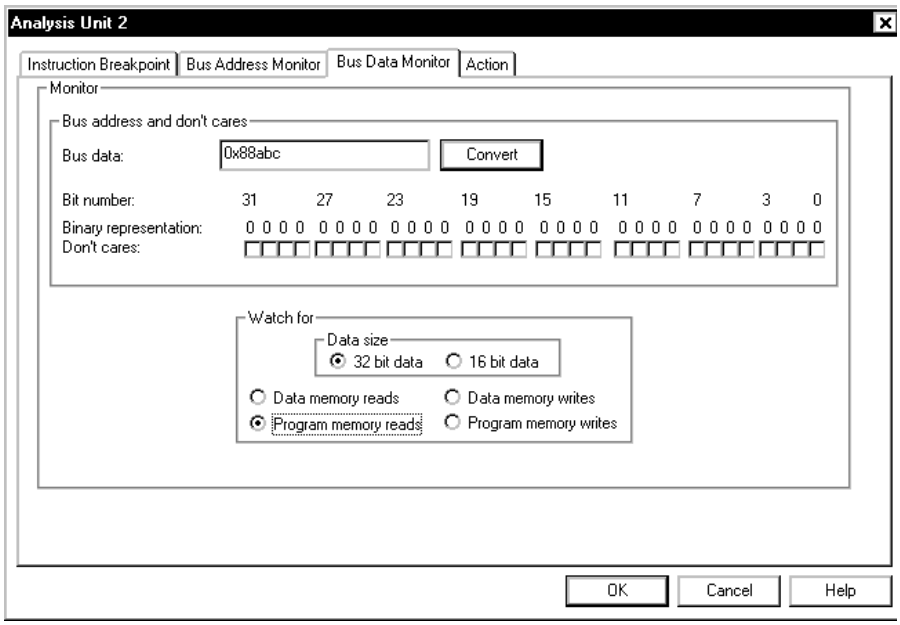
5) Click on the Action tab. From the displayed Action dialog box, select the desired action radio button. (The *Choosing an action for Analysis Unit 1* section, on page 9-16, discusses the Action tab in more detail.)

6) Click OK to accept your changes and to dismiss the dialog box.

Setting up the bus data monitor

The Bus Data Monitor tab dialog box allows you to monitor data bus activity on Analysis Unit 2. With the Analysis Unit 2 tab dialog boxes displayed, follow these steps to set up the bus data monitor:

1) Click the Bus Data Monitor tab. The Bus Data Monitor tab dialog box appears:



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

3) To specify the don't care bits, follow these steps:

   a) Click Convert. The binary representation of the address you entered appears in the Binary Representation field.

   b) Click the checkboxes of the bits that you want to ignore in the Don't cares field.

4) To specify the 16- or 32-bit data, in the Data size field, select the 32 bit or 16 bit radio button.

5) To watch for reads or writes, in the Watch for field, select the appropriate radio button for the type of memory reads or writes you want to watch for.

6) Click on the Action tab. From the displayed Action dialog box, select the desired action radio button. (The *Choosing an action for Analysis Unit 2* section, on page 9-22, discusses the Action tab in more detail.)

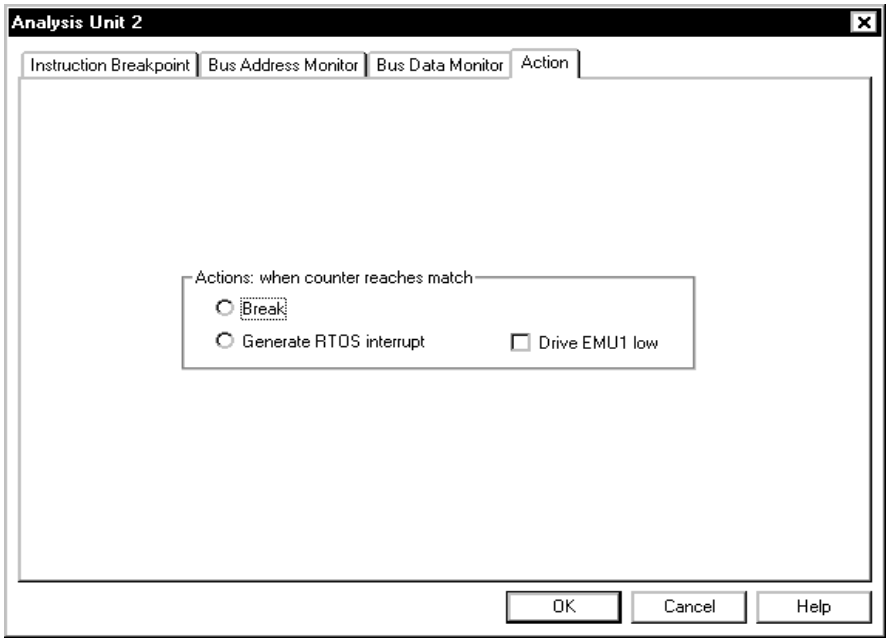
7) Click OK to accept your changes and to dismiss the dialog box.

Choosing an Action for Analysis Unit 2

The Action tab for Analysis Unit 2 allows you to specify the action that you want to occur when the Analysis Unit 2 event that you set up occurs.

Before the requested action can take place, you must set up your event in the Instruction Breakpoint, Bus Address Monitor, or Bus Data Monitor tab dialog boxes. You can specify actions on only one Analysis Unit 2 tab dialog box at a time. To specify an action:

1) With the Analysis Unit 2 dialog boxes displayed, click on Action. This displays the Action tab dialog box:



2) Select whether you want a hardware breakpoint or an RTOS interrupt to occur by selecting the appropriate radio button.

3) If you want EMU1 to drive low when the counter stops also, click the Drive EMU1 low checkbox.

4) Click OK to accept your changes to Analysis Unit 2 and to dismiss the dialog box.

9.7 Using Emulation Pin Control

By default, the EMU0/1 pins are set up as input signals; however, you can set them up as output signals or *trigger out* whenever the processor is halted by a software or hardware breakpoint. This is extremely useful when you have multiple ’C27xx processors in a system connected by their EMU0/1 pins.

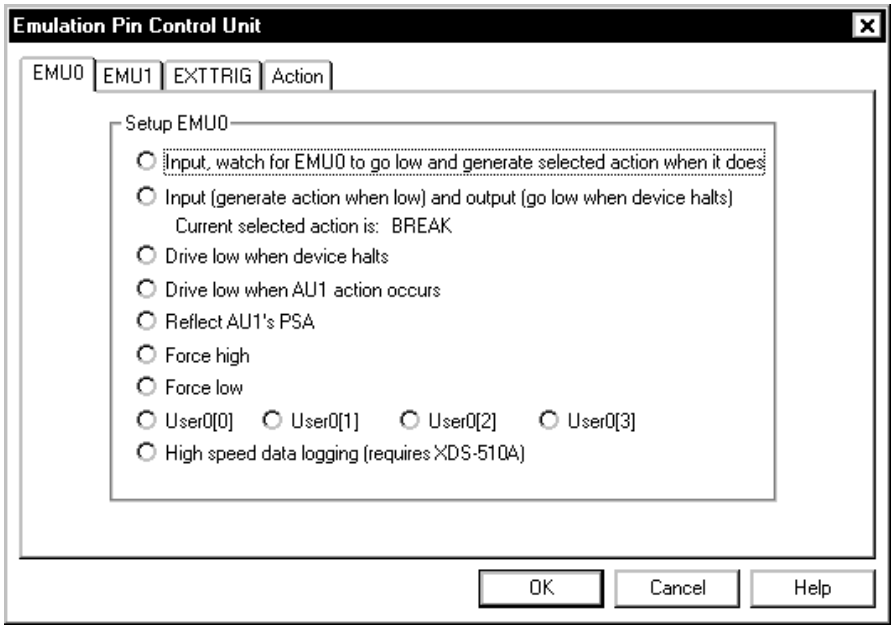
Selecting EMU0/1 does not, however, automatically halt all processors in the system. To do so, you must enable the EMU0/1 driven-low condition in the Action tab dialog box of Analysis Unit 1 or Analysis Unit 2 (respectively) for each processor. For example, if you have a system consisting of two processors connected by their EMU1 pins and you want to halt both processors when this pin is driven low, you must enable the EMU1 driven low option in the Action tab dialog box of Analysis Unit 2 for each processor.

The Emulation Pin Control dialog boxes allow you to set up three emulation pins: EMU0, EMU1, and EXTTRIG. To display these tab dialog boxes, click the Set Up Emulation Pin Control Unit icon on the tool bar:

Setting up the EMU0 pin

The EMU0 dialog box allows you to set up the EMU0 (emulation and test trigger channel 0) pin, which is used for asynchronous communications between the device and the scan controller. To set up the EMU0 pin with the Emulation Pin Control dialog boxes displayed, follow these steps:

1) Click the EMU0 tab. The EMU0 tab dialog box appears:



2) Indicate whether you want the pin to be used for input or for input and output by clicking the Input, watch for EMU0 to go low and generate selected action when it does, or the Input, (generate action when low) and output (go low when device halts) radio button.

   The other radio button selections are not available at this time.

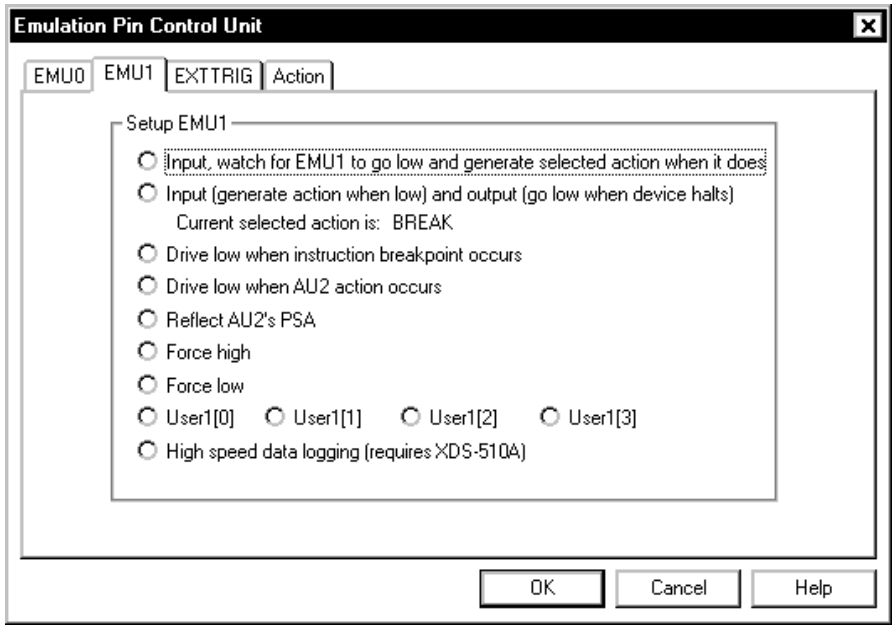
3) Click on the Action tab. From the displayed Action dialog box, select the desired action radio button. (The *Choosing an action for Emulation Pin Control* section, on page 9-27, discusses the Action tab in more detail.)

4) Click OK to accept your changes and to dismiss the dialog box.

Setting up the EMU1 pin

The EMU1 dialog box allows you to set up the EMU1 (emulation and test trigger channel 1) pin, which is used for asynchronous communications between the device and the scan controller. To set up the EMU1 pin with the Emulation Pin Control dialog boxes displayed, follow these steps:

1) Click the EMU1 tab. The EMU1 tab dialog box appears:



2) Indicate whether you want the pin to be used for input or for input and output by clicking the Input, watch for EMU0 to go low and generate selected action when it does, or the Input, (generate action when low) and output (go low when device halts) radio button.

   The other radio button selections are not available at this time.

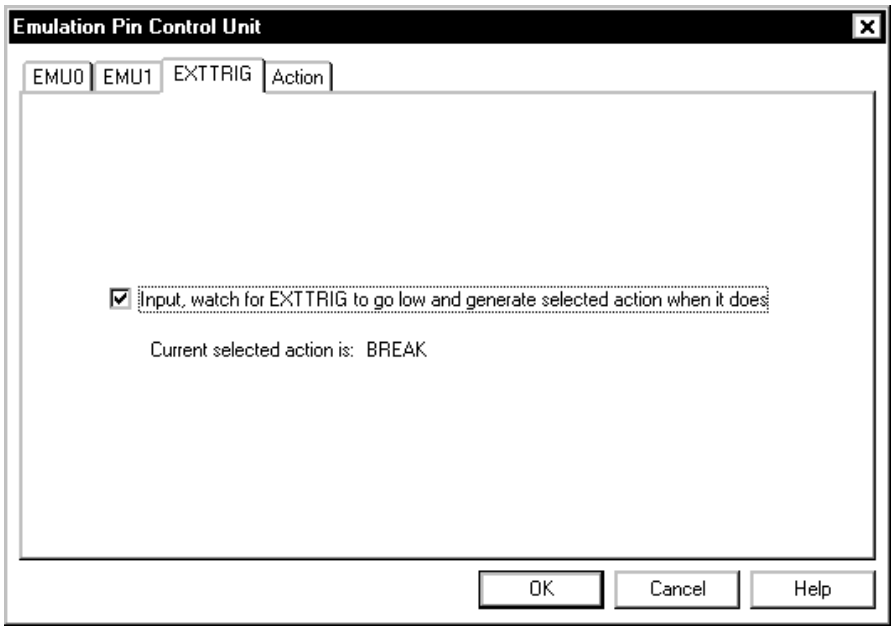
3) Click on the Action tab. From the displayed Action dialog box, select the desired action radio button. (The *Choosing an action for Emulation Pin Control* section, on page 9-27, discusses the Action tab in more detail.)

4) Click OK to accept your changes and to dismiss the dialog box.

Setting up the EXTTRIG pin

The EXTTRIG dialog box allows you to set up the EXTTRIG (external trigger) pin, which is used with the XDS524. The EXTTRIG dialog box allows you to generate the action specified in the Action tab when EXTTRIG goes low. To set up the EXTTRIG pin with the Emulation Pin Control dialog boxes displayed, follow these steps:

1) Click the EXTTRIG tab. The EXTTRIG tab dialog box appears:



2) Click the checkbox for input, watch for EXTTRIG to go low and generate selected action when it does.

3) If you want to change the specified action, click on the Action tab. From the displayed Action dialog box, select the desired action radio button. (The *Choosing an action for Emulation Pin Control* section, on page 9-27, discusses the Action tab in more detail.)

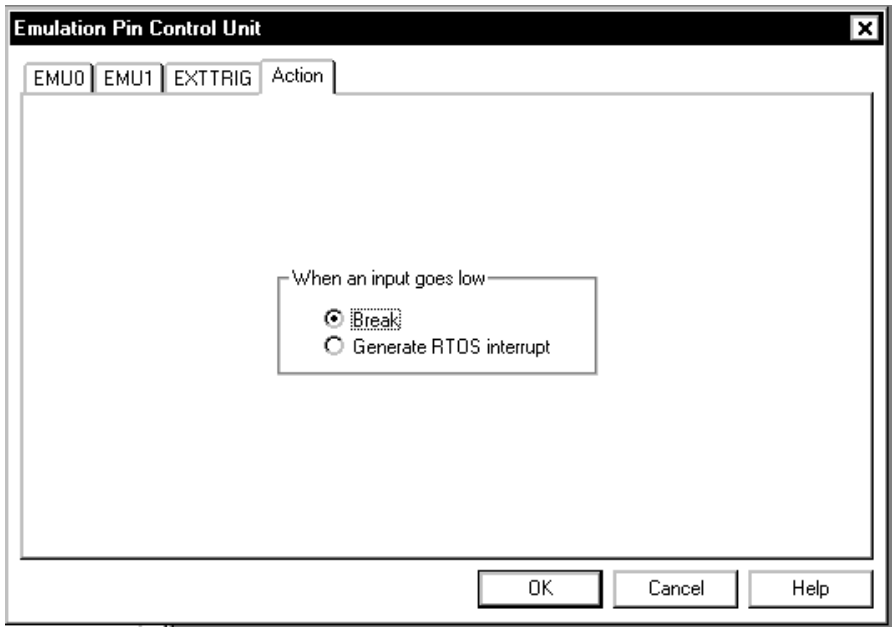
4) Click OK to accept your changes and to dismiss the dialog box.

Choosing an Action for Emulation Pin Control

The Action tab for Emulation Pin Control allows you to specify the action that you want to occur when the event that you set up occurs.

Before the requested action can take place, you must set up your event in the EMU0, EMU1, or EXTTRIG tab dialog boxes. You can specify an actions on only one Emulation Pin Control tab dialog box at a time. To specify an action:

1) With the Emulation Pin Control dialog boxes displayed, click on Action. This displays the Action tab dialog box:



2) Select whether you want a hardware breakpoint or an RTOS interrupt to occur by selecting the appropriate radio button.

3) Click OK to accept your changes to the Emulation Pin Control and to dismiss the dialog box.

9.8 Running Your Program

Once you have defined your parameters, the analysis module can begin collecting data as soon as you run your program. It will stop collecting data when the defined conditions are met. The analysis module monitors the progress of the defined events while your program is running.

Note:

The conditions for the analysis session must be defined *before* your analysis session begins; you cannot change conditions *during* execution of your program.

How to run the entire program

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

- ❏ Click the Run icon on the toolbar:

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

  run [*expression*]

You can use any of the debugger run commands (STEP, CSTEP, NEXT, etc.) described in Chapter 6 except the RUNB (run benchmarks) or RUNF (run free) command.

How the Run Benchmark (RUNB) command affects analysis

Running your program by selecting the Run Benchmarks option from the Debug menu or entering the RUNB command from the command line disables the current analysis settings and configures the counter to count CPU clock cycles. When the processor is halted after a RUNB, the analysis registers are restored to their original states.

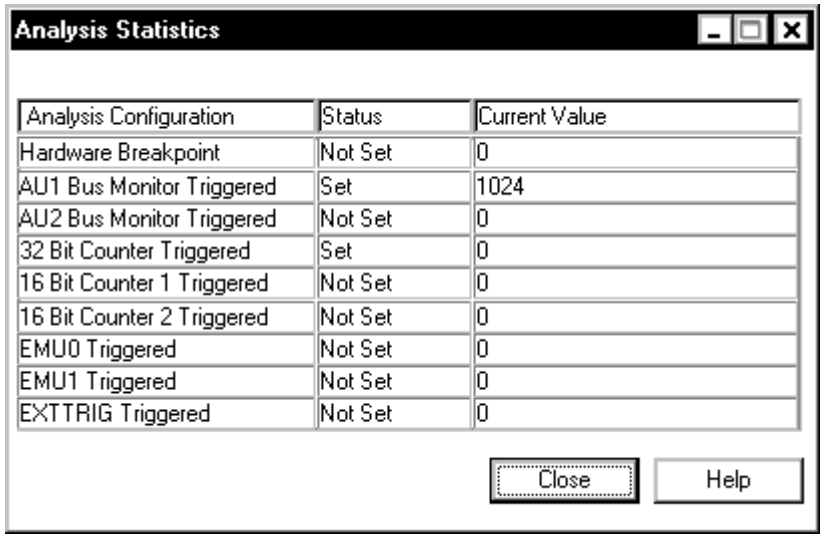
The analysis module provides capabilities in addition to those provided by the RUNB command. With the RUNB command you can count the number of CPU clock cycles only during the execution of a specific section of code. However, the analysis module not only allows you to count CPU clock cycles, it also allows you to count other events.

9.9 Viewing the Analysis Data

You can monitor the status of the analysis module by selecting Analysis Statistics from the View menu. This option displays the Analysis Statistics window. This window displays an ongoing progress report of the analysis module's activity. Through this window, you can monitor the status of the break events, the value of both the internal and external event counters, and the status of the EMU0/1 events.

You can check the status of the events that you defined in the Analysis Unit 1, Analysis Unit 2, and/or Emulation Pin Control dialog boxes in the Analysis Status window. If you change any of the analysis options in the analysis dialog boxes, after you issue a run or step command, the Analysis Status window updates to reflect the changes you made. Figure 9–1 illustrates the Analysis Status window.

Figure 9–1. Analysis Statistics Window Displaying a Status Report



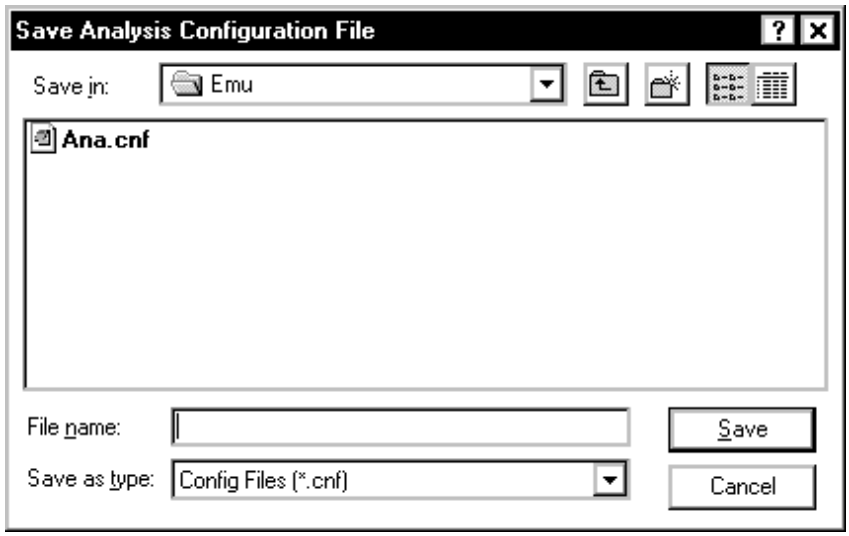
9.10 Using an Analysis Configuration File

An analysis configuration file allows you to save and reload your analysis module settings. This enables you to be more productive since you can use your settings over and over quickly and easily, rather than having to set the analysis configuration manually each time you reenter the debugger.

Saving analysis module settings

Analysis configuration settings are lost when you exit the debugger. However, you can save the analysis configuration that you have set by following these steps:

1) Open the Save Analysis Configuration File dialog box by choosing the Tools→Analysis→Save As State... menu option.



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

3) In the File name field, enter a name for the configuration file. You must use a .cnf extension to identify the file as a configuration file.

4) Click Save.

Notes:

1) The default analysis configuration file is ana.cnf.

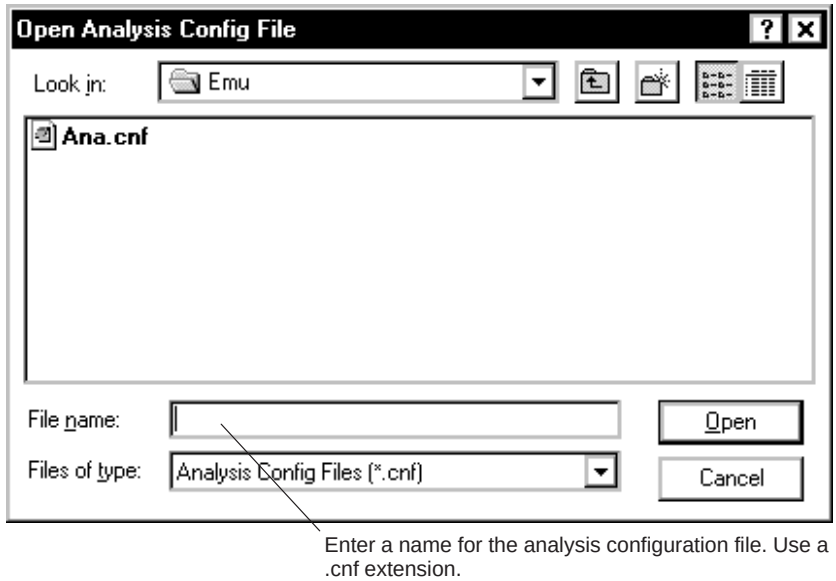
2) The configuration file is in ASCII form.

3) You can execute the configuration file with the Analysis menu Load State option to automatically set up the configuration that is defined in the file.

Loading saved breakpoint settings

To load a saved configuration, follow these steps:

1) Open the Open Analysis Config file dialog box by selecting the Tools→Analysis→Load State... menu option.



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

3) Click Open.

Chapter 10

Realtime Emulation

Realtime emulation provides advanced emulation features that can assist you in the development of your application system (software and hardware). This chapter describes the emulation features that are available on all ’C27xx devices using only the JTAG port (with TI extensions).

Topic **Page**

10.1 Overview of Realtime Emulation Features

Realtime emulation provides simple, inexpensive, and speed-independent access to the core for sophisticated debugging and economical system development, without requiring the costly cabling and access to processor pins required by traditional emulator systems or intruding on system resources.

The on-chip development interface provides:

- Control of the execution of background code while continuing to service time-critical interrupts.
  - Break on a software breakpoint instruction (instruction replacement)
  - Break on a specified program or data access without requiring instruction replacement (accomplished using bus comparators)
  - Break on external attention request from debug host or additional hardware
  - Break after the execution of a single instruction (single-stepping)
  - Control over the execution of code from device power–up
- Nonintrusive determination of device status
  - Detection of a system reset, emulation/test-logic reset, or power-down occurrence
  - Detection of the absence of a system clock or memory-ready signal
  - Determination of whether global interrupts are enabled
  - Determination of why debug accesses might be blocked
- Rapid transfer of memory contents between the device and a host (data logging).
- A cycle counter for performance benchmarking. With a 100 MHz cycle clock, the counter can benchmark activities up to 3 hours in duration.
- Minimally-intrusive access to internal and external memory.
- Minimally-intrusive access to CPU and peripheral registers.

10.2 Debug Terminology

The following definitions will help you to understand the information in the rest of this chapter:

- **Background code:** the body of code that can be halted during debugging because it is time-critical.
- **Foreground code:** the code of time-critical interrupt service routines, which are executed even when background code is halted.
- **Debug-halt state:** the state in which the device does not execute background code.
- **Time-critical interrupt:** an interrupt that must be serviced even when background code is halted. For example, a time-critical interrupt might service a motor controller or a high-speed timer.
- **Debug event:** an action such as the decoding of a software breakpoint instruction, the occurrence of an analysis breakpoint/watchpoint, or a request from a host processor that can result in special debug behavior such as halting the device or pulsing one of the signals EMU0 or EMU1.
- **Break event:** when a debug event causes the device to enter the debug-halt state.

10.3 Execution Control Modes

The emulator supports two debug execution control modes:

- ❑ Stop mode
- ❑ Real-time mode

Stop mode provides complete control of program execution, allowing for the disabling of all interrupts. Real-time mode allows time-critical interrupt service routines to be performed while execution of other code is halted. Both execution modes can suspend program execution at break events, such as occurrences of software breakpoint instructions or specified program-space or data-space accesses.

Stop Mode

Stop mode causes break events, such as software breakpoints and analysis watchpoints, to suspend program execution at the next interrupt boundary (which is usually identical to the next instruction boundary). When execution is suspended, all interrupts (including $\overline{\text{NMI}}$ and $\overline{\text{RS}}$) are ignored until the emulator receives a directive to run code again. In stop mode, the following execution states can be used:

- ❑ **Debug-halt state.** This state is accessed through a break event, such as decoding a software breakpoint instruction or the occurrence of an analysis breakpoint/watchpoint, or by the request of the host. In the stop mode debug-halt state, the CPU is halted. You can place the device into one of the other two states by giving the associated command to the emulator.

  Interrupts are not services, including $\overline{\text{NMI}}$ and $\overline{\text{RS}}$ (reset). When multiple instances of the same interrupts occur without the first instance being serviced, the later interrupts are lost.

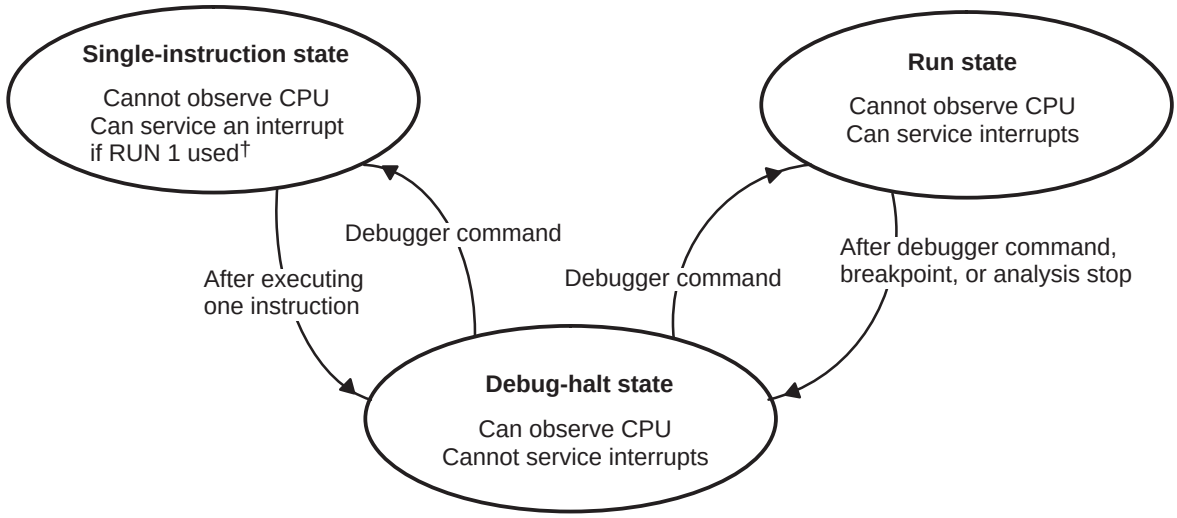
- ❑ **Single-instruction state.** This state is accessed when you enter a single instruction by using the RUN 1 command or the STEP command. The emulator executes the single instruction pointed to by the PC and then returns to the debug-halt state (it executes from one interrupt boundary to the next). The emulator is only in the single-instruction state until that single instruction is done.

  If an interrupt occurs in the single-instruction state, the command used to enter this state determines whether that interrupt is serviced. If a RUN 1 command is used, the emulator services the interrupt. If a STEP 1 command is used, the emulator does not service the interrupt, even if the interrupt is $\overline{\text{NMI}}$ or $\overline{\text{RS}}$.

- **Run state.** This state is accessed when you enter a command. Instructions executes until a debugger command or a debug event returns the CPU to the debug-halt state.

Figure 10–1 illustrates the relationship among the three states. Notice that the ’C27xx cannot pass directly between the single-instruction and run states. Also, the CPU can be observed only in the debug-halt state. In practical terms, this means the contents of CPU registers and memory are not updated in the emulator display in the single-instruction state or the run state. Maskable interrupts occurring in any state are latched in the interrupt flag register (IFR).

Figure 10–1. Stop Mode Execution States



† If you use a RUN 1 command to execute a single instruction, an interrupt can be serviced in the single-instruction state. If you use a STEP 1 command for the same purpose, an interrupt cannot be serviced.

Real-time Mode

Real-time mode allows you to debug code that interacts with interrupts that must not be disabled. Real-time debug allows the CPU to suspend background program execution at break events while continuing to service time-critical interrupt service routines (also referred to as foreground code). You can suspend program execution in multiple locations, which allows you to break within one time-critical interrupt while still servicing others. In real-time mode, the following execution states can be used:

- **Debug-halt state.** This state is accessed through a break event such as a software breakpoint instruction or an analysis breakpoint/watchpoint, or by a host request. You can place the device into one of the other two states by giving the associated command to the emulator.

  In this state, only time-critical interrupts are serviced. No other code is executed. Maskable interrupts are considered time-critical if they are enabled in the debug interrupt enable register (DBGIER). If they are also enabled in the interrupt enable register (IER), they are serviced. The interrupt global mask bit (INTM) is ignored. $\overline{\text{NMI}}$ and $\overline{\text{RS}}$ are also considered time critical, and are always serviced once requested. It is possible for multiple interrupts to occur and be serviced while the emulator is in the debug-halt state.

Note:

Should a time-critical interrupt occur in real-time mode at the precise moment that a RUN begins, the time-critical interrupt is taken and serviced in its entirety first.

- **Single-instruction state.** This state is accessed when you enter a single instruction by using a RUN 1 command or a STEP command. The emulator executes the single instruction pointed to by the PC and then returns to the debug-halt state.

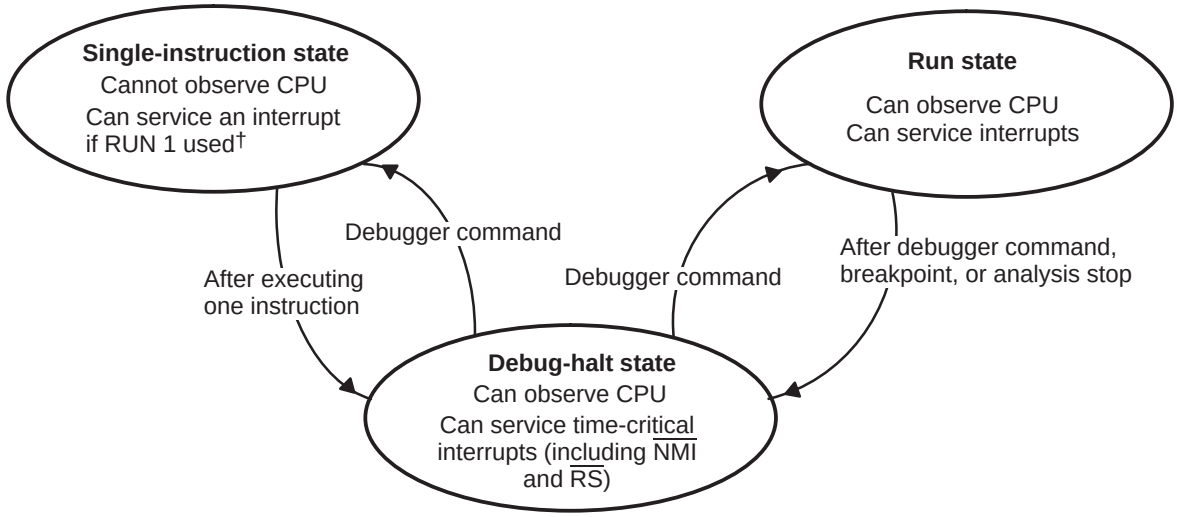
  If an interrupt occurs in this state, the command used to enter this state determines whether that interrupt is serviced. If a RUN 1 command is used, the emulator services the interrupt. If a STEP command is used, the emulator does not service the interrupt, even if the interrupt is $\overline{\text{NMI}}$ or $\overline{\text{RS}}$. In real-time mode, if the DBGM bit is 1 (debug events are disabled), a RUN 1 or STEP command forces continuous execution of instructions until DBGM is cleared.

- **Run state.** This state is accessed when you enter a RUN or RUN B command. The instructions execute until a debugger command or a debug event returns the emulator to the debug-halt state.

  All interrupts can be serviced in this state. When an interrupt occurs simultaneously with a debug event, the debug event has priority. However, if interrupt processing began before the debug event occurred, the debug event cannot processed until the ISR (interrupt service routine) begins.

Figure 10–2 illustrates the relationship among the three states. Notice that the ’C27xx cannot pass directly between the single-instruction and run states. Also, the CPU can be observed in the debug-halt state and in the run state. In the single-instruction state, the contents of CPU registers and memory are not updated in the debugger display. Maskable interrupts occurring in any state are latched in the interrupt flag register (IFR).

Figure 10–2. Real-time Mode Execution States



† If you use a RUN 1 command to execute a single instruction, an interrupt can be serviced in the single-instruction state. If you use a STEP 1 command for the same purpose, an interrupt cannot be serviced.

Caution about breakpoints within time-critical interrupt service routines

Do not use breakpoints within time-critical interrupt service routines. They cause the device to enter the debug-halt state, just as if the breakpoint were located in normal code. Once in the debug-halt state, $\overline{RS}$, $\overline{NMI}$, and those interrupts enabled in the DBGIER and the IER, are serviced.

After approving a maskable interrupt, the CPU disables the interrupt in the IER. This action prevents subsequent occurences of the interrupt from being serviced until the IER is restored by a return from interrupt (IRET) instruction, or until the interrupt is deliberately reenabled in the interrupt service routine (ISR). Do not reenable the IER bit for that IER while using breakpoints within the ISR. If you do so, a second triggering of the interrupt causes a new context save and restarts the ISR.

10.4 Using Analysis Resources

You can use analysis breakpoints, watchpoints, and a performance counter through the emulator, and you can use analysis or data logging through application code. This section shows the restrictions on which resources can be used at the same time. This section also explains other considerations for using the analysis resources.

Sharing of Analysis Resources

Table 10–1 lists the analysis resources, and Figure 10–3 shows which combinations of resources are valid.

Table 10–1. Analysis Resources

Resource	Purpose
BA0	Break on contents of program address or memory address bus.
BA1	Break on contents of program address or memory address bus.
BD	Break on contents of program data, memory read data, or memory write data in addition to an address bus.
Data log	Perform data logging using counter.
Benchmark	Count CPU cycles.

Figure 10–3. Valid Combinations of Debug and Test Resources

	BA0	BA1	BD	Data log	Benchmark
BA0	Yes	Yes	No	Yes†	Yes
BA1	Yes	Yes	No	No	No
BD	No	No	Yes	No	No
Data log	Yes†	No	No	Yes	No
Benchmark	Yes	No	No	No	Yes

† The data logging mode that uses the word counter allows this combination, but not the data logging mode that uses the end address (see section 10.6, *Data Logging*).

Using the Debugger to Perform Analysis Events

The emulator provides the following methods for using the memory-mapped emulation registers to perform analysis functions:

- **Analysis dialog boxes.** The analysis dialog boxes prompt you for information and set up the registers for you. For more information, see Chapter 9, *Monitoring Hardware Functions With the Emulator Analysis Module*.
- **Analysis aliases and registers.** The analysis aliases and registers provide a command-line interface to the analysis capabilities.
- **Command-line or memory windows.** You can use the command-line or the Memory windows to directly write to the memory locations.

10.5 Analysis Breakpoints, Watchpoints, and Counter(s)

This section describes three types of analysis features: analysis breakpoints, watchpoints, and counters. Data logging is described in section 10.6.

Analysis Breakpoints

An analysis breakpoint is sometimes called a hardware breakpoint, because it acts like a software breakpoint instruction (in this case, the ESTOP0 instruction) but does not require a modification to the application software. An analysis breakpoint triggers a debug event when an instruction at a breakpoint address would have entered the decode 2 phase of the pipeline (halting the CPU before the instruction is executed). A bus comparator watches the program address bus, comparing its contents against a reference address and a bit mask value.

Consider the following example. If a hardware breakpoint is set at T0, the emulator stops after returning from the T1 subroutine, with the instruction counter (IC) pointing to T0.

```
   NOP
   CALL   T1
T0:MOVB   AL, #0x00
   SB     TIMINGS, UNC
T1:NOP
   RET
T2:NOP
```

Hardware breakpoints allow for masking of address bits. For example, for the address range 00 0200_{16}–00 $02FF_{16}$, the mask address would be 00 $00FF_{16}$, and the reference address would be 00 $02FF_{16}$.

Watchpoints

A hardware watchpoint triggers a debug event when either an address or an address and data match a compare value. The address portion is compared against a reference address and bit mask, and the data portion is compared against a reference data value and a bit mask.

When comparing two addresses, you can set two watchpoints. When comparing an address and data, you can set only one watchpoint. When performing a read watchpoint, the address is available cycles earlier than the data; the watchpoint logic accounts for this.

Benchmark Counter/Event Counter(s)

The 40-bit performance counter on the ’C27xx can be used as a benchmark counter to increment every CPU clock cycle (it can be configured not to count when the CPU is in the debug-halt state). Wait states affect the counter. Wait states in the read 1 and write pipeline phases of an executing instruction affect the counter, regardless of whether an instruction is being single-stepped or run. However, wait states in the fetch 1 pipeline phase do not affect the counter during single-stepping because the cycle counting does not begin until the decode 2 pipeline phase. The counter counts wait states caused by instructions which are fetched but not executed. In most cases, these effects cancel each other out. Benchmarking is best used for larger portions of code. Do not rely heavily on the precision of the benchmarking. (For more information about the pipeline, refer to the *TMS320C27xx DSP CPU and Instruction Set Reference Guide*.)

You can configure the 40-bit performance counter as two 16-bit or one 32-bit event counter if you want to generate a debug event when the counter equals its match value. The comparison between the counter value and the match value is done before the count value is incremented. For example, suppose you initialize a counter to 0. A match value of 0 causes an immediate debug event (when the action to be counted occurs), and the counter holds 1 afterward.

You can also clear the counter when a hardware breakpoint or address watchpoint occurs. You can implement a mechanism similar to a watchdog timer: if a certain address is not seen on the address bus within a certain number of CPU clock cycles, a debug event occurs.

10.6 Data Logging

Data logging enables the ’C27xx to send selected information to a host computer using the standard JTAG port and an XDS510 or other compatible scan controller. Data logging does not affect ’C27xx operation.

You control data logging activity with your application code. To perform data logging, you must create a linear buffer of 32-bit words to hold a packet of information. Your application code controls the size, format, and location of this buffer and also determines when to send a buffer’s contents to the host. You can control the size of a data logging buffer in two ways:

- Specify an ending address
- Specify a count value in the upper eight bits of ADDRH when the number of 32-bit words you want to log is between 1 and 256.

The ’C27xx debugger writes the buffer to an ASCII file, *ptidlog.txt*, in its home directory. The ptidlog.txt file is created each time you invoke the debugger and it is closed when you quit the debugger. The text file records every packet of data received while the debugger was active.

Note:

When the debugger is not active, the data logging transfers are considered complete as soon as they are enabled to prevent the application software from getting stuck when there is nothing to receive the data.

10.7 Aborting Interrupts With the ABORTI Instruction

Generally, a program uses the IRET instruction to return from an interrupt. The IRET instruction restores all the values that were saved to the stack during the automatic context save. In restoring status register ST1 and the debug status register (DBGSTAT), it restores the debug context that was present before the interrupt.

In some target applications, you might have interrupts that must not be returned from by the IRET instruction. This can cause a problem for the emulation logic, because it assumes the original debug context will be restored. The abort interrupt (ABORTI) instruction is provided as a means to indicate that the debug context will not be restored and the debug logic needs to be reset to its default state. As part of its operation, the ABORTI instruction:

- ❏ Sets the DBGM bit in ST1. This disables debug events.
- ❏ Modifies select bits in DBGSTAT. The effect is a resetting of the debug context. If the emulator was in the debug-halt state before the interrupt occurred, the emulator does not halt when the interrupt is aborted.

The ABORTI instruction does not modify the DBGIER, the IER, the INTM bit or any analysis registers for example, regeristers used for breakpoints, watchpoints, and data logging).

10.8 DT-DMA Mechanism

The debug-and-test direct memory access (DT-DMA) mechanism provides access to memory, CPU registers, and memory-mapped registers, such as emulation registers and peripheral registers, without direct CPU intervention. DT-DMAs intrude on CPU time; however, you can block them by setting the debug enable mask bit (DBGM) in ST1.

Because the DT-DMA mechanism uses the same memory-access mechanism as the CPU, any read or write access that the CPU can perform in a single operation can be done by a DT-DMA. The DT-DMA mechanism presents an address (and data, in the case of a write) to the CPU, which performs the operation during an unused bus cycle (referred to as a *hole*). Once the CPU has obtained the desired data, it is presented back to the DT-DMA mechanism. The DT-DMA mechanism can operate in the following modes:

- **Nonpreemptive mode.**The DT-DMA mechanism waits for a hole on the desired memory buses. During the hole, the DT-DMA mechanism uses them to perform its read or write operation. These holes occur naturally while the CPU is waiting for newly fetched instructions, such as during a branch.

- **Preemptive mode.** In preemptive mode, the DT-DMA mechanism forces the creation of a hole and performs the access.

Nonpreemptive accesses to zero-wait-state memory take no cycles away from the CPU. If wait-stated memory is accessed, the pipeline stalls during each wait state, just as a normal memory access would cause a stall. In real-time mode, DT-DMAs to program memory cannot occur when application code is being run from memory with more than one wait state.

DT-DMAs can be polite or rude.

- **Polite accesses.** Polite DT-DMAs require that DBGM = 0.
- **Rude accesses.** Rude DT-DMAs ignore DBGM.

Note:

The information shown on the debugger screen is gathered at different times from the target; therefore, it does not represent a snapshot of the target state, but rather a composite. It also takes the host time to process and display the data. The data does not correspond to the current target state, but rather, the target state as of a few milliseconds ago.

Some key concepts of the DT-DMA mechanism are:

- Real-time-mode accesses are typically polite (although there may be reasons, such as error recovery, to perform rude accesses in real-time mode).
- In stop mode, DBGM is ignored, and the DT-DMA mode is set to preemptive.
- A DT-MA requests awakens the device from the idle state (initiated by the IDLE instruction). However, unlike returning from an interrupt, the CPU returns to the idle state upon completetion of the DT-DMA.

10.9 Debug Interface

The target level TI debug interface uses the five standard IEEE 1149.1 (JTAG) signals ($\overline{\text{TRST}}$, TCK, TMS, TDI, and TDO) and the two TI extensions (EMU0 and EMU1). Figure 10–4 shows the 14-pin JTAG header that interfaces the target to a scan controller, and Table 10–2 (page 10-18) defines the pins.

As shown in the figure, the header requires more than the five JTAG signals and the TI extensions. It also requires a test clock return signal (TCK_RET), the target supply (V_{CC}) and ground (GND). TCK_RET is a test clock out of the scan controller and into the target system. The target system uses TCK_RET if it does not supply its own test clock (in which case TCK would simply not be used). In many target systems, TCK_RET is simply connected to TCK and used as the test clock.

Figure 10–4. JTAG Header to Interface a Target to the Scan Controller

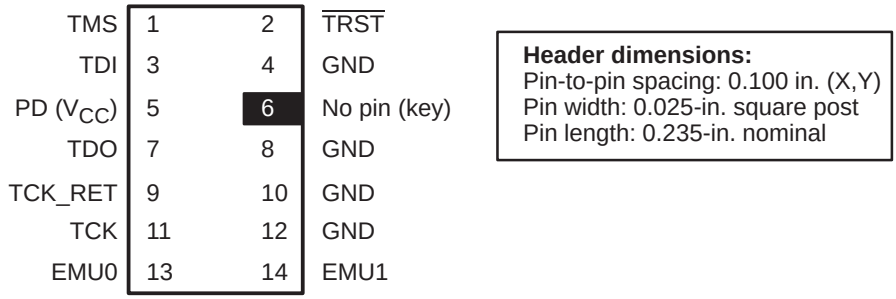


Table 10–2. 14-Pin Header Signal Descriptions

Signal	Description	Emulator State†	Target State†
EMU0	Emulation pin 0	I	I/O
EMU1	Emulation pin 1	I	I/O
GND	Ground		
PD (V_{CC})	Presence detect. Indicates that the emulation cable is connected and that the target is powered up. PD should be tied to V_{CC} in the target system.	I	O
TCK	Test clock. TCK is a clock source from the emulation cable pod. This signal can be used to drive the system test clock.	O	I
TCK_RET	Test clock return. Test clock input to the emulator. Can be a buffered or unbuffered version of TCK.	I	O
TDI	Test data input	O	I
TDO	Test data output	I	O
TMS	Test mode select	O	I
$\overline{\text{TRST}}$‡	Test reset	O	I

† I = input; O = output

‡ Do not use pullup resistors on $\overline{\text{TRST}}$: it has an internal pulldown device. In a low-noise environment, $\overline{\text{TRST}}$ can be left floating. In a high-noise environment, an additional pulldown resistor may be needed. (The size of this resistor should be based on electrical current considerations.)

Chapter 11

Summary of Commands

This chapter describes the basic debugger commands and profiling commands.

Topic **Page**

11.1 Functional Summary of Debugger Commands

This section summarizes the debugger commands according to these categories:

- **Changing modes.** These commands (listed on page 11-3) allow you to switch freely between the debugging modes (auto, mixed, and assembly) and execution modes (emulation and simulation).
- **Managing windows.** These commands (listed on page 11-3) allow you to make a window active and move or resize the active window.
- **Displaying and changing data.** These commands (listed on page 11-4) allow you to display and evaluate a variety of data items.
- **Performing system tasks.** These commands (listed on page 11-5) allow you to perform several system functions and provide you with some control over the target system.
- **Managing breakpoints.** These commands (listed on page 11-6) provide you with a command line method for controlling software breakpoints.
- **Displaying files and loading programs.** These commands (listed on page 11-3) 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 11-3) allow you to customize the debugger display, then save and later reuse the customized displays.
- **Memory mapping.** These commands (listed on page 11-6) allow you to define the areas of target memory that the debugger can access.
- **Running programs.** These commands (listed on page 11-7) provide you with a variety of methods for running your programs in the debugger environment.
- **Profiling commands.** These commands (listed on page 11-8) allow you to collect execution statistics for your code.

Changing modes

To put the debugger in...	Use this command...	See page...
Assembly mode	asm	11-11
Auto mode for debugging C code	c	11-13
Emulation mode	emu	11-19
Mixed mode	mix	11-29
Simulation mode	sim	11-42

Managing windows

To do this...	Use this command...	See page...
Reposition a window	move	11-30
Resize a window	size	11-42
Make a window active	win	11-51
Make a window as large as possible	zoom	11-52

Customizing the screen

To do this...	Use this command...	See page...
Change the command-line prompt	prompt	11-35
Load and use a previously saved custom screen configuration	sconfig	11-40
Save a custom screen configuration	ssave	11-44

Displaying files and loading programs

To do this...	Use this command...	See page...
Display a text file in a File window	file	11-20
Load an object file and its symbol table	load	11-24
Load only the object-code portion of an object file	reload	11-36
Load only the symbol-table portion of an object file	sload	11-43

Displaying and changing data

To do this...	Use this command...	See page...
Evaluate and display the result of a C expression	?	11-9
Display C and/or assembly language code at a specific point	addr	11-10
Display the Calls window	calls	11-13
Display assembly language code at a specific address	dasm	11-16
Display the values in an array or structure, or display the value that a pointer is pointing to	disp	11-17
Evaluate a C expression without displaying the results	eval	11-20
Display a specific line in the File window	line	11-23
Display a specific C function	func	11-21
Change the range of memory displayed in the Memory window or display an additional Memory window	mem	11-28
Change the format for displaying data values	setf	11-41
Display the current debugger version	version	11-49
Continuously display the value of a variable, register, or memory location within the Watch window	wa	11-49
Delete a data item from the Watch window	wd	11-50
Show the type of a data item	whatis	11-51
Delete all data items from the Watch window	wr	11-52

Performing system tasks

To do this...	Use this command...	See page...
Define your own command string	alias	11-11
Change the current working directory from within the debugger environment	cd, chdir	11-14
Clear all displayed information from the display area of the Command window	cls	11-14
List the contents of the current directory or any other directory	dir	11-16
Record the information shown in the display area of the Command window	dlog	11-18
Display a string to the Command window while executing a batch file	echo	11-19
Display a help topic for a debugger command	help	11-22
Conditionally execute debugger commands in a batch file	if/else/endif	11-23
Loop debugger commands in a batch file	loop/endloop	11-24
Pause the execution of a batch file	pause	11-31
Exit the debugger	quit	11-35
Reset communication with the emulator	reconnect	11-36
Reset the target system	reset	11-36
Associate a beeping sound with the display of error messages	sound	11-43
Enter any operating-system command or exit to a system shell	system	11-46
Execute commands from a batch file	take	11-47
Delete an alias definition	unalias	11-47
Name additional directories that can be searched when you load source files	use	11-48

Managing breakpoints

To do this...	Use this command...	See page...
Add a software breakpoint	ba	11-12
Delete a software breakpoint	bd	11-12
Display a list of all the software breakpoints that are set	bl	11-12
Reset (delete) all software breakpoints	br	11-13

Memory mapping

To do this...	Use this command...	See page...
Initialize a block of memory word by word	fill	11-20
Initialize a block of memory byte by byte	fillb	11-21
Add an address range to the memory map	ma	11-25
Enable or disable memory mapping	map	11-26
Connect a memory address to an input or output file (simulator only)	mc	11-26
Delete an address range from the memory map	md	11-27
Disconnect a file from memory (simulator only)	mi	11-28
Display a list of the current memory map settings	ml	11-29
Reset the memory map (delete all range definitions)	mr	11-30
Save a block of memory to a system file	ms	11-30
Connect an input file to the pin (simulator only)	pinc	11-32
Disconnect the input file from the pin (simulator only)	pind	11-33
List the pins that are connected to the input files (simulator only)	pinl	11-33

Running programs

To do this..	Use this command...	See page...
Single-step through assembly language or C code, one C statement at a time; step over function calls	cnext	11-15
Single-step through assembly language or C code, one C statement at a time	cstep	11-15
Run a program up to a certain point	go	11-22
Halt the target system	halt	11-22
Single-step through assembly language or C code; step over function calls	next	11-31
Reset the target system	reset	11-36
Reset the program to its entry point	restart	11-37
Execute code in a function and return to the function's caller	return	11-37
Run a program	run	11-38
Run a program with benchmarking—count the number of CPU clock cycles consumed by the executing portion of code	runb	11-38
Disconnect the emulator from the target system and run free	runf	11-39
Single-step through assembly language or C code	step	11-44
Cycle-step through assembly language code	stepcycle	11-45
Execute commands from a batch file	take	11-47

Profiling commands

All of the profiling commands can be entered from the Tools→Profile menu and associated dialog boxes. In many cases, using the Tools→Profile menu and dialog boxes is the easiest way to enter some of these commands. For this reason and also because there are over 100 profiling commands, most of these commands are not described individually in this chapter (as the basic debugger commands are).

Listed below are some of the profiling commands that you might choose to enter from the command line; these commands are also described in the alphabetical command summary. The remaining profiling commands are summarized in section 11.3, *Summary of Profiling Commands*, on page 11-53.

To do this...	Use this command...	See page...
Run a full profiling session	pf	11-32
Run a quick profiling session	pq	11-33
Resume a profiling session	pr	11-34
Switch to profiling environment	profile	11-34
Add a stopping point	sa	11-39
Delete a stopping point	sd	11-40
List all the stopping points	sl	11-42
Delete all the stopping points	sr	11-43
Save all the profile data to a file	vaa	11-48
Save currently displayed profile data to a file	vac	11-48
Reset the display in the Profile window to show all areas and the default set of data	vr	11-49

11.2 Alphabetical Summary of Debugger Commands

There are two types of debugger commands:

- ❑ Basic debugger commands
- ❑ Profiler commands that allow you to control the debugger profiling environment

Some commands can be used in more than one environment; other commands can be used in only one of the environments. Each command description identifies the applicable environments for the command.

Commands are not case sensitive; to emphasize this, command names are shown in both uppercase and lowercase throughout this book.

? *Evaluate Expression*

Syntax **?** *expression* [*, 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 *expression* identifies an address, you can follow it with @prog to identify program memory or @data to identify data memory. Without the suffix, the debugger treats an address expression as a program-memory location.

If the result of *expression* is not an array or structure, then the debugger displays the results in the Command window. If *expression* is a structure or array, ? displays the entire contents of the structure or array; you can halt long listings by pressing ESC.

When you use the optional *display format* parameter, data is displayed in one of the following formats:

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

addr — *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 Configure→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 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 View→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 — *Add Software Breakpoint*

Syntax **ba** *address*

Menu selection Configure→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 Configure→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. The *address* is treated as a program-memory address.

bl — *List Software Breakpoints*

Syntax **bl**

Menu selection Configure→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 Configure→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 View→C (Auto)

Toolbar selection none

Environments ☑ basic debugger ☐ profiling

Description The C command changes from the current debugging mode to auto mode. If you are already in auto mode, the C command has no effect.

calls — *Opens Calls Window*

Syntax **calls**

Menu selection View→Call Stack Window

Toolbar selection none

Environments ☑ basic debugger ☐ profiling

Description The CALLS command displays the Calls window. The debugger displays this window automatically when you are in auto/C or mixed mode. However, you can close the Calls window; the CALLS command opens the window again.

cd, chdir — *Change Directory*

Syntax

cd [*directory name*]
chdir [*directory name*]

Menu selection

none

Toolbar selection

none

Environments

☑ basic debugger ☑ 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.

cnext — *Single-Step C, Next Statement*

Syntax **cnext** [*expression*]

Menu selection Debug→Next C

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. (Section 6.5, *Running Code Conditionally*, page 6-12, discusses this in detail.)

cstep — *Single-Step C*

Syntax **cstep** [*expression*]

Menu selection Debug→Step C

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. (Section 6.5, *Running Code Conditionally*, page 6-12, discusses this in detail.)

dasm — *Display Disassembly at Specific Address*

Syntax **dasm** {*address*[**@prog** | **@data**] | *function name*}

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description The DASM command displays code beginning at a specific point within the Disassembly window. You can follow the *address* with @prog to identify program memory or @data to identify data memory. Without the suffix, the debugger treats an address as a program-memory location.

dir — *List Directory Contents*

Syntax **dir** [*directory name*]

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☑ 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:

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

You can use the *display format* parameter only when you are displaying a scalar type, an array of scalar type, or an individual member of an aggregate type.

You can also use the DISP command with a typecast expression to display memory contents in any format. Here are some examples:

```
disp *0
disp *(float *)123
disp *(char *)0x111
```

This shows memory in the Watch window as an array of locations; the location that you specify with the *expression* parameter is member [0], and all other locations are offset from that location.

dlog — *Record Display Area*

Syntax

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

Menu selection

File→Open→Log File
or
File→Close→Log File

Toolbar selection

none

Environments

☑ basic debugger ☑ 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* — Batch File Only

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.

else — *Execute Alternative Commands* — Batch File Only

Description ELSE provides an alternative list of commands in the IF/ELSE/ENDIF command sequence. See page 11-23 for more information about these commands.

emu — *Enable Emulation Execution Mode* — Simulator Only

Syntax **emu**

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☐ profiling

Description The EMU command selects the emulation execution mode. When you halt the debugger, the pipeline is flushed.

The simulator supports two modes of execution: simulation and emulation. The principal difference between the modes is in the state of the pipeline when execution is halted. Section 2.9, *Execution Modes*, on page 2-17 discusses this in detail.

endif — *Terminate Conditional Sequence* — Batch File Only

Description ENDIF identifies the end of a conditional-execution command sequence begun with an IF command. See page 11-23 for more information about these commands.

endloop — *Terminate Looping Sequence* — **Batch File Only**

Description ENDLOOP identifies the end of the LOOP/ENDLOOP command sequence. See page 11-24 for more information about the LOOP/ENDLOOP commands.

eval — *Evaluate Expression*

Syntax **eval** *expression*
e *expression*

Menu selection none

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 — *Display Text File*

Syntax **file** *filename*

Menu selection File→Open

Toolbar selection 

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 as multiple text files at the same time using multiple File windows.

fill — *Fill Memory Word by Word*

Syntax **fill** *address*, *page*, *length*, *data*

Menu selection Configure→Memory Fill

Toolbar selection none

Environments ☑ basic debugger ☐ profiling

Description The FILL command fills a block of memory word by word with a specified value.

- The *address* parameter identifies the first address in the block.

- ❏ The *page* parameter is a 1-digit number that identifies the type of memory (program or data) that a range occupies:

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

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

fillb — *Fill Memory Byte by Byte*

Syntax **fillb** *address*, *length*, *data*

Menu selection Configure→Memory Fill

Toolbar selection none

Environments ☑ basic debugger ☐ 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; an *address* parameter is treated as a program-memory address. FUNC works the same way FILE works, but with FUNC you do not need to identify the name of the file that contains the function.

go — *Run to Specified Address*

Syntax **go** [*address*]

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☐ profiling

Description The GO command executes code up to a specific point in your program. The *address* parameter is treated as a program-memory address. If you do not supply an *address*, then GO acts like a RUN command without an *expression* parameter.

halt — *Halt Target System*

Syntax **halt**

Menu selection Debug→Halt!

Toolbar selection 

Environments ☑ basic debugger ☐ profiling

Description The HALT command halts your program, if you are using a simulator, or halts the target system after you have entered a RUNF command, if you are using an emulator. When you invoke the debugger, it automatically executes a HALT command. If you enter a RUNF, quit the debugger, and later reinvoke the debugger, you will effectively reconnect the emulator to the target system and run the debugger in its normal mode of operation.

help — *Display Help Topic for Debugger Command*

Syntax **help** [*debugger command*]

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☑ 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

Conditionally Execute Debugger Commands **Batch File Only**

Syntax

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

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description

These commands allow you to execute debugger commands conditionally in a batch file. If the *expression* if nonzero, the debugger executes the commands between the IF and the ELSE or ENDIF. The ELSE portion of the command sequence is optional.

You can substitute a keyword for the expression. Keywords evaluate to true (1) or false (0). You can use the following keywords with the IF command:

- **$$EMU$$** (tests for the emulator version of the debugger)
- **$$SIM$$** (tests for the simulator version of the debugger)

The conditional commands work with the following provisions:

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

line

Display the specified line number in the FILE window

Syntax **line** *line number*

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description

Use the LINE command to view specific lines of code. The LINE command displays the specified *line number* in the middle of the FILE window. When the *line number* is already displayed in the FILE window, the LINE command does not affect the display.

load — *Load Executable Object File*

Syntax **load** *object filename*

Menu selection File→Load→Load Program

Toolbar selection none

Environments ☑ basic debugger ☑ 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/endloop — *Loop Through Debugger Commands* — **Batch File Only**

Syntax

loop *expression*
debugger commands
endloop

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 — *Add Block to Memory Map*

Syntax **ma** *address*, *page*, *length*, *type*

Syntax **ma** *address*, *page*, *length*, *type*

Menu selection Configure→Memory Maps

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 data or program memory. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.
- The *page* parameter is a 1-digit number that identifies the type of memory (program or data) that a range occupies:

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

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

To identify this kind of memory . . .	Use this keyword as the *type* parameter . . .
Read-only memory	**R** or **ROM**
Write-only memory	**W** or **WOM**
Read/write memory	**R|W** or **RAM**
External read/write memory	**EX|RAM**
External read-only memory	**EX|ROM**
Single-access read/write memory	**SARAM**
Dual-access read/write memory	**DARAM**
Nonvolatile reprogrammable memory	**FLASH**

map — *Enable/Disable Memory Mapping*

Syntax **map** {**on** | **off**}

Menu selection Configure→Memory Maps

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description The MAP command enables or disables memory mapping. Disabling memory mapping can cause bus fault problems in the target because the debugger may attempt to access nonexistent memory.

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

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

mc — *Connect Memory to a File* **Simulator Only**

Syntax **mc** *port address*, *page* , *length*, *filename*, {**READ** | **WRITE**}

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description The MC command connects a memory address to an input or output file. Before you can connect the address, you must add it to the memory map with the MA command.

- The *port address* parameter defines the memory address. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.
- The *page* parameter is a 1-digit number that identifies the type of memory (program or data) that a range occupies:

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

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

- ❏ The *filename* parameter can be any filename. If you connect a memory address to read from a file, the file must exist or the MC command will fail.
- ❏ The final parameter is specified as **READ** or **WRITE** and defines how the file will be used (for input or output, respectively).

The file is accessed during an assembly language read or write of the associated memory address. Any memory address can be connected to a file. A maximum of one input and one output file can be connected to a single memory address; multiple addresses can be connected to a single file.

md — *Delete Block From Memory Map*

Syntax **md** *address*, *page*

Menu selection Configure→Memory Maps

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description The MD command deletes a range of memory from the debugger's memory map.

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

  ```
  Specified map not found
  ```

- ❏ The *page* parameter is a 1-digit number that identifies the type of memory (program or data) that a range occupies:

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

Note:

If you want to use the MD command to remove a memory address that is connected to a file, you must first disconnect the address 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:

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

mi — *Disconnect a File From Memory* — **Simulator Only**

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

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description The MI command disconnects a memory address from its associated input or output file.

- The *port address* parameter identifies the memory address, which must be defined previously with the MC command.

- The *page* parameter is a 1-digit number that identifies the type of memory (program or data) that a range occupies:

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

- The read/write characteristics must match the parameter used when the memory address was connected.

mix — *Enter Mixed Mode*

Syntax **mix**

Menu selection View→Mixed

Toolbar selection none

Environments ☑ basic debugger ☐ profiling

Description The MIX command changes from the current debugging mode to mixed mode. If you are already in mixed mode, the MIX command has no effect.

ml — *List Memory Map*

Syntax **ml**

Menu selection Configure→Memory Maps

Toolbar selection none

Environments ☑ basic debugger ☑ 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 need to specify only enough letters to identify the window.

mr — *Reset Memory Map*

Syntax **mr**

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description The MR command resets the debugger's memory map by deleting all defined memory ranges from the map.

ms — *Save Memory Block to File*

Syntax **ms** *address*, *page*, *length*, *filename*

Menu selection File→Save→Memory

Toolbar selection none

Environments ☑ basic debugger ☑ 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 a .obj extension.

next — *Single-Step, Next Statement*

Syntax **next** [*expression*]

Menu selection Debug→Next

Toolbar selection 

Environments ☑ basic debugger ☐ profiling

Description The NEXT command is similar to the STEP command. If you are in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time. Unlike STEP, NEXT never updates the display when executing called functions; NEXT always steps to the next consecutive statement. Unlike STEP, NEXT steps over function calls rather than stepping into them—you do not see the single-step execution of the function call.

The optional *expression* parameter specifies the number of statements that you want to single-step. You can use a conditional *expression* for conditional single-step execution. (Section 6.5, *Running Code Conditionally*, page 6-12, discusses this in detail.)

pause — *Pause Execution* — **Batch File Only**

Syntax **pause**

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description The PAUSE command allows you to pause the debugger while running a batch file. Pausing is especially helpful in debugging the commands in a batch file.

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

pf — *Profile, Full*

Syntax **pf** *starting point* [**,** *update rate*]

Menu selection Tools→Profile→Profile Mode
Debug→Run

Toolbar 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:

Value	Description
0	This is the default. Statistics are not updated until the session is halted (although you can force an update by clicking the mouse in the window).
≥ 1	Statistics are updated during the session. A value of **1** means that data is updated as often as possible.

pinc — *Connect Pin*

Syntax **pinc** *pinname, filename*

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The PINC command connects an input file to an interrupt pin.

- The *pinname* parameter identifies the interrupt pin and must be one of the 18 interrupt signals (INT1–INT14, DLONGINT, RTOSINT, NMI, or EMUINT).
- The *filename* parameter is the name of your input file.

pind — *Disconnect Pin*

Syntax **pind** *pinname*

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The PIND command disconnects an input file from an interrupt pin. The *pinname* parameter identifies the interrupt pin and must be one of the 18 interrupt signals (INT1–INT14, DLONGINT, RTOSINT, NMI, or EMUINT)18 interrupt signals (INT1–INT14, DLONGINT, RTOSINT, NMI, or EMUINT).

pinl — *List Pin*

Syntax **pinl**

Menu selection none

Environments ☑ basic debugger ☐ profiling

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

pq — *Profile, Quick*

Syntax **pq** *starting point* [**,** *update rate*]

Menu selection Tools→Profile→Profile Mode
Debug→Run

Toolbar 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 — *Resume Profiling Session*

Syntax **pr** [*clear data* [**,** *update rate*]]

Menu selection Tools→Profile→Profile Mode
Debug→Run

Toolbar selection 

Environments ☐ basic debugger ☑ profiling

Description The PR command resumes the last profiling session (initiated by PF or PQ), starting from the current program counter.

The optional *clear data* parameter tells the debugger whether or not it should clear out the previously collected data. The *clear data* parameter can have one of these values:

Value	Description
0	This is the default. The profiler continues to collect data (adding the data to the existing data for the profiled areas) and to use the previous internal profile stacks.
nonzero	All previously collected profile data and internal profile stacks are cleared.

The *update rate* parameter is the same as for the PF and PQ commands.

profile — *Switch to Profiling Environment*

Syntax **profile**

Menu selection Tools→Profile→Profile Mode

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description The PROFILE command toggles between the basic debugger and profiling environments. If you enter PROFILE from the basic debugger environment, the debugger switches to the profiling environment. If you enter PROFILE from the profiling environment, the debugger switches to the basic debugger environment.

prompt — *Change Command-Line Prompt*

Syntax **prompt** *new prompt*

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☑ 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.

quit — *Exit Debugger*

Syntax **quit**

Menu selection File→Exit

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description The QUIT command exits the debugger and returns to the operating system.

realtime — *Begin Realtime Mode Execution*

Syntax **realtime**

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The REALTIME command begins realtime mode. In realtime mode, the routine code execution stops and high priority interrupts continue to occur. When no debugger events occur, the 'C27xx in both stopmode and realtime mode continues to execute the routine code in the normal manner as well as to enable all interrupts.

reconnect — *Reset Communication With Emulator* — **Emulator Only**

Syntax **reconnect**

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description The RECONNECT command reinitializes communication between the debugger and the emulator. This command can be used after an unrecoverable fatal error.

Any software breakpoints set before a reconnect may still reside in memory after the reconnect. However, the debugger does not recognize that the breakpoints are set. You should reload memory in order to clear out any residual breakpoints.

reload — *Reload Object Code*

Syntax **reload** [*object filename*]

Menu selection File→Load→Reload Program

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description The RELOAD command loads only an object file *without* loading its associated symbol table. This is useful for reloading a program when target memory has been corrupted. If you enter the RELOAD command without specifying a filename, the debugger reloads the file that you loaded last.

reset — *Reset Target System*

Syntax **reset**

Menu selection Debug→Reset Target

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description The RESET command resets the target system (emulator only) or simulator. This is a *software* reset.

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

restart — *Reset PC to Program Entry Point*

Syntax
restart
rest

Menu selection
Debug→Restart

Toolbar selection


Environments
☑ basic debugger ☑ profiling

Description
The RESTART or REST command resets the program to its entry point. (This assumes that you have already used one of the load commands to load a program into memory.)

return — *Return to Function's Caller*

Syntax
return
ret

Menu selection
Debug→Return

Toolbar selection


Environments
☑ basic debugger ☐ profiling

Description
The RETURN or RET command executes the code in the current C function and halts when execution reaches the caller. Breakpoints do not affect this command, but you can halt execution by doing one of the following actions:

- Click the Halt icon on the toolbar:

- From the Debug menu, select Halt!.
- Press ESC.

run — *Run Code*

Syntax **run** [*expression*]

Menu selection Debug→Run

Toolbar selection 

Environments ☑ basic debugger ☐ profiling

Description The RUN command is the basic command for running an entire program. The command's behavior depends on the type of parameter you supply:

- If you do not supply an *expression*, the program executes until it encounters a breakpoint or until you do one of the following actions:
  - Click the Halt icon on the toolbar:

    
  - From the Debug menu, select Halt!.
  - Press ESC.
- If you supply a logical or relational *expression*, the run becomes conditional. (Section 6.5, *Running Code Conditionally*, page 6-12, discusses this in detail.)
- If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger executes *count* instructions, halts, and updates the display.

runb — *Benchmark Code*

Syntax **runb**

Menu selection Debug→Run Benchmark

Toolbar selection none

Environments ☑ basic debugger ☐ profiling

Description The RUNB command executes a specific section of code and counts the number of CPU clock cycles consumed by the execution. For RUNB to operate correctly, *execution must be halted by a software breakpoint*. After RUNB execution halts, the debugger stores the number of cycles into the CLK pseudoregister. For a complete explanation of the RUNB command and the benchmarking process, read section 6.6, *Benchmarking*, on page 6-13.

runf — *Run Free* — Emulator Only

Syntax **runf**

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☐ profiling

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

The HALT command stops a RUNF; the debugger automatically executes a HALT when the debugger is invoked.

sa — *Add Stopping Point*

Syntax **sa** *address*

Menu selection none

Toolbar selection none

Environments ☐ basic debugger ☑ profiling

Description The SA command adds a stopping point at *address*. The *address* can be a label, a function name, or a memory address.

sconfig — *Load Screen Configuration*

Syntax **sconfig** [*filename*]

Menu selection File→Load→Screen Layout

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description The SCONFIG command restores the display to a specified configuration. This restores the window locations and sizes that were saved with the SSAVE command into *filename*. The debugger searches for the specified file in the current directory and then in directories named with the D_DIR environment variable. If you do not supply a *filename*, the debugger looks for init.clr.

When you use SCONFIG to restore a configuration that includes multiple File, Watch, or Memory windows, the additional windows are not opened automatically. However, when you open an additional window and use a *window name* that matches a window name that you used before you saved the configuration, the window is placed in the saved location.

sd — *Delete Stopping Point*

Syntax **sd** *address*

Menu selection none

Toolbar selection none

Environments ☐ basic debugger ☑ 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:

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

Only a subset of the display formats can be used for each data type. Listed below are the valid combinations of data types and display formats.

	Valid Display Formats										Valid Display Formats									
Data Type	c	d	o	x	e	f	p	s	u	Data Type	c	d	o	x	e	f	p	s	u
char (c)	√	√	√	√					√	long (d)	√	√	√	√					√
uchar (d)	√	√	√	√					√	ulong (d)	√	√	√	√					√
short (d)	√	√	√	√					√	float (e)			√	√	√	√			
int (d)	√	√	√	√					√	double (e)			√	√	√	√			
uint (d)	√	√	√	√					√	ptr (p)			√	√			√	√	

To return all data types to their default display format, enter:

```
setf * ↵
```

sim — *Enable Simulation Execution Mode* — **Simulator Only**

Syntax **sim**

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☐ profiling

Description The SIM command selects the simulation execution mode. When you halt the debugger, the pipeline is not flushed.

The simulator supports two modes of execution: simulation and emulation. The principal difference between the modes is in the state of the pipeline when execution is halted. Section 2.9, *Execution Modes*, on page 2-17 discusses this in detail.

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 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→Program 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→Screen Layout New File

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 and you save and reload the screen configuration, the debugger does not display any windows as docked. If you want the windows docked, you must follow the docking procedure again.

The *filename* parameter names the screen configuration file. You can include path information (including relative pathnames); if you do not supply path information, the debugger places the file in the current directory. If you do not supply a *filename*, the debugger saves the current configuration into a file named init.clr and places the file in the current directory.

If you use a filename that already exists, the debugger overwrites the file with the current configuration.

step — *Single-Step*

Syntax **step** [*expression*]

Menu selection Debug→Step

Toolbar selection 

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. (Section 6.5, *Running Code Conditionally*, page 6-12, discusses this in detail.)

stepcycle — *Clock (Cycle) Step*

Syntax **stepcycle** [*expression*]

Menu selection Debug→Clock Step

Toolbar selection 

Environments ☑ basic debugger ☐ profiling

Description The STEPCYCLE command cycle-steps through assembly language code. In assembly or mixed mode, the simulator executes one cycle at a time.

If you want to see what stage of the pipeline the cycle-stepped instructions are in, you must turn on the pipeline display first. To display the pipeline stages, select Pipeline Display from the Disassembly window context menu. (Section 5.2, *Displaying Pipeline Stages With Assembly Language Code*, page 5.2, discusses this in detail.)

The *expression* parameter specifies the number of statements that you want to single-step. You can use a conditional *expression* for conditional single-step execution. (Section 6.5, *Running Code Conditionally*, page 6-12, discusses this in detail.)

Note:

Cycle-step is available only in simulation mode. If you are running the simulator in emulation execution mode, cycle-step is disabled. See section 2.9, *Execution Modes*, on page 2-17 for information on how the simulator responds in the two execution modes.

stopmode — *Begin Stop Mode Execution*

Syntax stopmode

Menu selection none

Environments ☑ basic debugger ☐ profiling

Description The STOPMODE command begins stopmode execution. In stopmode, the ’C27xx execution stops and the debugger controls all further action. When no debugger events occur, the ’C27xx in both stopmode and realtime mode continues to execute the routine code in the normal manner as well as to enable all interrupts.

system — *Enter Operating-System Command*

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

Menu selection none

Toolbar selection none

Environments ☑ basic debugger ☑ profiling

Description The debugger version of the SYSTEM command allows you to enter operating-system commands without explicitly exiting the debugger environment. If you enter SYSTEM with no parameters, the debugger opens a system shell and displays the operating-system prompt. At this point, you can enter any operating-system command. When you finish, enter:

```
exit ↵
```

If you prefer, you can supply the operating-system command as a parameter to the SYSTEM command. If the result of the command is a message or other display, the debugger blanks the top of the debugger display to show the information. In this case, you can use the *flag* parameter to tell the debugger whether or not it should hesitate after displaying the information. The *flag* can be 0 or 1.

0 If you supply a value of 0 for *flag*, the debugger immediately returns to the debugger environment after the last item of information is displayed.

1 If you supply a value of 1 for *flag*, the debugger does not return to the debugger environment until you enter:

`exit ↵.`

(This is the default.)

take — *Execute Batch File*

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

Menu selection File→Execute Take File

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* | *****}

Menu selection Configure→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 — *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 Tools→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 Tools→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 version number, as well as the device name.

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.

wa — *Add Item to Watch Window*

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

Menu selection Configure→Watch Add

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:

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

If you want to use a *display format* parameter without a *label* parameter, be sure to include an extra comma. For example:

```
wa PC,,o ↵
```

You can open additional Watch windows by using the *window name* parameter. When you open an additional Watch window, the debugger appends the *window name* to the Watch window label. You can create as many Watch windows as you need.

If you omit the *window name* parameter, the debugger displays the expression in the default Watch window (labeled Watch).

wd — *Delete Item From Watch Window*

Syntax **wd** *expression* [**,** *window name*]

Menu selection Configure→Watch Add

Toolbar selection none

Environments ☑ basic debugger ☐ 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 Configure→Watch Add

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.

11.3 Summary of Profiling Commands

The following tables summarize the profiling commands that are used for marking, enabling, disabling, and unmarking areas and for changing the display in the Profile window. These commands are easiest to use from the Tools→Profile menu and associated dialog boxes, so they are not included in the alphabetical command summary. The syntaxes for these commands are provided here so that you can include them in batch files.

Table 11–1. Marking areas

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

Table 11–2. Disabling marked areas

To disable this area...	In C only	In disassembly only	In C *and* disassembly
Lines			
❑ By line number, address	**DCLE** *filename*, *line number*	**DALE** *address*	not applicable
❑ All lines in a function	**DCLF** *function*	**DALF** *function*	**DBLF** *function*
❑ All lines in a module	**DCLM** *filename*	**DALM** *filename*	**DBLM** *filename*
❑ All lines everywhere	**DCLG**	**DALG**	**DBLG**
Ranges			
❑ By line number, address	**DCRE** *filename*, *line number*	**DARE** *address*	not applicable
❑ All ranges in a function	**DCRF** *function*	**DARF** *function*	**DBRF** *function*
❑ All ranges in a module	**DCRM** *filename*	**DARM** *filename*	**DBRM** *filename*
❑ All ranges everywhere	**DCRG**	**DARG**	**DBRG**

To disable this area...	In C only	In disassembly only	In C *and* disassembly
Functions			
❑ By function name	**DCFE** *function*	not applicable	not applicable
❑ All functions in a module	**DCFM** *filename*		**DBFM** *filename*
❑ All functions everywhere	**DCFG**		**DBFG**
All areas			
❑ All areas in a function	**DCAF** *function*	**DAAF** *function*	**DBAF** *function*
❑ All areas in a module	**DCAM** *filename*	**DAAM** *filename*	**DBAM** *filename*
❑ All areas everywhere	**DCAG**	**DAAG**	**DBAG**

Table 11–3. Enabling disabled areas

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

Table 11–4. Unmarking areas

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

Table 11–5. Changing the profile window display

(a) Viewing specific areas

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

(b) Viewing different data

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

(c) Sorting the data

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

Chapter 12

Basic Information About C Expressions

Many of the debugger commands take C expressions as parameters. This allows the debugger to have a relatively small, yet powerful, instruction set. Because C expressions can have side effects—that is, the evaluation of some types of expressions can affect existing values—you can use the same command to display or to change a value. This reduces the number of commands in the command set.

This chapter contains basic information that helps you use C expressions as debugger command parameters.

Topic **Page**

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	–	negation (unary)
(*type*)	type cast		

❑ **Relational and logical operators**

>	greater than	>=	greater than or equal to
<	less than	<=	less than or equal to
==	is equal to	!=	is not equal to
&&	logical AND	||	logical OR
!	logical NOT (unary)		

❑ Increment and decrement operators

++	increment	– –	decrement

These unary operators can precede or follow a symbol. When the operator precedes a symbol, the symbol value is incremented/decremented before it is used in the expression; when the operator follows a symbol, the symbol value is incremented/decremented after it is used in the expression. Because these operators affect the symbol's final value, the parameters they are used with have side effects.

❑ Bitwise operators

&	bitwise AND	|	bitwise OR
^	bitwise exclusive-OR	<<	left shift
>>	right shift	~	1s complement (unary)

❑ Assignment operators

=	assignment	+=	assignment with addition
–=	assignment with subtraction	/=	assignment with division
%=	assignment with modulo	&=	assignment with bitwise AND
^=	assignment with bitwise XOR	|=	assignment with bitwise OR
<<=	assignment with left shift	>>=	assignment with right shift
*=	assignment with multiplication		

These operators support a shorthand version of the familiar binary expressions; for example, X = X + Y can be written in C as X += Y. Because these operators affect a symbol's final value, the parameters they are used with have side effects.

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.

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 'C27xx auxiliary registers are treated as integers and/or pointers.
- ❏ Void expressions are legal (treated like integers).
- ❏ The specification of variables and functions can be qualified with context information. Local variables (including local statics) can be referenced with the expression form:

  *function name***.***local name*

This expression format is useful for examining the automatic variables of a function that is not currently being executed. Unless the variable is static, however, the function must be somewhere in the current call stack. If you want to see local variables from the currently executing function, you need not use this form; you can simply specify the variable name (just as in your C source).

File-scoped variables (such as statics or functions) can be referenced with the following expression form:

*filename***.***function name*
or *filename***.***variable name*

This expression format is useful for accessing a file-scoped static variable (or function) that may share its name with variables in other files.

In this expression form, *filename* **does not include** the file extension; the debugger searches the object symbol table for any source filename that matches the input name, disregarding any extension. Thus, if the variable *ABC* is in file source.c, you can specify it as source.ABC.

These expression forms can be combined into an expression of the form:

*filename***.***function name***.***variable name*

- Any integral or void expression can be treated as a pointer and used with the indirection operator (*). Here are several examples of valid use of a pointer in an expression:

```
*123
*A5
*(A2 + 123)
*(I*J)
```

  By default, the values are treated as integers (that is, these expressions point to integer values).

- Any expression can be type cast to a pointer to a specific type (overriding the default of pointing to an integer, as described above).

  Hint: You can use casting with the WA and DISP commands to display data in a desired format.

  For example, the expression:

```
*(float *)10
```

  treats 10 as a pointer to a floating-point value at location 10 in memory. In this case, the debugger fetches the contents of memory location 10 and treats the contents as a floating-point value. If you use this expression as a parameter for the DISP command, the debugger displays memory contents as an array of floating-point values within the DISP window, beginning with memory location 10 as array member [0].

Note how the first expression differs from the expression:

```
(float)*10
```

In this case, the debugger fetches an integer from address 10 and converts the integer to a floating-point value.

You can also type cast to user-defined types such as structures. For example, in the expression:

```
((struct STR *)10)–>field
```

The debugger treats memory location 10 as a pointer to a structure of type STR (assuming that a structure is at address 10) and accesses a field from that structure.

Appendix A

What the Debugger Does During Invocation

In some circumstances, you may find it helpful to know the steps that the debugger goes through during the invocation process. These are the steps, in order, that the debugger performs. (For more information on the environment variables mentioned below, see Chapter 2, *Getting Started With the Debugger.*)

The debugger:

1) Reads options from the operating system's command line.
2) Reads any information specified with the D_OPTIONS environment variable.
3) Reads information from the D_DIR and D_SRC environment variables.
4) Looks for the init.clr screen-configuration file.

   (The debugger searches for the screen-configuration file in directories named with D_DIR.)
5) Initializes the debugger screen and windows.
6) Finds the batch file that defines your memory map by searching in directories named with D_DIR. The debugger expects this file to set up the memory map and follows these steps to look for the batch file:

   - When you invoke the debugger, it checks to see if you have used the –t debugger option. If it finds the –t option, the debugger reads and executes the specified file.
   - If you have not used the –t option, the debugger looks for the default initialization batch file emuinit.cmd. The batch file name differs for each version of the debugger:
     - For the emulator, this file is named *emuinit.cmd.*
     - For the simulator, this file is named *siminit.cmd.*

If the debugger finds the file corresponding to your tool, it reads and executes the file. If the debugger does not find the –t option or the initialization batch file, it looks for a file called init.cmd. This allows you to have one initialization batch file for more than one debugger tool. To set up this file, you can use the IF/ELSE/ENDIF commands (see page 3-8 for more information) to indicate which memory map applies to each tool.

7) Loads any object files specified with D_OPTIONS or specified on the command line during invocation.

8) Determines the initial mode (auto, assembly, or mixed) and displays the appropriate windows on the screen.

At this point, the debugger is ready to process any commands that you enter.

Where the debugger looks for files

You can perform all load-type commands by using menu options. However, if you choose to use the command-line equivalents to these menu options, you need to know where the debugger looks for source files.

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

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

    cd *directory name*

- If you are using the FILE command, you have several options:
  - Within the operating-system environment, you can name additional directories with the D_SRC environment variable. The format for this environment variable is:

    SET D_SRC=*pathname;pathname*

    You can name several directories for the debugger to search.
  - When you invoke the debugger, you can use the – i option to name additional source directories for the debugger to search. The format for this option is **–i** *pathname.*

You can specify multiple pathnames by using several –i options (one pathname per option). The list of source directories that you create with –i options is valid until you quit the debugger.

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

  use *directory name*

  You can specify only one directory at a time.

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

Appendix B

Describing Your Target System to the Debugger

For the debugger to understand how you have configured your target system, you must supply the target configuration information in a file for the debugger to read.

- If you are using an emulation scan path that contains only one 'C27xx and no other devices, you can use the *board.dat* file that comes with the 'C27xx emulator kit. This file describes to the debugger the single 'C27xx in the scan path and gives the 'C27xx the name CPU_A. Because the debugger automatically looks for a file called board.dat in the current directory and in the directories specified with the D_DIR environment variable, you can skip this appendix.

- If you plan to use a target system that has multiple 'C27xx devices or that includes devices other than the 'C27xx, you must follow these steps:

  Step 1: Create the board configuration text file.

  Step 2: Translate the board configuration text file to a binary, structured format so that the debugger can read it.

  Step 3: Specify the formatted configuration file when invoking the debugger.

  These steps are described in this appendix.

Topic **Page**

B.1 Step 1: Create the Board Configuration Text File

To describe the emulation scan path of your target system to the debugger, you must create a board configuration file. Each entry of the file describes one device on your scan path and the entries follow the order of the devices in the scan path. The text version of the configuration file is referred to as *board.cfg* in this book.

Example B–1 shows a board.cfg file that describes a possible 'C27xx device chain. It lists six octals named A1–A6, followed by five 'C27xx devices named CPU_A, CPU_B, CPU_C, CPU_D, and CPU_E.

Example B–1. A Sample TMS320C27xx Device Chain

(a) A sample board.cfg file

Device Name	Device Type	Comments
"A1"	BYPASS08	;the first device nearest TDO ;(test data out)
"A2"	BYPASS08	;the next device nearest TDO
"A3"	BYPASS08	
"A4"	BYPASS08	
"A5"	BYPASS08	
"A6"	BYPASS08	
"CPU_A"	TMS320C27xx	;the first 'C27xx
"CPU_B"	TMS320C27xx	
"CPU_C"	TMS320C27xx	
"CPU_D"	TMS320C27xx	
"CPU_E"	TMS320C27xx	;the last 'C27xx nearest TDI ;(test data in)

(b) A sample 'C27xx device chain

The order in which you list each device is important. The emulator scans the devices, assuming that the data from one device is followed by the data of the next device on the chain. Data from the device that is closest to the emulation header's TDO (test data out) reaches the emulator first. The device whose data reaches the emulator first is listed first in the board.cfg file; the device whose data reaches the emulator last is listed last in the board.cfg file.

The board.cfg file can have any number of each of these three types of entries:

- **Debugger devices** such as the 'C27xx. These are the only devices that the debugger can recognize.
- The **TI ACT8997 scan path linker**, or **SPL**. The SPL allows you to have up to four secondary scan paths that can each contain debugger devices ('C27xxs) and other devices.
- **Other devices**. These are any other devices in the scan path. These devices cannot be debugged and must be worked around or bypassed when trying to access the 'C27xxs.

Each entry in the board.cfg file consists of at least two pieces of data:

- **The name of the device.** The device name always appears first and is enclosed in double quotes:

  "*device name***"**

  This is the same name that you use with the –n debugger option, which tells the debugger the name of the 'C27xx. The *device name* can consist of up to eight alphanumeric characters or underscore characters and must begin with an alphabetic character.

- **The type of the device.** The debugger supports the following device types:

  - **TMS320C27xx** is an example of a debugger-device type. TMS320C27xx describes the 'C27xx.
  - **SPL** specifies the scan path linker and must be followed by four subpaths, as in this syntax:

    "*device name"* **SPL {***subpath0***} {***subpath1***} {***subpath2***} {***subpath3***}**

    Each *subpath* can contain any number of devices. However, an SPL subpath *cannot* contain another SPL. A subpath that contains no devices must still be listed.

Example B–2 shows a file that contains an SPL.

Example B–2. A board.cfg File Containing an SPL

Device Name	Device Type	Comments
″A1″	BYPASS08	;the first device nearest TDO
″A2″	BYPASS08	
″CPU_A″	TMS320C27xx	;the first ’C27xx
″HUB″	SPL	;the scan path linker
{		;the first subpath
″B1″	BYPASS08	
″B2″	BYPASS08	
″CPU_B″	TMS320C27xx	;the second ’C27xx
}		
{		;the second subpath
″C1″	BYPASS08	
″C2″	BYPASS08	
″CPU_C″	TMS320C27xx	;the third ’C27xx
}		
{		;the third subpath (contains nothing)
}		
{		;the fourth subpath
″D1″	BYPASS08	
″D2″	BYPASS08	
″CPU_D″	TMS320C27xx	;the fourth ’C27xx
}		
″CPU_E″	TMS320C27xx	;the last ’C27xx nearest TDI

Note: The indentation in the file is for readability only.

B.2 Step 2: Translate the Configuration File to a Debugger-Readable Format

After you have created the board.cfg file, you must translate it from text to a binary, conditioned format so that the debugger can understand it. To translate the file, use the composer utility that is included with the emulator kit. At the system prompt, enter the following command:

composer [*input file* [*output file*]]

- The *input file* is the name of the board.cfg file that you created in step 1; if the file is not in the current directory, you must supply the entire pathname. If you omit the input filename, the composer utility looks for a file called board.cfg in your current directory.
- The *output file* is the name that you can specify for the resulting binary file; ideally, use the name board.dat. If you want the output file to reside in a directory other than the current directory, you must supply the entire pathname. If you omit an output filename, the composer utility creates a file called board.dat and places it in the current directory.

To avoid confusion, use a .cfg extension for your text filenames and a .dat extension for your binary filenames. If you enter only one filename on the command line, the composer utility assumes that it is an input filename.

B.3 Step 3: Specifying the Configuration File When Invoking the Debugger

When you invoke a debugger, the debugger must be able to find the board.dat file so that it knows how you have set up your scan path. The debugger looks for the board.dat file in the current directory and in the directories named with the D_DIR environment variable.

If you used a name other than board.dat or if the board.dat file is not in the current directory or in a directory named with D_DIR, you must use the –f option when you invoke the debugger. The –f option allows you to specify a board configuration file (and pathname) to be used instead of board.dat. The format for this option is:

–f *filename*

Appendix C

Debugger Messages

This appendix contains an alphabetical listing of the progress and error messages that the debugger might display in the display area of the Command window. Each listing contains both a description of the situation that causes the message and an action to take if the message indicates a problem or error.

Topic **Page**

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-21.

‘)’ 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-21.

A

Aborted by user

Description The debugger halted a long Command display listing because you pressed the ESC key.

Action None required; this is normal debugger behavior.

B

Breakpoint already exists at *address*

Description During single-step execution, the debugger attempted to set a breakpoint where one already existed. (This is not necessarily a breakpoint that you set—it may have been an internal breakpoint that the debugger set for single-stepping).

Action None should be required; you may want to reset the program entry point (Debug→Restart) and reenter the single-step command.

Breakpoint table full

Description 200 breakpoints are already set, and there was an attempt to set another. The maximum limit of 200 breakpoints includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.

Action Open the Breakpoint Control dialog box by selecting Breakpoints from the Configure menu. Delete individual software breakpoints.

C

Cannot allocate host memory

Description This is a fatal error—it means that the debugger is running out of memory.

Action You can invoke the debugger with the –v option so that fewer symbols may be loaded, or you can relink your program and link in fewer modules at a time.

Cannot allocate system memory

Description This is a fatal error—it means that the debugger is running out of memory.

Action You can invoke the debugger with the –v option so that fewer symbols may be loaded, or you can relink your program and link in fewer modules at a time.

Cannot connect file to program memory

Description An attempt has been made to connect a file to program memory using the MC command.

Action You cannot connect a file to any location in program memory using the MC command.

Cannot detect target power

Description This hardware error occurs after the emurst command is reset. Follow the steps described below and then restart your emulator.

Action

- ❑ Check the emulator board to be sure it is installed snugly.
- ❑ Check the cable connecting your emulator and target system to be sure it is not loose.
- ❑ Check your target board to be sure it is getting the correct voltage.
- ❑ Check your emulator scan path to be sure it is uninterrupted.
- ❑ Ensure that your port address is set correctly:
  - ■ Check to be sure the –p option used with the D_OPTIONS environment variable matches the I/O address defined by your switch settings. (See page 2-5 for more information on the D_OPTIONS environment variable.)
  - ■ Check to see if you have a conflict in address space with another bus setting. If you have a conflict, change the switches on your board to one of the alternate settings listed in the installation guide. Modify the –p option of the D_OPTIONS environment variable to reflect the change in your switch settings.

Cannot edit field

Description Expressions that are displayed in the Watch window cannot be edited.

Action If you attempted to edit an expression in the Watch window, you may have actually wanted to change the value of a symbol or register used in the expression. Use the ? or EVAL command to edit the actual symbol or register. The expression value is automatically updated.

Cannot find/open initialization file

Description The debugger cannot find the init.cmd file.

Action Be sure that init.cmd is in the appropriate directory. If it is not, copy it from the debugger product diskette. If the file is already in the correct directory, verify that the D_DIR environment variable is set up to identify the directory. See the information about setting up the debugger environment information included with your installation instructions.

Cannot halt the processor

Description This is a fatal error—for some reason, pressing ESC did not halt program execution.

Action Exit the debugger. Invoke the autoexec.bat file, then invoke the debugger again.

Cannot initialize target system

Description This error occurs while you are invoking the debugger with the emulator. A variety of events may cause this error to occur.

Action

- ❑ Check the cable connecting the emulator to the target system to be sure it is not loose.
- ❑ Ensure that your port address is set correctly:
  - ■ Check to be sure the –p option used with the D_OPTIONS environment variable matches the I/O address defined by your switch settings.
  - ■ Check to see if you have a conflict in address space with another bus setting. If you have a conflict, change the switches on your board to one of the alternate settings listed in the installation guide. Modify the –p option of the D_OPTIONS environment variable to reflect the change in your switch settings.
- ❑ Check the end of your autoexec.bat or initdb.bat file for the emurst.exe command. Execute this command *after* powering up the target board. See section 2.5 on page 2-6.

For more information on setting up the D_OPTIONS environment variable, see page 2-5 .

Cannot map into reserved memory: ?

Description The debugger tried to access unconfigured/reserved/nonexistent memory.

Action Remap the reserved memory accesses.

Cannot map port address

Description You attempted to do a connect/disconnect on an illegal port address.

Action Be sure that you are connecting to or disconnecting from an address that is mapped in as an input, output, or I/O port.

Cannot open config file

Description The SCONFIG command cannot find the screen-customization file that you specified. The debugger also displays this message when you try to load a screen-customization file that was saved by an older version of the debugger.

Action

- ❑ Be sure that the filename was typed correctly. If it was not, reenter the command with the correct name. If it was, reenter the command and specify full path information with the filename.
- ❑ Be sure that the screen-customization file was saved using the current version of the debugger rather than an older version of the debugger.

Cannot open "*filename*"

Description The debugger attempted to show *filename* in the File window but could not find the file.

Action Be sure that the file exists as named. If it does, enter the USE command to identify the file's directory.

Cannot open new window

Description A maximum of 127 windows can be open at once. The last request to open a window would have made 128, which is not possible.

Action Close any unnecessary windows. Windows that can be closed include Watch, File, Calls, and Memory windows. To close any of these windows, make the desired window active and press CONTROL F4 .

Cannot open object file: "*filename*"

Description The file specified with the LOAD, SLOAD, or RELOAD command is not an object file that the debugger can load.

Action Be sure that you are loading an actual object file. Be sure that the file was linked. You may want to run cl27 (with the –z option) or lnk27 again to create an executable object file.

Cannot read processor status

Description This is a fatal error—for some reason, pressing ESC did not halt program execution.

Action Exit the debugger. Invoke the autoexec.bat file, then invoke the debugger again. If you are using the emulator, check the cable connections also.

Cannot reset the processor

Description This is a fatal error—for some reason, pressing ESC did not halt program execution.

Action Exit the debugger. Invoke the autoexec.bat file, then invoke the debugger again. If you are using the emulator, check the cable connections, also.

Cannot restart processor

Description The debugger attempted to reset the PC to the program entry point, but the debugger could not find an entry point.

Action Either define an entry point in your program, or do not use Debug→Restart or RESTART when your program does not have an explicit entry point.

Cannot set/verify breakpoint at *address*

Description Either you attempted to set a breakpoint in read-only or protected memory, or there are hardware problems with the target system. This may also happen when you enable or disable on-chip memory while using breakpoints.

Action Check your memory map. If the address that you wanted to breakpoint was not in ROM, see section C.4, *Additional Instructions for Hardware Errors*, page C-21.

Cannot step

Description There is a problem with the target system.

Action See section C.4, *Additional Instructions for Hardware Errors*, page C-21.

Cannot take address of register

Description This is an expression error. C does not allow you to take the address of a register.

Action See section C.3, *Additional Instructions for Expression Errors*, page C-21.

Command "*command*" not found

Description The debugger did not recognize the command that you typed.

Action Reenter the correct command. See Chapter 11, *Summary of Commands*.

Command timed out, emulator busy

Description There is a problem with the target system.

Action See section C.4, *Additional Instructions for Hardware Errors*, page C-21.

Conflicting map range

Description A block of memory specified with the Configure→Memory Maps menu option or the MA command overlaps an existing memory map entry. Blocks cannot overlap.

Action Use Configure→Memory Maps or the ML command to list the existing memory map; this helps you find the existing block that the new block would overlap. If the existing block is not necessary, delete it with the Memory Map Control dialog box or with the MD command. Use the Memory Map Control dialog box or the MA command to redefine the block of memory. If the existing block is necessary, use the Memory Map Control dialog box or the MA command to define a range that does not overlap the existing block.

Corrupt call stack

Description The debugger tried to update the Calls window and could not. This message is displayed in the following situations:

- ❏ A function was called that did not return.
- ❏ The program stack was overwritten in target memory.
- ❏ You are debugging code that has optimization enabled (for example, you did not use the –g compile option); if this is the case, ignore this message—code execution is not affected.

Action If your program called a function that did not return, then this is normal behavior (as long as you intended for the function not to return). Otherwise, you may be overwriting program memory.

EMU is jammed in D2

Description While you were cycle stepping with the pipeline display on, an emulation interrupt occurred during the Decode2 phase of the pipeline. That interrupt service routine is in control of the debugger.

Action Wait for the interrupt routine to complete processing. Control of the debugger returns to the Disassembly window automatically.

Emulator I/O address is invalid

Description The debugger was invoked with the –p option, and an invalid *port address* was used.

Action For valid *port address* values, see page 2-10.

EOF reached –connected at port: <*memory addr*>

Description The last data of the input file has been read.

Action You can disconnect the file with the MI command and connect a new file with the MC command. If you do not do anything and resume execution, then the input file automatically rewinds, and input data is read from the beginning of the file.

Error in expression

Description This is an expression error.

Action See section C.3, *Additional Instructions for Expression Errors*, page C-21.

Execution error

Description There is a problem with the target system.

Action See section C.4, *Additional Instructions for Hardware Errors*, page C-21.

F

File already tied to port

Description You attempted to connect to an address that already has a file connected to it.

Action Connect the file to a mapped port that is not connected to a file.

File already tied to this pin

Description You attempted to connect an input file to an interrupt pin that already has a file connected to it.

Action Use the PINC command to connect the file to another interrupt pin that is not connected to a file.

File does not exist

Description The port file could not be opened for reading.

Action Be sure that the file exists as named. If it does, enter the USE command to identify the file's directory.

Files must be disconnected from ports

Description You attempted to delete a memory map that has files connected to it.

Action You must disconnect a port with the MI command before you can delete it from the memory map.

File not found

Description The filename specified for the FILE command was not found in the current directory or any of the directories identified with D_SRC.

Action Be sure that the filename was typed correctly. If it was, reenter the FILE command and specify full path information with the filename.

File not found : "*filename*"

Description The filename specified for the LOAD, RELOAD, SLOAD, or TAKE command was not found in the current directory or any of the directories identified with D_SRC.

Action Be sure that the filename was typed correctly. If it was, reenter the command and specify full path information with the filename.

File too large (*filename*)

Description You attempted to load a file that exceeded the maximum loadable COFF file size.

Action Loading the file without the symbol table (SLOAD), or use cl27 (with the –z option) or lnk27 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-21.

Function required

Description The parameter for the FUNC command must be the name of a function in the program that is loaded.

Action Reenter the FUNC command with a valid function name.

I

Illegal cast

Description This is an expression error—the expression parameter uses a cast that does not meet the C language rules for casts.

Action See section C.3, *Additional Instructions for Expression Errors*, page C-21.

Illegal left hand side of assignment

Description This is an expression error—the left-hand side of an assignment expression does not meet C language assignment rules.

Action See section C.3, *Additional Instructions for Expression Errors*, page C-21.

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-21.

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-21.

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-21.

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-21.

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-21.

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-21.

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-21.

Interrupt *number* is jammed in D2

Description While you were cycle stepping with the pipeline display on, an external interrupt, *number*, occurred during the Decode2 phase of the pipeline. That interrupt service routine is in control of the debugger.

Action Wait for the interrupt routine to complete processing. Control of the debugger returns to the Disassembly window automatically.

Invalid address
––– Memory access outside valid range: *address*

Description The debugger attempted to access memory at *address*, which is outside the memory map.

Action Check your memory map to be sure that you access valid memory.

Invalid argument

Description One of the command parameters does not meet the requirements for the command.

Action Reenter the command with valid parameters. Refer to the appropriate command description in Chapter 11, *Summary of Commands*.

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:

SARAM	(single-access read/write memory)
DARAM	(dual-access read/write memory)
FLASH	(nonvolatile reprogrammable memory)
R, ROM	(read-only memory)
W, WOM	(write-only memory)
R|W, RAM	(read/write memory)
EX|RAM	(external read/write memory)
EX|ROM	(external write memory)

Invalid object file

Description Either the file specified with File→Load→Load Program, File→Load→Reload Program, File→Load→Program Symbols, the LOAD, the SLOAD, or the RELOAD command is not an object file that the debugger can load, or it has been corrupted.

Action Be sure that you are loading an actual object file. Be sure that the file was linked. You may want to run cl27 (with the –z option) or lnk27 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 cl27.

Invalid watch delete

Description The debugger cannot delete the parameter supplied with the WD command.

Action Reenter the WD command. Be sure to specify the symbol name that matches the item you want to delete.

Invalid window position

Description The debugger cannot move the window to the XY position specified with the MOVE command. Either the XY parameters are not within the screen limits, or the active window may be too large to move to the desired position.

Action Reenter the MOVE command. Enter the X and Y parameters in pixels.

Invalid window size

Description The width and length specified with the SIZE or MOVE command may be too large or too small. If valid width and length were specified, then the active window is already at the far right or bottom of the screen and so cannot be made larger.

Action Reenter the SIZE command. Enter the width and length in pixels.

L

Load aborted

Description This message always follows another message.

Action Refer to the message that preceded *Load aborted*.

Lost power (or cable disconnected)

Description Either the target cable is disconnected, or the target system is faulty.

Action Check the target cable connections. If the target seems to be connected correctly, see section C.4, *Additional Instructions for Hardware Errors*, page C-21.

Lost processor clock

Description Either the target cable is disconnected, or the target system is faulty.

Action Check the target cable connections. If the target seems to be connected correctly, see section C.4, *Additional Instructions for Hardware Errors*, page C-21.

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-21.

M

Memory access error at *address*

Description Either the processor is receiving a bus fault, or there are problems with target system memory.

Action See section C.4, *Additional Instructions for Hardware Errors*, page C-21.

Memory map table full

Description Too many blocks have been added to the memory map. This rarely happens unless blocks are added word by word (which is inadvisable).

Action Stop adding blocks to the memory map. Consolidate any adjacent blocks that have the same memory attributes.

N

Name "*name*" not found

Description The command cannot find the object named *name*.

Action If *name* is a symbol, be sure that it was typed correctly. If it was not, reenter the command with the correct name. If it was, then be sure that the associated object file is loaded.

Nesting of repeats cannot exceed 100

Description The debugger cannot simulate more than 100 levels of repeat nesting in an input data file. If more than 100 instances are requested, the debugger disconnects the input file from the pin.

Action Correct the input file so that the data does not include nesting repetition exceeding 100. Use the PINC command to reconnect the input file to the desired pin.

NMI is jammed in D2

Description While you were cycle stepping with the pipeline display on, the non-maskable interrupt occurred during the Decode2 phase of the pipeline. That interrupt service routine is in control of the debugger.

Action Wait for the interrupt routine to complete processing. Control of the debugger returns to the Disassembly window automatically.

No file connected to this pin

Description You tried to disconnect the input file from a pin that was not previously connected to that pin.

Action Use the PINL command to list all of the pins and the files connected to them. Use the PIND command to reenter the correct pinname and filename.

P

Pinname not valid for this chip

Description You attempted to connect or disconnect an input file to an invalid interrupt pin.

Action Reconnect or disconnect the input file to an unused interrupt pin ($\overline{\text{INT1}}$–$\overline{\text{INT14}}$, $\overline{\text{DLONGINT}}$, $\overline{\text{RTOSINT}}$, $\overline{\text{NMI}}$, or $\overline{\text{EMUINT}}$).

Pipeline Stall – Front end buffer is empty

Description The first three stages of the pipeline, which are decoupled from the remaining five stages, have nothing to feed into the Decode2 phase.

Action Reading instructions from slow memory is the most common cause of this error.

Pipeline Stall – Memory pipeline conflict

Description The highlighted instruction cannot execute the Decode2 phase. The instruction reads from a memory location that is in use by a previous instruction.

Action Reorganize your code so that this instruction occurs after the previous instruction has completed writing to the memory location.

Pipeline Stall – READY is pulled low by memory

Description The READY signal being pulled low indicates that the memory bank is not ready to respond. The pipeline is stalled until the READY signal goes high.

Action Refer to the *TMS320C27xx DSP CPU and Instruction Set User's Guide* for more information on the READY signal.

Pipeline Stall – Register pipeline conflict

Description The highlighted instruction cannot execute the Decode2 phase. The instruction reads from or writes to a register that is in use by a previous instruction.

Action Reorganize your code so that this instruction occurs after the previous instruction has completed writing to the register.

Pointer not allowed

Description This is an expression error.

Action See section C.3, *Additional Instructions for Expression Errors*, page C-21.

Processor is already running

Description One of the RUN commands was entered while the debugger was running free from the target system.

Action Enter the HALT command to stop the free run, then reenter the desired RUN command.

R

Read not allowed for port

Description You attempted to connect a file for input operation to an address that is not configured for read.

Action Remap the port of correct the access in your source code.

Register access error

Description Either the processor is receiving a bus fault, or there are problems with target-system memory.

Action See section C.4, *Additional Instructions for Hardware Errors*, page C-21.

S

Specified map not found

Description The MD command was entered with an address or block that is not in the memory map.

Action Use the ML command to verify the current memory map. When using MD, you can specify only the first address of a defined block.

Structure member name required

Description This is an expression error—a symbol name is followed by a period but no member name.

Action See section C.3, *Additional Instructions for Expression Errors*, page C-21.

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-21.

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-21.

T

Take file stack too deep

Description Batch files can be nested up to ten levels deep. The batch file that you tried to execute with File→Execute Take File or the TAKE command calls batch files that are nested more than ten levels deep.

Action Edit the batch file that caused the error. Instead of calling another batch file from within the offending file, you can copy the contents of the second file into the first. This will removes a level of nesting.

Too many breakpoints

Description 200 breakpoints are already set, and there was an attempt to set another. The maximum limit of 200 breakpoints includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.

Action Open the Breakpoint Control dialog box by selecting Breakpoints from the Configure menu. Delete individual software breakpoints.

Too many paths

Description More than 20 paths have been specified cumulatively with the USE command, D_SRC environment variable, and –i debugger option.

Action Do not enter the USE command before entering another command that has a *filename* parameter. Instead, enter the second command and specify full path information for the *filename*.

U

Undeclared port address

Description You attempted to do a connect/disconnect on an address that is not declared as a port.

Action Verify the address of the port to be connected or disconnected.

User halt

Description The debugger halted program execution because you clicked the Halt icon on the toolbar, you selected Halt! from the Debug menu, or you pressed the ESC key.

Action None required; this is normal debugger behavior.

W

Window not found

Description The parameter supplied for the WIN command is not a valid window name.

Action Reenter the WIN command. Here are the valid window names; the bold letters show the smallest acceptable abbreviations:

Calls	**CP**U	**Co**mmand
Disassembly	**M**emory	**P**rofile
Watch		

Write not allowed for port

Description You attempted to connect a file for output operation to an address that is not configured for write.

Action Either change the software to write a port that is configured for write, or change the attributes of the port.

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.

C.4 Additional Instructions for Hardware Errors

If you continue to receive the messages that send you to this section, this indicates persistent hardware problems.

- ❏ If a bus fault occurs, the emulator may not be able to access memory.
- ❏ The ’C27xx must be reset before you can use the emulator. Most target systems reset the ’C27xx at power-up; your target system may not be doing this.

Appendix D

Glossary

A

active window: The window that is currently selected for moving, sizing, editing, closing, or some other function.

aggregate type: A C data type, such as a structure or array, in which a variable is composed of multiple variables, called members.

aliasing: A method of customizing debugger commands; aliasing provides a shorthand method for entering often-used command strings.

ANSI C: A version of the C programming language that conforms to the C standards defined by the *American National Standards Institute.*

assembly mode: A debugging mode that shows assembly language code in the Disassembly window and does not show the File window, no matter what type of code is currently running.

autoexec.bat: A batch file that contains DOS commands for initializing your PC.

auto mode: A context-sensitive debugging mode that automatically switches between showing assembly language code in the Disassembly window and C code in the File window, depending on what type of code is currently running.

B

batch file: One of two 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.

big endian: An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *little endian*

breakpoint: A point within your program where execution will halt because of a previous request from you.

break event: An event that causes the processor to halt.

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.

cl27: A shell utility that invokes the 'C27xx compiler, assembler, and linker to create an executable object file version of your program.

click: To press and release a mouse button without moving the mouse.

code-display windows: Windows that show code, text files, or code-specific information. This category includes the Disassembly, File, and Calls windows.

command line: The portion of the Command window where you can enter commands.

Command window: A window that provides an area for you to enter commands and for the debugger to echo command entry, show command output, and list progress or error messages.

common object file format (COFF): A binary object file format that promotes modular programming by supporting the concept of *sections*. All COFF sections are independently relocatable in memory space; you can place any section into any allocated block of target memory.

CPU window: A window that displays the contents of 'C27xx 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 'C27xx programs running on a 'C27xx emulator or simulator.

decode 1: The pipeline phase in which 32-bit and 16-bit instructions are differentiated and aligned to even or odd addresses, and in which it is determined whether or not an instruction is legal.

decode 2: The pipeline phase in which decode operations are completed and address generation is performed.

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.

E

EISA: *Extended Industry Standard Architecture.* A standard for PC buses.

emulator: A debugging tool that is external to the target system and provides direct control over the ’C27xx processor that is on the target system.

emurst: A utility that resets the emulator.

environment variable: A special system symbol that the debugger uses for finding directories or obtaining debugger options.

execute: The pipeline phase in which arithmetic, shift, and logic operations are performed.

F

fetch 1: The pipeline phase in which an instruction address is sent out.

fetch 2: The pipeline phase in which one or two instructions are read.

File window: A window that displays the contents of the current C code. The File window is intended primarily for displaying C code but can be used to display any text file.

float (a window): To cause a debugger window to sit on top of the debugger application window outside the edges of the debugger application window. A floating window always appears active.

I

init.cmd: A batch file that contains debugger-initialization commands. If this file is not present when you first invoke the debugger, then all memory is invalid.

I/O switches: Hardware switches on the emulator that identify the PC I/O memory space used for emulator-debugger or EVM-debugger communications.

ISA: *Industry Standard Architecture*. A subset of the EISA standard.

L

little endian: An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*

M

memory map: A map of memory space that tells the debugger which areas of memory can and cannot be accessed.

Memory window: A window that displays the contents of memory.

menu bar: A row of pulldown menu selections found at the top of the debugger display.

mixed mode: A debugging mode that simultaneously shows both assembly language code in the Disassembly window and C code in the File window.

O

open-collector output: An output circuit that actively drives both high and low logic levels.

P

PC: Personal computer or program counter, depending on the context and where it is used in this book: 1) In installation instructions or information relating to hardware and boards, *PC* means *personal computer*. 2) In general debugger and program-related information, *PC* means *program counter*, which is the register that identifies the current statement in your program.

point: To move the mouse cursor until it overlays the desired object on the screen.

port address: The PC I/O memory space that the debugger uses for communicating with the emulator. The port address is selected via switches on the emulator board and communicated to the debugger with the –p debugger option.

pulldown menu: A command menu that is accessed by name or with the mouse from the menu bar at the top of the debugger display.

R

read 1: The pipeline phase in which data is addressed.

read 2: The pipeline phase in which data is read from the read 1 address.

ripple-carry output signal: An output signal from a counter indicating that the counter has reached its maximum value.

S

scalar type: A C type in which the variable is a single variable, not composed of other variables.

scroll bar: A bar on the right side or bottom of a window that allows you to adjust the contents of the window to display hidden information.

scroll bar handle: The rectangular box in the center of the right scroll bar in the Disassembly or Memory window that marks the center of disassembled code or memory contents.

scrolling: A method of moving the contents of a window up, down, left, or right to view contents that were not originally shown.

section: A relocatable block of code or data that 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 'C27xx and lets you execute and debug applications programs by using the C source debugger.

single-step: A form of program execution that allows you to see the effects of each statement. The program is executed statement by statement; the debugger pauses after each statement to update the data-display windows.

status bar: An area at the bottom of the debugger application window that displays context-sensitive help and the status of the processor.

symbol table: A file that contains the names of all variables and functions in your program.

T

target system: A ’C27xx board that works with the emulator; the emulator doesn’t contain a ’C27xx device, so it must use a ’C27xx target board. Usually, the target system is a board that you have designed; you use the emulator and debugger to help you debug your design.

totem-pole output: An output circuit that actively drives both high and low logic levels.

W

Watch window: A window that displays the values of selected expressions, symbols, addresses, and registers.

window: A defined rectangular area of space on the display.

write: The pipeline phase in which if a transferred value or result is to be written to memory, the write occurs.

Index

B

C

D

E

F

G

H

I

L

M

N

O

P

S

T

U

V

W

X

Z