# TMS470 High-End Timer (HET) Simulator User's Guide

## Preliminary

Literature Number: SPNU181
June 1998

## 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*

The TMS470 high-end timer (HET) module provides sophisticated timing functions for complex, real-time applications, such as automobile engine management or power-train management. These applications require measurement of information from multiple sensors and drive actuators with complex timing.

This book tells you how to use the HET assembly source debugger with the simulator to refine and correct your code.

Before you use this book, you should use the appropriate installation instructions to install the assembly source debugger.

## *How to Use This Manual*

This book helps you learn to use the Texas Instruments standard programmer's interface for debugging. This book is divided into three parts:

- **Part I: Hands-On Information** is presented first so that you can start using your debugger the same day you receive it.

  Chapter 1 lists the key features of the debugger, describes additional HET software tools, tells you how to prepare an HET program for debugging, and provides instructions and options for invoking the debugger.

- **Part II: Debugger Description** contains detailed information about using the debugger.

  - Chapter 2 describes all of the debugger's windows and tells you how to move and size them.
  - Chapter 3 describes everything you need to know about entering commands.
  - Chapter 4 tells you how to define a memory map.
  - Chapter 5 contains instructions for loading, displaying, and running code.
  - Chapter 6 describes how to manage data.

- Chapter 7 describes the use of software breakpoints.
- Chapter 8 tells you how to customize the debugger display.

❑ **Part III: Reference Material** provides supplementary information.

- Chapter 9 gives a summary of all the tasks introduced in Parts I and II. This includes a functional summary of the debugger commands, a summary of the debugger commands in alphabetical order, and a topical summary of function key actions.
- Chapter 10 provides information about C expressions. The debugger commands are powerful because they accept C expressions as parameters; however, you can also use the debugger to debug assembly language programs. The information about C expressions aids assembly language programmers who are unfamiliar with C.
- Part III also includes a description of what the debugger does during invocation, an alphabetical listing of debugger messages with corrective actions, and a glossary.

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

❑ If you have used TI development tools or other debuggers before, you may want to:

- Read the introductory material in Chapter 1.
- Read the alphabetical command reference in Chapter 9.

❑ If this is the first time that you have used a debugger or similar tool, you may want to:

- Read the introductory material in Chapter 1.
- Read all of the chapters in Part II.

## Notational Conventions

This document uses the following conventions.

- The TMS470 high-end timer is abbreviated as the HET.
- The TMS470 high-end timer is referred to as the HET.
- The debugger has a very flexible command-entry system; there are usually a variety of ways to perform any specific action. For example, you may be able to perform the same action by typing in a command, using the mouse, or using function keys. This document uses three symbols to identify the methods that you can use to perform an action:

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

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

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

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

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

| Command | Result Displayed in the COMMAND Window |
|---|---|
| **? A** | 0xffff |
| **? HETADDR** | 0x0100 |
| **? *0x0100** | 0100h |

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

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

  **load** *object filename*

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

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

  **run** [*expression*]

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

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

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

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

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

## Related Documentation From Texas Instruments

The following books describe the TMS470 devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

***TMS470R1x User's Guide*** (literature number SPNU134) describes the TMS470R1x RISC microcontroller, its architecture (including registers), the ICEBreaker module, interfaces (memory, coprocessor, and debugger), 16-bit and 32-bit instruction sets, and electrical specifications.

***TMS470R1x Assembly Language Tools User's Guide*** (literature number SPNU118) describes the assembly language tools (the assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the TMS470R1x devices.

***TMS470R1x Optimizing C Compiler User's Guide*** (literature number SPNU119) describes the TMS470R1x C compiler. This C compiler accepts ANSI standard C source code and produces assembly language source code for the TMS470R1x devices.

***TMS470R1x Optimizing C/C++ Compiler User's Guide*** (literature number SPNU151) describes the TMS470R1x C/C++ compiler. This C/C++ compiler accepts ANSI standard C/C++ source code and produces assembly language source code for the TMS470R1x devices.

***TMS470R1x C Source Debugger User's Guide*** (literature number SPNU124) describes the TMS470R1x emulator and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

***TMS470R1x Simulator Getting Started Guide*** (literature number SPNU122) describes how to install the TMS470R1x simulator and the C source debugger for the '470 on Windows™ 3.1, Windows™ 95, Windows™ NT, HP-UX™, SunOS™, and Solaris™ systems.

***TMS470 High-End Timer (HET) Assembly Language Tools User's Guide*** (literature number SPNU182) describes the assembler, assembler directives, and macro language for the TMS470 high-end timer.

## Trademarks

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

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

Windows and Windows NT are registered trademarks of Microsoft Corporation.

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

## *If You Need Assistance . . .*

### ❑ World-Wide Web Sites

| | |
|---|---|
| TI Online | http://www.ti.com |
| Semiconductor Product Information Center (PIC) | http://www.ti.com/sc/docs/pic/home.htm |
| Microcontroller Home Page | http://www.ti.com/sc/micro |

### ❑ 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 | | |
| Microcontroller Hotline | (281) 274-2370 | Fax: (281) 274-4203 | Email: micro@ti.com |
| Microcontroller Modem BBS | (281) 274-3700 8-N-1 | | |

### ❑ 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 |

### ❑ 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 |

### ❑ 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: micro@ti.com

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

# Contents

## *Part I: Hands-On Information*



## *Part II: Debugger Description*

## *Part III: Reference Material*

# Figures



# Tables



# Examples

*Part I*

# *Hands-On Information*

*Part II*

# *Debugger Description*

*Part III*

# *Reference Material*

Chapter 1

# Overview of a Code Development and Debugging System



The TMS470 high-end timer (HET) assembly source debugger is an advanced programmer's interface that helps you to develop, test, and refine HET assembly language programs. The debugger is the interface to the HET simulator.

This chapter gives an overview of the key features of the HET debugger, describes the HET code development environment, and provides instructions and options for invoking the debugger.

**Topic** **Page**

## 1.1 Description of the HET Assembly Source Debugger

The HET assembly source debugging interface improves productivity by allowing you to debug a program written in HET assembly language.



### *Key features of the debugger*

- ❏ **On-screen editing.** Change any data value displayed in any window—just point the mouse, click, and type.
- ❏ **Automatic update.** The debugger automatically updates information on the screen, highlighting changed values.
- ❏ **Powerful command set.** Unlike many other debugging systems, this debugger does not force you to learn a large, intricate command set. The HET assembly source debugger supports a small, but powerful, command set that makes full use of C expressions. One debugger command performs actions that would take several commands in another system.
- ❏ **Flexible command entry.** There are various ways to enter commands. You can type commands or use a mouse, function keys, or the pulldown menus; choose the method that you like best. Do you want to reenter a command? There is no need to retype it—simply use the command history, which is an internal list that the debugger keeps of the commands that you enter.


- ❏ **Create your own debugger.** The debugger display is completely configurable, allowing you to create the interface that is best suited for your use.
  - ■ If you are using a color display, you can change the colors of any area on the screen.
  - ■ You can change the physical appearance of display features, such as window borders.
  - ■ You can interactively set the size and position of windows in the display.

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

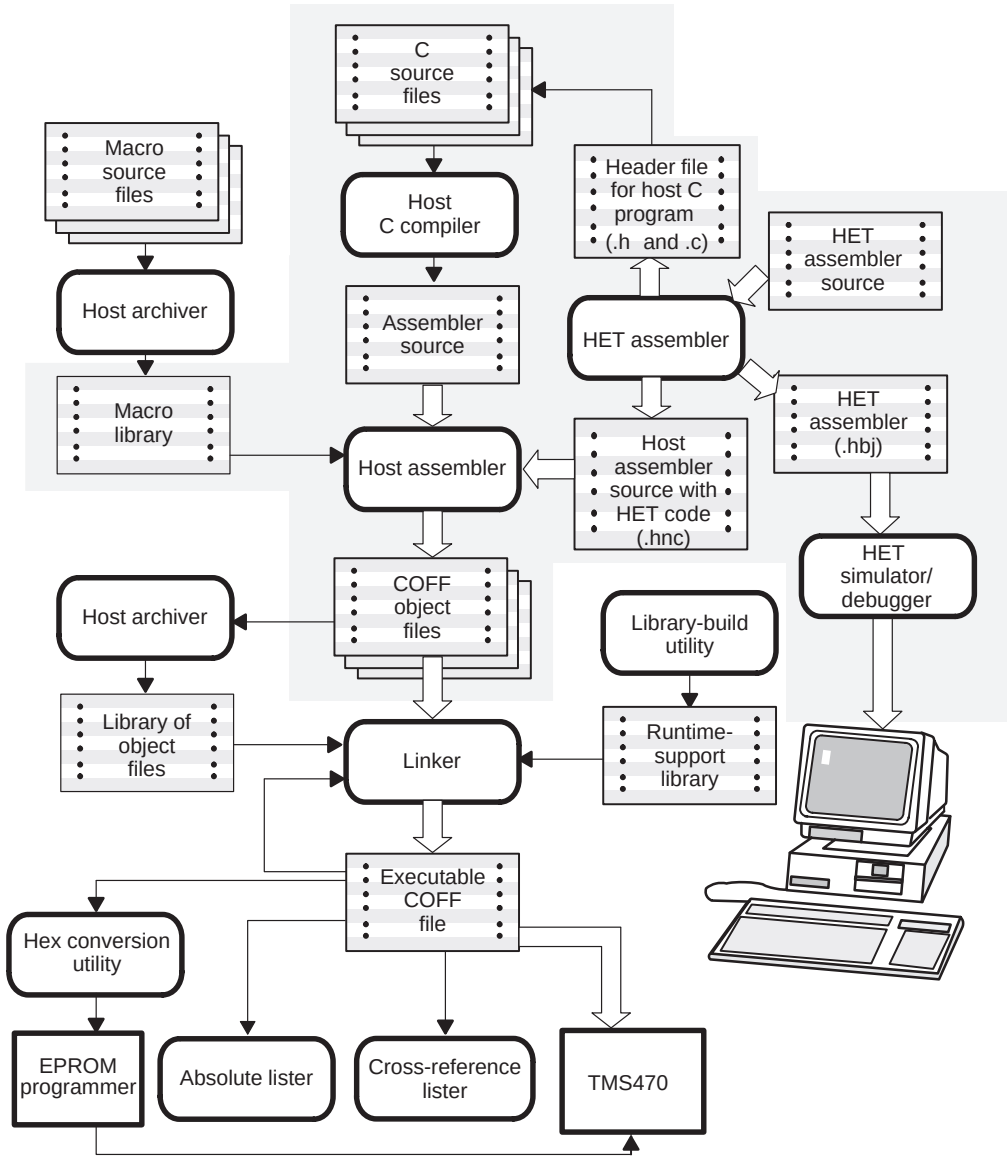
- **All the standard features you expect in a world-class debugger.** The debugger provides you with complete control over program execution with features like conditional execution and single-stepping. You can set or clear a breakpoint with a click of the mouse or by typing commands. You can define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

## 1.2 Developing Code for the HET

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



*Figure 1–1. HET Software Development Flow*



The HET simulator accepts only common object file format (COFF). 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 1–1.

HET assembler

The **HET assembler** translates HET assembly language source files into machine language object files. The HET assembler can generate these files:

- COFF object file (*.hbj*) for the HET simulator
- 470 assembler directives file (*.hnc*) for the host assembler
- C language header file (*.h*) and source file (*.c*) for the host C compiler

HET simulator

The main purpose of the development process is to produce a module that can be executed in an **HET target system**. You can use the simulator to simulate the operation of the HET target system and the HET debugger to refine and correct your code.

HET debugger

The **HET debugger** is a programmer's interface that helps you to develop, test, and refine HET assembly language programs. You can use the debugger as an interface for the software simulator.

Host C compiler

The **host C compiler** accepts C source code and the C header file and produces HET 32-bit assembly language source. See the appropriate C language tools user's guide for your device for an explanation of how to use the compiler.

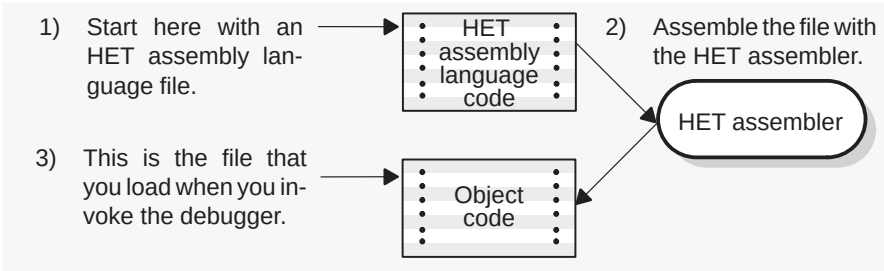
Host assembler

The **host assembler** translates assembly language source files into machine language COFF object files. See the appropriate assembly language tools user's guide for your device for an explanation of how to use the assembler.

## 1.3 Preparing Your Program for Debugging

Figure 1–2 illustrates the steps to prepare a program for debugging.

*Figure 1–2. Steps to Prepare a Program for Debugging*




Assemble the assembly language source file. This produces an object file that you can load into the debugger.

For more information about assembling code, see the *TMS470 High-End Timer (HET) Assembly Language Tools User's Guide.*

## 1.4 Verifying the Debugger Installation

When you invoke the debugger for the first time, enter the appropriate command at the system prompt to ensure that you have correctly installed the simulator and debugger software:

**simhet sample** ↵ For UNIX™ workstations

**simhet c:\hethll\sim\sample** ↵ For PC™ workstations

You should see a display similar to this one:


If you see a display similar to this one, you have correctly installed your simulator and debugger.

If you do not see a similar display, then your debugger or simulator may not be installed properly. Go back through the installation instructions on the CD-ROM insert and be sure that you have followed each step correctly. Also, if you are running the debugger on a PC and you see a display but the lines of code say *Invalid address* or the fields in the MEMORY window are shown in red, the debugger may not be able to find the init.cmd file which contains the debugger initialization commands . Check for that file in the directories specified by the D_DIR environment variable and in the current directory; ensure that it is in one place or the other. Reenter the appropriate command above. (For information about setting the D_DIR environment variable, see the CD_ROM insert.)

## 1.5 Invoking the Debugger

Enter the following command on the command line to invoke the standalone debugger (if you are invoking the debugger for the first time, see section 1.4, *Verifying the Debugger Installation*).

**simhet** [*filename*] [*–options*]

| | |
|---|---|
| **simhet** | is the command that invokes the debugger. |
| *filename* | is an optional parameter that names an object file that the debugger loads into memory during invocation. The debugger looks for the file in the current directory; if the file is not in the current directory, you must supply the entire pathname. If you do not supply an extension for the filename, the debugger assumes that the extension is *.hbj*. |
| *–options* | supply the debugger with additional information. |

Table 1–1 lists the debugger options that you can use when invoking a debugger, and the subsections that follow the table describe these options. You can also specify filename and option information with the D_OPTIONS environment variable (see the information about setting the D_OPTIONS environment variable on the CD-ROM insert).

*Table 1–1. Summary of Debugger Options*

| Option | Brief Description |
|---|---|
| –b | Select a preset screen size (80 characters by 43 lines). |
| –bb | Select a slightly larger preset screen size (80 characters by 50 lines). |
| –d *machinename* | Display the debugger on a different machine (X Window System™ only). |
| –i *pathname* | Identify additional directories. |
| –min | Select the minimal debugging mode. |
| –s | Load the symbol table only. |
| –t *filename* | Identify a new initialization file. |
| –v | Load without the symbol table. |
| –x | Ignore D_OPTIONS. |

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

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

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

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

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

If you are using the X Window System, you can use the -d option to display the debugger on a different machine than the one the program is running on. For example, if you are running a debugger on a machine called opie and you want the debugger display to appear on a machine called barney, use the following command to invoke the debugger:

```
simhet -d barney:0 ↵
```

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

### Identifying additional directories (–i option)

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

**simhet –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 the D_SRC environment variable on the CD-ROM insert). If you name directories with both –i and D_SRC, the debugger first searches through directories named with –i. The debugger can track a cumulative total of 20 paths (including paths specified with –i, D_SRC, and the debugger USE command).

### Selecting the minimal debugging mode (–min option)

By default, the MEMORY, COMMAND, DISASSEMBLY, and CPU windows are displayed. You may also display a WATCH window.

The debugger has a *minimal* debugging mode that displays the COMMAND and WATCH windows only. The WATCH window is displayed only if you cause it to display (by entering the WA command).

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

```
simhet –min ...
```

For more information about the windows in the debugger interface, see section 2.2, *Descriptions of the Different Kinds of Windows and Their Contents*, page 2-4.

### Loading the symbol table only (–s option)

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

### Identifying a new initialization file (–t option)

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

**–t** *filename*

### Loading without the symbol table (–v option)

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

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

### Ignoring D_OPTIONS (–x option)

The –x option tells the debugger to ignore any information supplied with the D_OPTIONS environment variable.

## 1.6 Using the Debugger With the X Window System

If you use the X Window System to run the HET debugger, you need to know about the keyboard's special keys, the debugger font, and using the debugger on a monochrome monitor.

### *Using the special keys on the keyboard*

The debugger uses some special keys that you can map differently than your particular keyboard. Some keyboards, such as the Sun Type 5 keyboard, have these special symbols on separate keys. Other keyboards, such as the Sun Type 4 keyboard, do not have the special keys, but the functions are available.

The special keys that the debugger uses are shown in the following table with their corresponding keysym. A *keysym* is a label that interprets a keystroke; it allows you to modify the action of a key on the keyboard.

| Debugger Key Needed | Keysym for That Function |
|---|---|
| F1 to F10 | F1 to F10 |
| PAGE UP | Prior |
| PAGE DOWN | Next |
| HOME | Home |
| END | End |
| INSERT | Insert |
| → | Right |
| ← | Left |
| ↑ | Up |
| ↓ | Down |

Use the X utility xev to check the keysyms associated with your keyboard. If you need to change the keysym definitions, use the xmodmap utility. For example, you could create a file that contains the following commands and use that file with xmodmap to map a Sun Type 4 keyboard to the keys listed above:

```
          key code                keysym
keysym R13    = End
keysym Down   = Down
keysym F35    = Next
keysym Left   = Left
keysym Right  = Right
keysym F27    = Home
keysym Up     = Up
keysym F29    = Prior
keysym Insert = Insert
```

See your X Window System documentation for more information about using xev and xmodmap.

### Changing the debugger font

You can change the font of the debugger screen by using the xrdb utility and modifying the .Xdefaults file in your root directory. For example, to change the HET debugger font to courier, add the following line to the .Xdefaults file:

```
hethll*font:courier
```

For more information about using xrdb to change the font, see your X Window System documentation.

### Color mappings on monochrome screens

Although a color monitor is recommended, you can use a monochrome monitor. The following table shows the color mappings for monochrome screens:

| Color | Appearance on Monochrome Screen |
|---|---|
| black | black |
| blue | black |
| green | white |
| cyan | white |
| red | black |
| magenta | black |
| yellow | white |
| white | white |

## 1.7 Exiting the Debugger

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

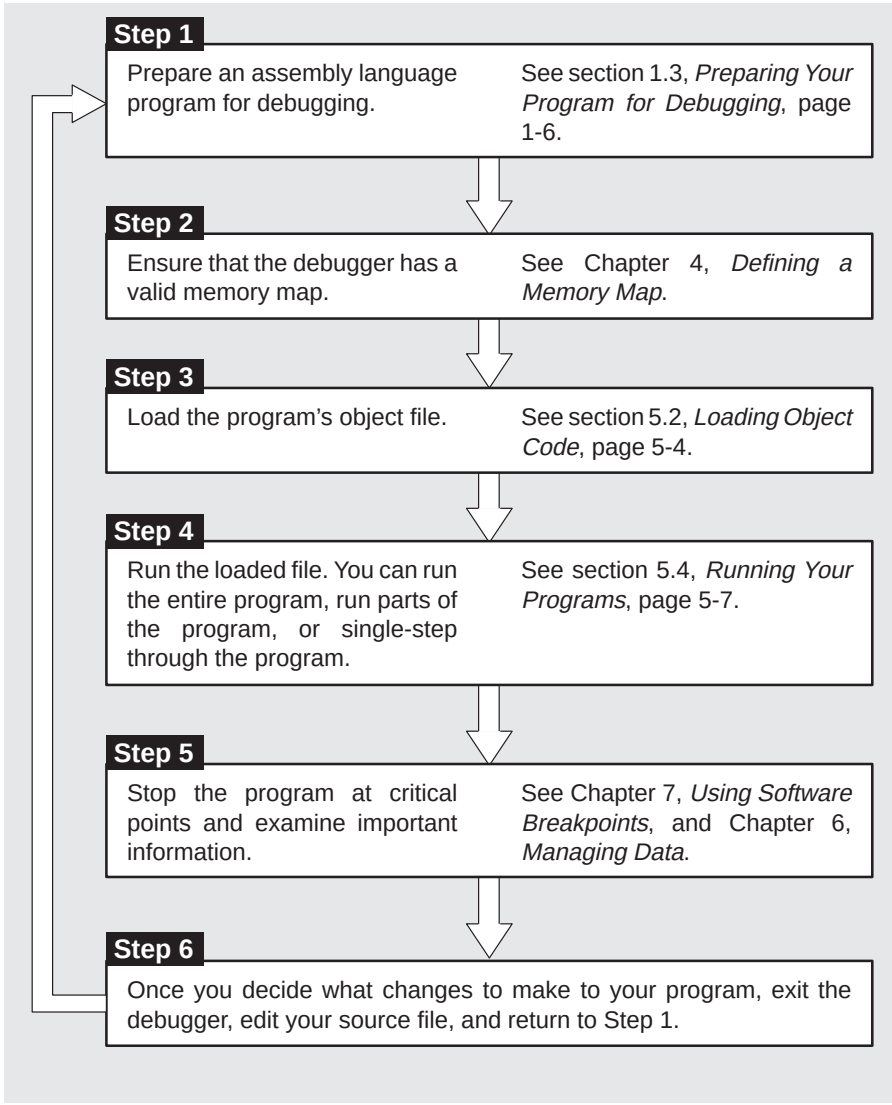
```
quit ↵
```

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

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

## 1.8 Debugging HET Programs

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



*Part I*

# *Hands-On Information*

*Part II*

# *Debugger Description*

*Part III*

# *Reference Material*

*Part II*

Chapter 2

# The Debugger Display

The HET assembly source debugger has a window-oriented display. This chapter shows what windows look like and describes the basic types of windows that you will use.

Part II

**Topic** **Page**

## 2.1 The Debugger Modes

The HET debugger has two modes for displaying assembly language programs: assembly mode and minimal mode. This section describes these two modes and explains how to select a debugging mode. Chapter 9, *Summary of Commands and Special Keys*, summarizes the debugger commands used in this chapter.

### *Assembly mode*

When you invoke the debugger, you see a display similar to the one shown in Figure 2–1. This is the *assembly mode* of the debugger. Windows that are displayed in assembly mode automatically are the MEMORY window, the DISASSEMBLY window, the CPU register window, and the COMMAND window. If you choose, you can open a WATCH window by using the WA command.



*Figure 2–1. Typical Assembly Display (for Assembly Mode)*

### Minimal mode

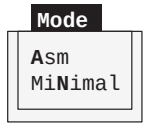
A second debugger mode, *minimal mode*, allows you to query the target system without displaying any additional information. The only window that is displayed automatically in minimal mode is the COMMAND window (shown in Figure 2–1). You can display the contents of CPU registers, memory addresses, or symbols within the COMMAND window by using the WA, ?, and EVAL commands. You can use any of the standard debugger commands in the COMMAND window. If you use the ASM command, the debugging mode changes to the assembly mode. To return to minimal mode, use the MINIMAL command.

### Selecting a debugging mode

Unless you use the –min command-line option (which selects minimal mode and is discussed on page 1-10), when you first invoke the debugger, it automatically comes up in assembly mode. You can then switch to minimal mode. There are several ways to do this.



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



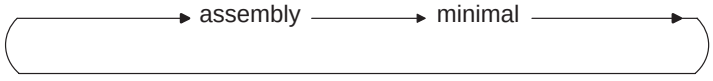

1) Point to the menu name.

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

3) Release the mouse button.

For more information about the pulldown menus, see Section 3.2, *Using the Menu Bar and the Pulldown Menus*, on page 3-7.


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




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

**asm** Changes from the current mode to assembly mode

**minimal** Changes from the current mode to minimal mode

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

## 2.2 Descriptions of the Different Kinds of Windows and Their Contents

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

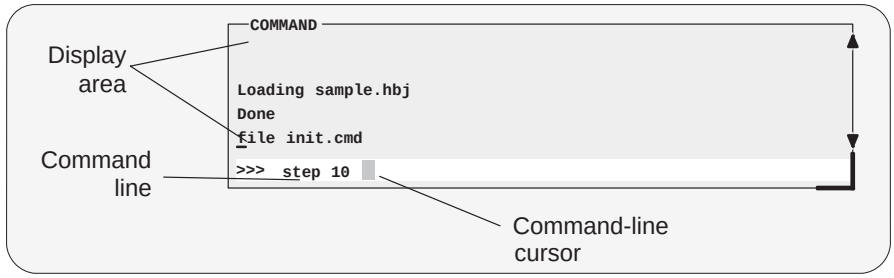
Every window is identified by a name in its upper left corner. Each type of window serves a specific purpose and has unique characteristics. There are five different windows, divided into these general categories:

- The COMMAND window provides an area for typing in commands and for displaying various types of information, such as progress messages, error messages, or command output.
- The DISASSEMBLY window displays the disassembly (assembly language version) of memory contents.
- Data-display windows are used for observing and modifying various types of data. There are three data-display windows:
  - A MEMORY window displays the contents of a range of memory. You can display multiple MEMORY windows at one time.
  - The CPU window displays the contents of HET registers.
  - A WATCH window displays selected data such as variables, specific registers, or memory locations. You can display multiple WATCH windows simultaneously.

You can move or resize any of these windows, or edit any value in a data-display window. Before you perform any of these actions, however, you must select the window you want to move, resize, or edit and make it the *active window*. For more information about making a window active, see section 2.4, *The Active Window*.

The remainder of this section describes the individual windows.

### *COMMAND window*




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

*Editable?* Command line is editable; command output shown in the display area is not.

*Created* Automatically

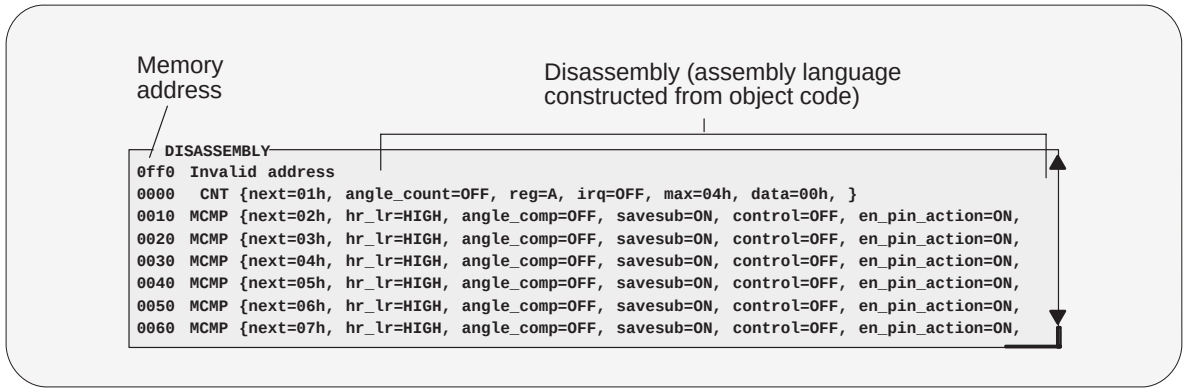
*Affected by*
- ❑ All commands entered on the command line
- ❑ All commands that display output in the display area
- ❑ Any condition or input that creates an error

The COMMAND window has two parts:

- ❑ **Command line**. This is where you enter commands. When you want to enter a command, just type—no matter which window is active. The debugger retains a list of the last 50 commands that you entered. You can select and reenter commands from the list without retyping them. (For more information, see *Using the command history,* page 3-5.)
- ❑ **Display area**. This area of the COMMAND window echoes the command that you entered, shows any output from the command, and displays debugger messages.

For more information about the COMMAND window and entering commands, see Chapter 3, *Entering and Using Commands*.

## DISASSEMBLY window




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

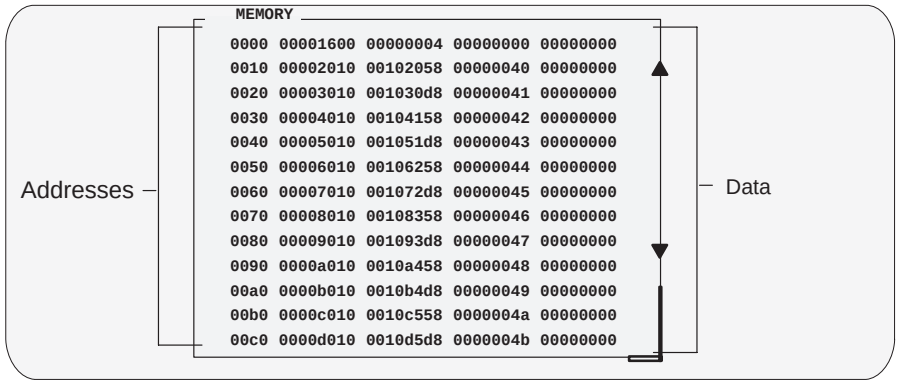
*Editable?* No—pressing the edit key (F9) or the left mouse button sets a software breakpoint on an assembly language statement.

*Created* Automatically

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

Within the DISASSEMBLY window, the debugger highlights any statements with software breakpoints.

### *MEMORY window*




| | |
|---|---|
| *Purpose* | Displays the memory contents |
| *Editable?* | Yes—you can edit the data (but not the addresses). |
| *Created* | ❑ Automatically (the default MEMORY window only)<br>❑ With the MEM command and a unique *window name* |
| *Affected by* | MEM command |

A MEMORY window has two parts:

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

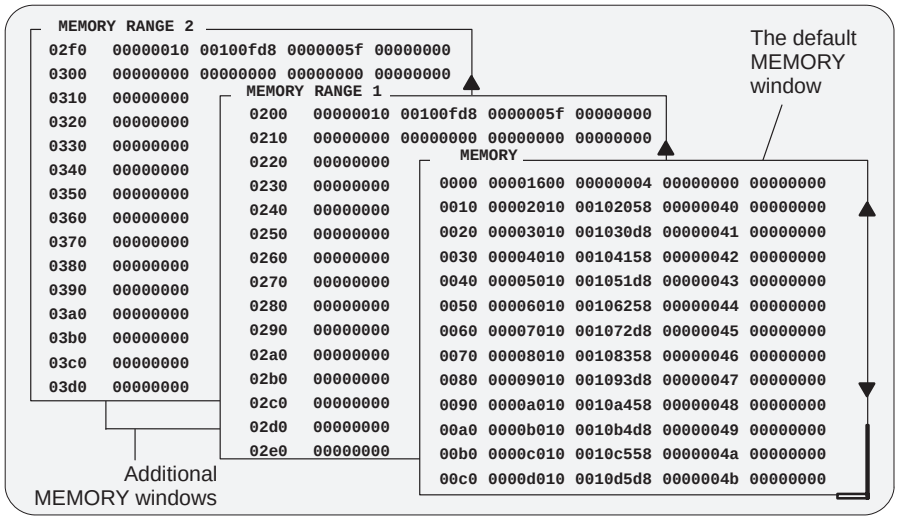
- **Data.** The remaining columns display the values at the listed addresses. The data is shown in hexadecimal format as 128-bit words. You can display more data by making the window wider and/or longer. For more information, see section 2.5, *Manipulating a Window*.

  By default, the MEMORY window above has four columns of data, and each new address is incremented by 16 (0x10). Although the window shows four columns of data, there is still only one column of addresses; the first value is at address 0x0000, the second at address 0x0004, etc.; the fith value (first value in the second row) is at address 0x0010, the next at address 0x0014, etc.

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

The debugger opens one MEMORY window by default. You can open any number of additional MEMORY windows to display different ranges of memory. See Figure 2–2.

*Figure 2–2. Default and Additional MEMORY Windows*




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

❑ **Opening an additional MEMORY window**

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

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

For example, if you want to open a new MEMORY window starting at address 0x100 named RANGE1, enter:

```
mem 0x0010,,RANGE1 ↵
```

This displays a new window, labeled MEMORY RANGE1, showing the contents of memory starting at the address 0x100.

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

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

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

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

- The *display format* parameter for the MEM command is optional. When used, the data is displayed in the selected format as shown in Table 6–1 on page 6-15.
- The *window name* parameter is optional if you are displaying a different memory range in the default MEMORY window. Use the *window name* parameter when you want to display a new memory range in one of the additional MEMORY windows.

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

❑ **Closing a MEMORY window**

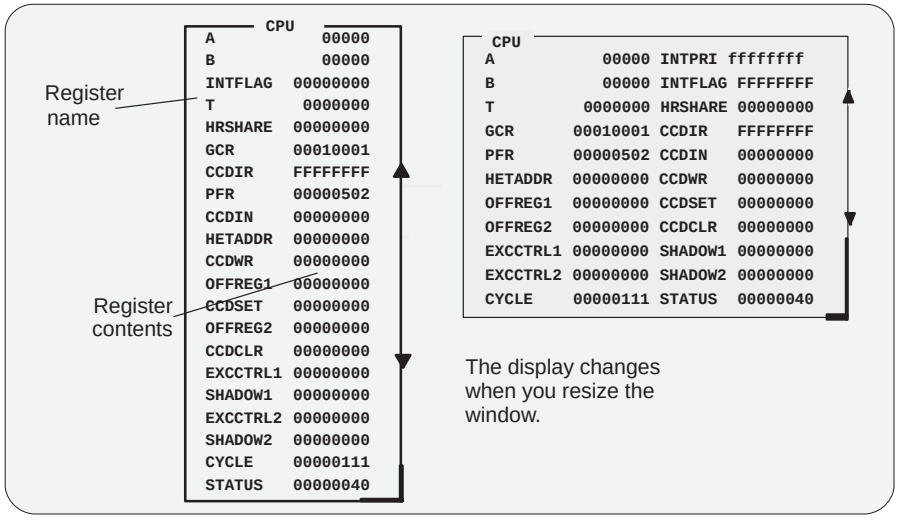
Closing a window is a two-step process:

1) Make the appropriate MEMORY window the active window (see Section 2.4 on page 2-15).
2) Press F4 .

❑ **Reopening a MEMORY window**

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

### CPU window




*Purpose* Shows the contents of the HET registers

*Editable?* Yes—you can edit the value of any displayed register.

*Created* Automatically

*Affected by* Data-management commands

As you execute programs, some values displayed in the CPU window change. The debugger highlights changed values. The CYCLE and STATUS registers are pseudoregisters defined by the simulator to offer more information and control to the user.

The CYCLE register displays the current count of CPU cycles that the simulator has at that point. This is the cycle count used when simulating the **pinc** commands. You can reset the cycle count using the reset command. (For more information, see section 4.7, *Simulating External Signals*, page 4-10.)

The STATUS register shows which flags/pins are set. The following table shows you the bit number in the status register, the symbol of the flag/pin at that bit location, and a description of the flag/pin.

| Bit | Symbol | Flag/Pin Name |
|---|---|---|
| 0 | Z | Z flag |
| 1 | X | X flag |
| 2 | NAF | New angle flag |
| 3 | ACF | Acceleration flag |
| 4 | DCF | Deceleration flag |
| 5 | GPF | Gap flag |
| 6 | MODE | Privileged mode flag |



Setting the MODE pin puts the HET in privileged mode, allowing you to write to registers that can only be written to while in privileged mode. You can set the MODE pin three different ways:

1) You can manually edit data in the CPU window. All register value displays are editable fields.

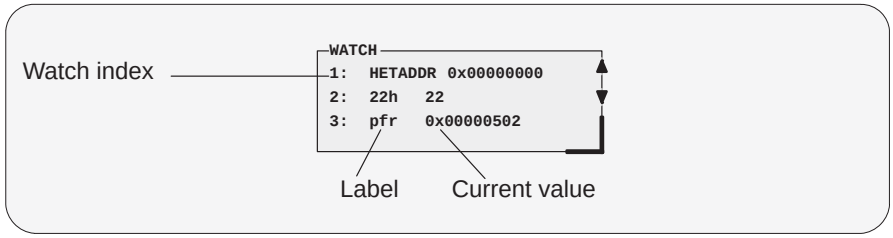
2) You can use the **pinc** MODE command. (For more information, see section 4.7, *Simulating External Signals*, page 4-10.)

3) From the command line, you can use the **?** command to set the register value, for example:

   ```
   ?STATUS |= 0x040
   ```

   This sets the MODE bit on.

### *WATCH window*




| | |
|---|---|
| *Purpose* | Displays the values of selected expressions |
| *Editable?* | Yes—you can edit the value of any expression whose value corresponds to a single storage location (in registers or memory). |
| *Created* | With the WA command |
| *Affected by* | WA, WD, and WR commands |

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

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

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

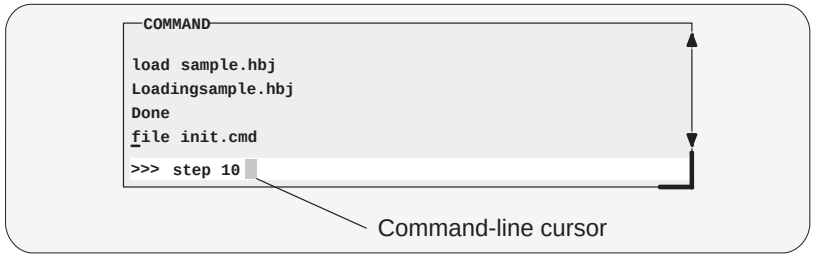
- ❏ WA adds *expression* to the WATCH window. (If there is no WATCH window, then WA also opens a WATCH window.)
- ❏ The *label* parameter is optional. When used, it provides a label for the watched entry. If you do not use a *label*, the debugger displays the *expression* in the label field.
- ❏ The *display format* parameter is optional. When used, the data is displayed in the selected format as shown in Table 6–1 on page 6-15.
- ❏ If you omit the *window name* parameter, the debugger displays the expression in the default WATCH window (labeled WATCH). You can open additional WATCH windows by using the *window name* parameter. When you open an additional WATCH window, the debugger appends the *window name* to the WATCH window label. You can create as many WATCH windows as you need.

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

## 2.3 Cursors

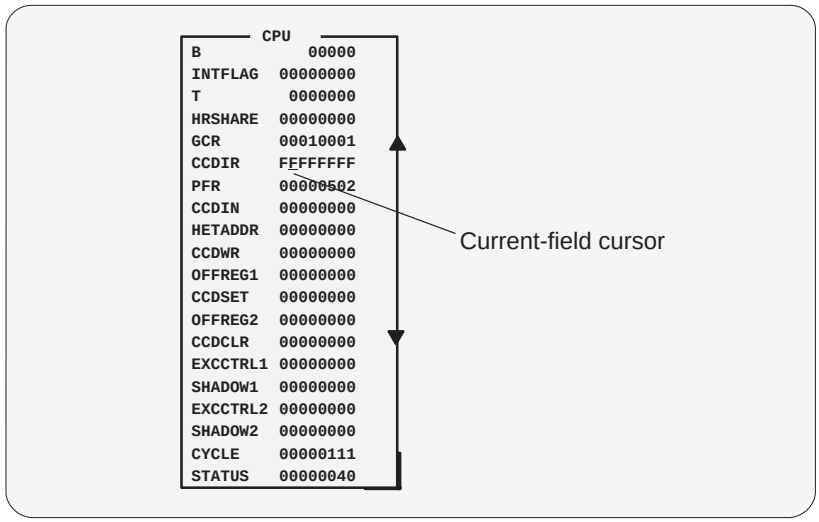
The debugger display has three types of cursors:

- The *command-line cursor* is a block-shaped cursor that identifies the current character position on the command line. When the COMMAND window is active (see section 2.4, *The Active Window*), arrow keys affect the position of this cursor.




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

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



## 2.4 The Active Window

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

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

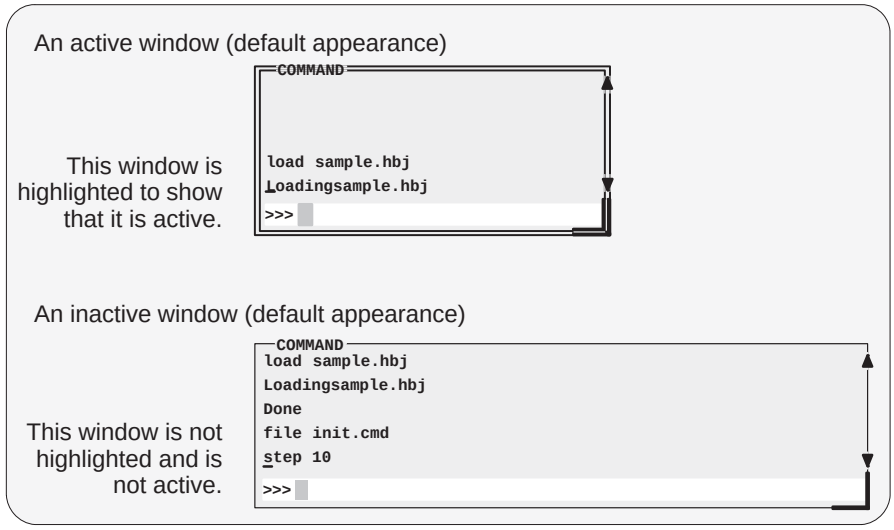
### *Identifying the active window*

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

You can alter the active window's border style and colors if you wish; Figure 2–3 illustrates the default appearance of an active window and an inactive window.



*Figure 2–3. Default Appearance of an Active and an Inactive Window*




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

### *Selecting the active window*

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

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

2) Click the left mouse button.

If you point within the window, you might also select the current field. For example:

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

  *Press ESC to deselect the current field.*

- If you point inside the DISASSEMBLY window, you will set a breakpoint on the statement you are pointing to.

  *Press the button again to clear the breakpoint.*


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

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

**win** *WINDOW NAME*

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

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

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

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

If you supply an ambiguous name, 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.

## 2.5 Manipulating a Window

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

**Note:**

You can resize or move any window, but first the window must be *active*. For information about selecting the active window, see Section 2.4 on page 2-15.

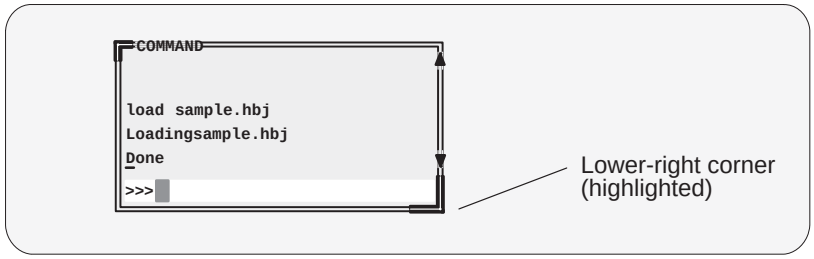
### *Resizing a window*

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

There are two ways to resize a window:

- ❏ Using the mouse
- ❏ Using the SIZE command

1) Point to the lower-right corner of the window. This corner is highlighted:




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

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

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

**size** [*width*, *length* ]

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

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

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

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

If a window is in the middle of the display, you cannot size it to the maximum height and width—you can size it only to the right and bottom screen borders. The easiest way to make a window as large as possible is to zoom it, as described on page 2-20.

For example, if you want to use commands to make the COMMAND window 60 characters wide by 15 lines long, you could enter:

```
win  COMMAND ↵
size 60, 15 ↵
```

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

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

When you finish using the arrow keys, you must press ESC or ↵.

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

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

## *Zooming a window*

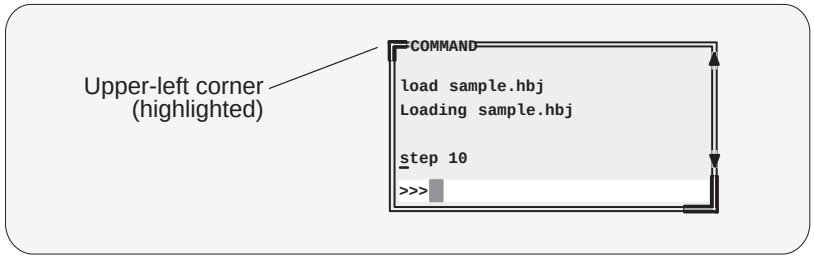
Another way to resize the active window is to zoom it. Zooming a window makes it as large as possible, so that it fills the entire display (except for the menu bar) and hides all of the other windows. Unlike the SIZE command, zooming is not affected by the window's position in the display.

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

There are two ways to zoom or unzoom a window:

- ❏ Using the mouse
- ❏ Using the ZOOM command

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




2) Click the left mouse button.

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

**zoom**

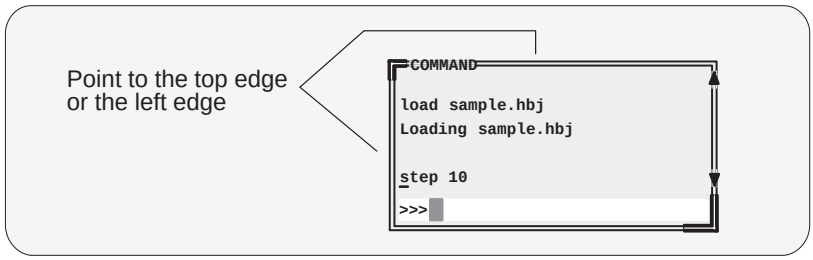
### *Moving a window*

The windows in the debugger display do not have fixed positions—you can move them around.

There are two ways to move a window:

- ❏ Using the mouse
- ❏ Using the MOVE command

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




2) Press the left mouse button but do not release it; now move the mouse in any direction.

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

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

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

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

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

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

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

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

**Note:**

If you choose, you can resize a window at the same time you move it. To do this, use the *width* and *length* parameters in the same way that they are used for the SIZE command (see page 2-19).



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

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

When you are finished using the arrow keys, you must press ESC or ↵.

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

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

## 2.6 Manipulating a Window's Contents

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

**Note:**

You can scroll and edit only the *active window*. For information, see Section 2.4 on page 2-15.

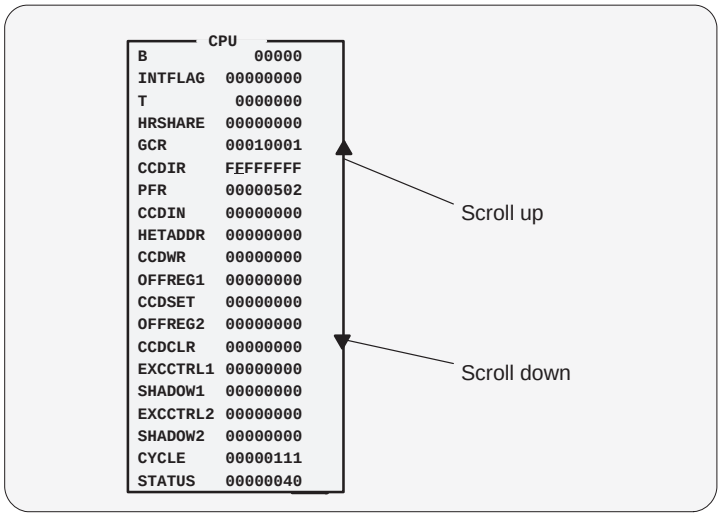
### *Scrolling through a window's contents*

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

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

- Use the mouse to scroll the contents of the window.
- Use function keys and arrow keys.

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



To scroll window contents up or down:

1) Point to the appropriate scroll arrow.

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

3) Release the mouse button when you are finished scrolling.

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


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

PAGE UP — The page-up key scrolls up through the contents of the active window, one window length at a time.

PAGE DOWN — The page-down key scrolls down through the contents of the active window, one window length at a time.

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

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

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

### *Editing the data displayed in windows*

You can edit the data displayed in the MEMORY, CPU, and WATCH windows by using an overwrite click-and-type method or by using commands that change the values. This is described in detail in section 6.3, *Basic Methods for Changing Data Values*, page 6-4.

**Note:**

In the DISASSEMBLY window, pressing F9 or the mouse button sets or clears a breakpoint on any line of code that you select. You cannot modify text in a DISASSEMBLY window.

## 2.7 Closing a Window

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

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

1) Make the appropriate window active.

2) Press F4.

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

**wr** [*window name*]

When you close a window, the debugger remembers the window's size and position. The next time you open the window, it has the same size and position. When you open a WATCH or MEMORY window, it occupies the same position as the last one of that type that you closed.

Chapter 3

# Entering and Using Commands

The debugger provides you with several methods for entering commands:

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

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



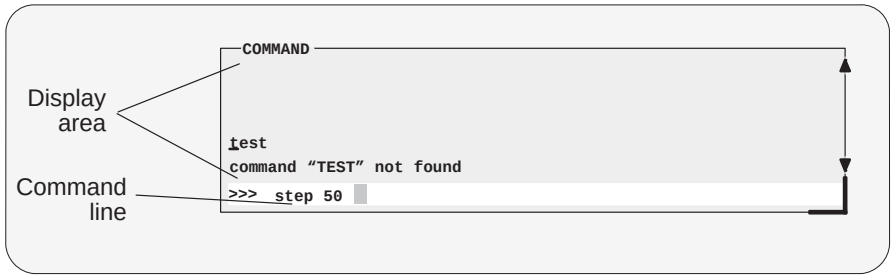
**Topic** **Page**

## 3.1 Entering Commands From the Command Line

The debugger supports a complete set of commands that help you to control and monitor program execution, customize the display, and perform other tasks. These commands are discussed in the various sections throughout this book, as they apply to the topic that is being discussed. Chapter 9, *Summary of Commands and Special Keys*, summarizes all of the debugger commands with an alphabetic reference.

Although there are a variety of methods for entering most of the commands, *all* of the commands can be entered by typing them on the command line in the COMMAND window. Figure 3–1 shows the COMMAND window.

*Figure 3–1. The COMMAND Window*




The COMMAND window serves two purposes:

- The *command line* portion of the window provides an area for entering commands. For example, the command line in Figure 3–1 shows that a STEP command was typed in (but ↵ has not yet been pressed).
- The *display area* provides the debugger with a space for echoing commands, displaying command output, or displaying errors and messages for you to read. The command output in Figure 3–1 shows the message that is displayed when you enter "test", which is an invalid command.

  If you enter a command through an alternative method (using the mouse, a pulldown menu, or function keys), the COMMAND window does not echo the entered command.

### *Typing in and entering commands*

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

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

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

Once you type a command, you can edit the text on the command line with these keystrokes:

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

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

**Notes:**

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

2) Typing a command does not make the COMMAND window the active window.

### *Sometimes you cannot type a command*

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

- ❏ When you press the ALT key, typing the first letter of the pulldown menu you wish to view causes the debugger to display that pulldown menu.
- ❏ When a pulldown menu is displayed, typing the first letter of your selection causes the debugger to execute that selection from the menu.
- ❏ When you press the CONTROL key, pressing H or L moves the command-line cursor backward or forward through the text on the command line.
- ❏ When you edit a field, typing enters a new value in the field.
- ❏ When you use the MOVE or SIZE command interactively, pressing keys affects the size or position of the active window. Before you can enter any more commands, you must press ESC to terminate the interactive moving or sizing.
- ❏ When you bring up a dialog box, typing enters a parameter value for the current field in the box. See Section 3.3 on page 3-11 for more information on dialog boxes.

### Using the command history

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

Use these keystrokes to move through the command history.

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

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



### Clearing the display area

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

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

**cls**

### Recording information from the display area

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

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

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

**dlog** *filename*

This command opens a log file called *filename* to record the information into.

❏ To end the recording session, enter:

**dlog close** ↵

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

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

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

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

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

❏ **Writing over an existing file.** Use the **w** parameter to open an existing file to write over the current contents of the file. This is the default action if you specify an existing filename without using either the **a** or **w** option; you lose the contents of an existing file if you do not use the append (a) option.

## 3.2 Using the Menu Bar and the Pulldown Menus

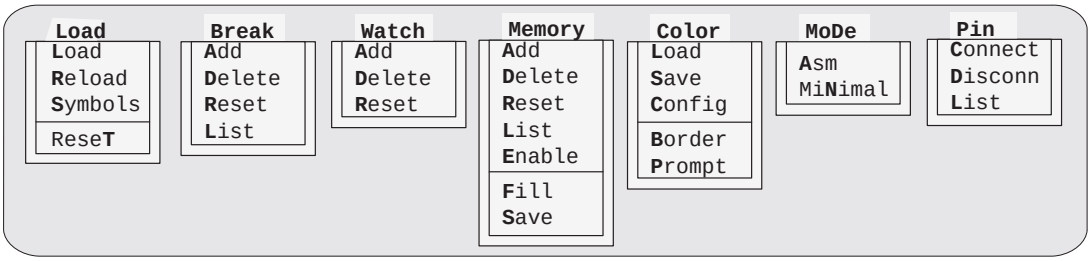
In both debugger modes, you see a menu bar at the top of the screen. The menu selections offer you an alternative method for entering many of the debugger commands. Figure 3–2 points out the menu bar in an assembly mode display. There are several ways to use the selections on the menu bar, depending on whether the selection has a pulldown menu or not.

*Figure 3–2. The Menu Bar in the Debugger Display*


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

The menu bar and associated pulldown menus occupy fixed positions on the display. Unlike windows, you cannot move, resize, or cover the menu bar or pulldown menus.

*Figure 3–3. All of the Pulldown Menus*



## *Using the pulldown menus*

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

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

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



**Mouse method 1**

1) Point the mouse cursor at the appropriate selection in the menu bar.
2) Press the left mouse button; do not release the button.
3) While pressing the mouse button, move the mouse downward until your selection is highlighted on the menu.
4) When your selection is highlighted, release the mouse button.

**Mouse method 2**

1) Point the cursor at the appropriate selection in the menu bar.
2) Click the left mouse button. This displays the menu until you are ready to make a selection.
3) Point the mouse cursor at your selection on the pulldown menu.
4) When your selection is highlighted, click the left mouse button.


**Keyboard method 1**

ALT 1) Press the ALT key; do not release it.

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

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

**Keyboard method 2**

ALT 1) Press the ALT key; do not release it.

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

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

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



## *Escaping from the pulldown menus*

Here is how to escape from a pulldown menu:

- ❑ If you display a menu and then decide that you do not want to make a selection from this menu, you can use one of these methods:
  - ■ Press ESC.
  - ■ Point the mouse outside of the menu; press and then release the left mouse button.
- ❑ If you pull down a menu and see that it is not the menu you want, you can use one of these methods:
  - ■ Point the mouse at another entry and press the left mouse button.
  - ■ Press the ← or → key to display an adjacent menu.

### *Using menu bar selections that do not have pulldown menus*

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

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

There are two ways to execute these choices.

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

2) Click the left mouse button.

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


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

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

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

For more information about the RUN, STEP, and NEXT commands, see section 5.4, *Running Your Programs*, page 5-7.

## 3.3 Using Dialog Boxes

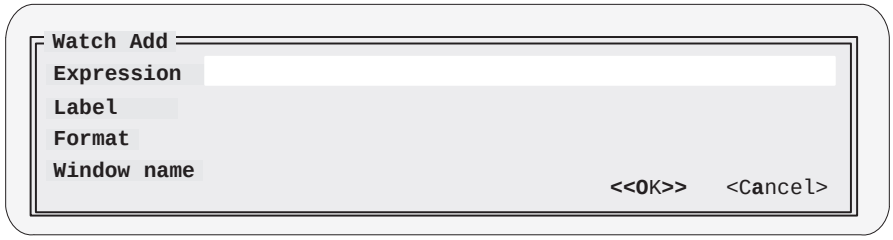
Many debugger commands have parameters. When you execute these commands from pulldown menus, you must have some way of providing parameter information. The debugger allows you to do this by displaying a *dialog box* that asks for this information.

### *Entering text in a dialog box*

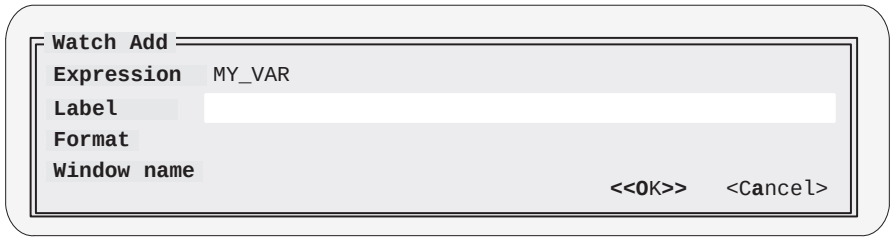
Entering text in a dialog box is much like entering commands on the command line. For example, the Add entry on the Watch menu is equivalent to entering the WA command. This command has four parameters:

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

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




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




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

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

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

- When you display a dialog box for the first time during a debugging session, the parameter fields are empty. Bringing up the same dialog box again, displays the last values that you entered. (This is similar to having a command history.) If you want to use the same value, just press TAB or ↓ to move to the next parameter.
- You can edit what you type (or values that remain from a previous entry) in the same way that you can edit text on the command line. See Section 3.1 for more information on editing text on the command line.

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

## 3.4 Entering Commands From a Batch File

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

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

The format for the TAKE command is:

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

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

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

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

- By default, the debugger echoes the commands in the display area of the COMMAND window and updates the display as it reads commands from the batch file.
  - If you do not use the *suppress echo flag* parameter, or if you use it but supply a nonzero value, the debugger behaves in the default manner.
  - If you want to suppress the echoing and updating, use the value 0 for the *suppress echo flag* parameter.

### Echoing strings in a batch file

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

**echo** *string*

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

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

```
echo Creating new memory map
```

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

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

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

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



### Controlling command execution in a batch file

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

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

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

The debugger includes a predefined constant ($$SIM$$) for use with the IF command. This constant evaluates to 0 (false) or 1 (true). The simulator is the corresponding tool for $$SIM$$.

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 10, *Basic Information About C Expressions*, for more information about expressions and expression analysis.)

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

**loop** *expression*
*debugger command*
*debugger command*
.
.
**endloop**

These looping commands evaluate using the same method as the run conditional command expression. (See Chapter 10, *Basic Information About C Expressions*, for more information about expressions and expression analysis.)

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

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

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

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

| | | |
|---|---|---|
| > | > = | < |
| < = | = = | ! = |
| && | \|\| | ! |

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

```
loop !0
step
? PCR
? A
endloop
```



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

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

## 3.5 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 enables you to define an alias name for the command(s) and then enter the alias name as if it were a debugger command.

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

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

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

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

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

  ```
  alias init,"load test.hbj;file init.cmd;step 10"
  ```

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

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

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

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

  You could enter:

  ```
  mfil 0x01,0x18,0x1122
  ```

  In this example, the first value (0x01) is substituted for the first FILL parameter and the MEM parameter (%1). The second and third values are substituted for the second and third FILL parameters (%2 and %3).

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

  For example, assume that the init and mfil aliases have been defined as shown on page 3-17. If you enter:

  **`alias`** ↵

  you will see:

```
  Alias       Command
----------------------------------------
INIT    -->  load test.hbj;file autoexec.bat;step
10
MFIL    -->  fill %1,%2,%3;mem %1
```

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

  For example, if you define the init alias as shown on page 3-17, you could enter:

  **`alias init`** ↵

  Then you would see:

```
"INIT" aliased as "load test.hbj; file autoexec.bat;step
10"
```

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

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

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

  **unalias** *alias name*

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

  **`unalias *`**

  The * symbol *does not* work as a wildcard.

**Notes:**

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

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

Chapter 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. You can use the Memory pulldown menu to enter the commands described in this chapter (see section 3.2, *Using the Menu Bar and the Pulldown Menus*, page 3-7).

Part II

**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. Memory maps vary, depending on the application.

**Note:**

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

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

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



### *Defining the memory map in a batch file*

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

- ❏ Redefine the memory map defined in the initialization batch file.
- ❏ Define the memory map in a separate batch file of your own.

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

1) It checks to see whether you 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*. If the debugger finds the file, it reads and executes the file.

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

### 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 cannot access any target-memory locations initially. Invalid memory addresses and their contents are highlighted in the data-display windows. (On color monitors, invalid memory locations, by default, are displayed in red.)

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

- **Loading a COFF file with sections that cross a memory range boundary.** Be sure that the map ranges you specify in a COFF file match those that you define with the MA command (described on page 4-5). Alternatively, you can turn memory mapping off during a load by using the MAP OFF command (described on page 4-7).

## 4.2 A Sample Memory Map

Because you must define a memory map before you can run any programs, define the memory map in the initialization batch files. Figure 4–1 *(a)* shows the memory-map commands that are defined in the initialization batch file that accompanies the HET simulator. You can use the file as is, edit it, or create your own memory map batch file, but it must match your own configuration.

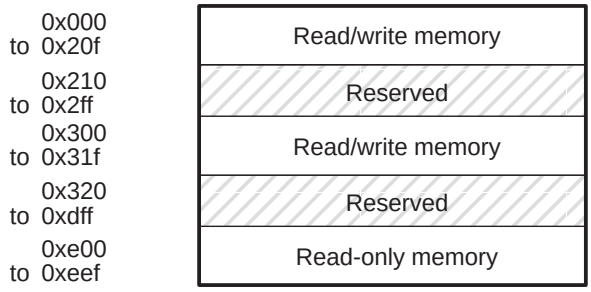
The MA (map add) commands define valid memory ranges and identify the read/write characteristics of the memory ranges. By default, mapping is enabled when you invoke the debugger. Figure 4–1 *(b)* illustrates the memory map defined by the MA commands in Figure 4–1 *(a)*.

*Figure 4–1. Sample Memory Map for Use With an HET Simulator*

*(a) Memory-map commands*

```
ma 0x000,0x210,RAM
ma 0x300,0x020,RAM
ma 0xe00,0xf0, ROM
```

*(b) Memory map for HET local memory*



## 4.3 Identifying Usable Memory Ranges

The debugger provides a command to identify usable memory ranges.

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

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

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

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

  ```
  Conflicting map range
  ```

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

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

**Notes:**

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

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

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

   does not equal

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

   If you plan to load a COFF block that spans the length of y + z, use the second MA command example. Alternatively, you could turn memory mapping off during a load by using the MAP OFF command.

## 4.4 Enabling Memory Mapping

By default, mapping is enabled when you invoke the debugger. In some cases, you may want to explicitly enable or disable memory. The debugger provides a command for this.

**map** If you want to explicitly enable or disable memory, use the MAP command. The syntax for this command is:

**map on**
or
**map off**

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

**Note:**

When memory mapping is enabled, you cannot:

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

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

```
Error in expression
```

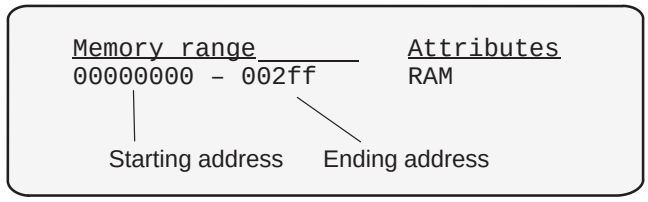
## 4.5 Checking the Memory Map

The debugger provides a command that lists the memory ranges defined for the memory map.

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

**ml**

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



## 4.6 Modifying the Memory Map During a Debugging Session

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

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

**md** *address*

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

```
Specified map not found
```

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

**mr**

This resets the debugger memory map.

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

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

The MA command is described in detail on page 4-5.

### *Returning to the original memory map*

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

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

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

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

## 4.7 Simulating External Signals

The ’470 HET simulator allows you to simulate external signals, and to specify the clock cycle where you want the signal to change. To do this, you create a data file and connect it to one of the pins: CC0–CC31, CPU, INT, MODE, and AINC

**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 for capture/compare pins*



The ’470 HET simulator allows you to simulate the external signals CC0–CC31 and to select the clock cycle where you want the input on the pin to change. To do this, you create a data file that contains a clock cycle followed by a logic value, in the following format:

**[***clock cycle***,** *logic***]**

**Note:**

The pinc command can only be used on CC pins that are configured as input pins, which is determined by the CCDIR register value. If you try to use the pinc command on a CC pin configured as an output pin, the simulator displays an error mesage “Cannot connect pin”, and ignores the pinc comand.

- The *clock cycle* parameter represents the CPU clock cycle where you want the signal to change. The *logic* parameter indicates whether the pin is high (1) or low (0).

  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 a signal. For example:

    ```
    [12,1] [56,0] [78,1]
    ```

    Signals are simulated at the 12th, 56th, and 78th CPU clock cycles. The pin is low by default. It goes high (1) at the 12th cycle and remains high until the 56th cycle. Then it goes low (0) and remains low until the 78th cycle. Then it becomes high again, and remains high until the end of the simulation. No operation is performed on the clock cycle value; the signal 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:

  ```
  [12,1] [+34,0] [55,1]
  ```

  In this example, a total of three signals 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 signals:

- **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:

  ```
  [5,1] ([+10,0] [+10,1]) rpt 2
  ```

  The values inside the parentheses represent the portion that is repeated. Therefore, a signal is simulated at the 5th CPU cycle, then the15th (5 + 10), 25th (15 + 10), 35th (25 + 10), and 45th (35 + 10) 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:

  ```
  ([10,1] [+5,0] [+20,1]) rpt EOS
  ```

  Signals 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.

❑ You can also **acccelerate** or **decelerate** input to the CC pins. This option is only available with the rpt option. To do this, you create a data file that contains a clock cycle followed by a logic and a deceleration/acceleration value, in the following format:

**[***clock cycle***,** *logic***]** {**acc** *m* | **dec** *x*} **rpt** *n*

The *clock cycle* parameter represents the CPU clock cycle where you want the signal to change.

The *logic* parameter indicates whether the pin is high (1) or low (0).

Acc/dec determines whether the operation is accelerated (acc) or decelerated (dec).

The positive integers, m and x, determine the percentage of acceleration and deceleration.

- **Acceleration and Deceleration.** To accelerate or decelerate the repeat option, you can use the keywords acc and dec followed by a number which specifies the percentage of acceleration and deceleration. For example:

```
[5,1] ([+10,0] [+10,1]) acc 12 rpt 3
([+5,0] [+5,1]) dec 20 rpt 3
```

The following table shows the clock cycle signal pattern and pin level for the input file shown above.

| Clock Cycle | Pin |
|---|---|
| 0 | Low |
| 5 | High |
| 5+10 | Low |
| 15+10 | High |
| 25+10+ Round(10*12/100) = 36 | Low |
| 36+10+ Round(10*12/100) = 47 | High |
| 47+11+ Round(11*12/100) = 59 | Low |
| 59+11+ Round(11*12/100) = 71 | High |
| 71+5 = 76 | Low |
| 76+5 = 81 | High |
| 81+5 – Round(5*20/100) = 85 | Low |
| 85+5 – Round(5*20/100) = 89 | High |
| 89+4 – Round(5*20/100) = 92 | Low |
| 92+4 – Round(5*20/100) = 95 | High |

### Setting up your input file for the CPU pin

The ’470 HET simulator allows you to simulate CPU read and write access to HET RAM and HET interface registers through the CPU pin. To do this, you create a data file that contains the clock cycle, read/write access, key word, address, and data, in the following format:

*clock cycle* {**read** | **write**} {*key word*} *address data*

- The *clock cycle* parameter represents the CPU clock cycle where you want a signal to change.
- The *read*/*write* parameter indicates whether read or write access is simulated. The default value is write.
- The *keyword* parameter determines what is read and from where. For example:
  - **w_word** indicates word to be accessed from memory
  - **reg** indicates word to be accessed from registers
  - **reg_h** indicates halfword to be accessed from registers
  - **reg_b** indicates byte to be accessed from registers
- The *address* parameter specifies the address to be accessed if a read simulation, or to be written to in case of a write simulation.
- The *data* parameter specifies the data to be written in case of a write simulation.

You can have two types of access simulation:

- **Read**. To simulate read access, create a data file that contains the clock cycle where you want the signal to change, **read**/write access, keyword, and address. For example:

  ```
  +5 read reg 0x44
  ```

- **Write**. To simulate write access, create a data file that contains the clock cycle where you want the signal to change, read/**write** access (write access is the default), keyword, address, and data for the item to be written. For example:

  ```
  +4 reg 0xc 0x66666666
  ```

  In the following example a data file called *cpu_data* is created. Read and write access is combined in one data file:

  ```
  5          w_word 0x60 0xffff5f00
  +4         reg    0xc  0x66666666
  +10        reg_h  0x6  0x007f
  +11        reg_b  0x3a 0xff
  +5  read   reg    0x44
  ```

  At clock cycle 5, `0xffff5f00` is written to address `0x60`. At clock cycle 9, `0x66666666` is written to control register OFFREG1. At clock cycle 19, `0x007f` is written to the two most significant bytes of control register PFR. At clock cycle 30, `0xff` is written to the third byte of register CCDIN. At clock cycle 32, the 32 bit value in the CCDCLR register is read.

**Note:**

The 470 HET only allows 32-bit word access from memory. If a 16-bit read/write is performed on an odd address, the previous even address is accessed. If a 32-bit read/write is performed on an address whose two least significant bits are not 0, the previous word address is accessed.

### Setting up your input file for the AINC pin

The ’470 HET simulator allows you to simulate an angle increment modification which is synchronous to an engine toothed wheel edge that is performed by the hardware angle generator. This value is required by the CNT instruction in angle mode. To do this, you create a data file includes the clock cycle, ainc (angle increment modification), and an absolute number, in the following format:

*clock cycle* **ainc** *number*

- The *clock cycle* parameter represents the CPU clock cycle where you want the increment modification to occur.
- The keyword *ainc* indicates an angle increment modification.
- The *number* parameter indicates the increment value.

  In the following example, a data file called *angle_increment_data* is created.

```
5  ainc 2
+3 ainc 1
+1 ainc 3
```

  From clock cycle 5, the angle increment value is 2. From clock cycle 8, the angle increment value is 1. From clock cycle 9, the angle increment value is 3.

### Setting up your input file for the MODE pin

Some control registers are writable only in the privileged mode. You can use the MODE pin to simulate the privileged mode. To do this, create a data file that contains clock cycle followed by a mode setting, in the following format:

*clock cycle* {**NORMAL** | **PRIVILEGED**}

- The *clock cycle* parameter represents the CPU clock cycle where you want the mode to change.
- The **NORMAL**/**PRIVILEGED** parameter indicates whether the mode is normal or privileged.

  In the following example, a data file called mode_data is created.

```
0 NORMAL
22 PRIVILEGED
```

  From clock cycle 0, the mode is normal. From clock cycle 22, the mode is privileged, allowing you to write to privileged control registers.

### Setting up your input file for the INT pin

The ’470 HET has 35 interrupt sources (32 software interrupts and 3 exceptions). The INT pin is used to simulate interrupt handling. Each interrupt source is associated with a priority level (pr1 or pr2). The HET can generate interrupts of two priority levels on any instruction that has an instruction enable bit in its instruction format.

When one or more interrupt simulations are pending on the same priority level, the offset value determines the interrupt shown. The interrupt with the lower offset value has the highest priority.

When the interrupt condition in an instruction is true and the interrupt enable bit of that instruction is set, an interrupt flag is set. The address code of this flag is determined by the five least significant bits of the current timer program address. Depending on the priority level of the interrupt given by the INTPRI control register, the offset register (OFFREG1 or OFFREG2) is set. The source number (0–34) plus 1 equals the offset value of the interrupt.

These interrupts are simulated using the INT pin. To do this, you create a data file that contains key words, an interrupt flag, address, and data information, in the following format:

{**pr1** / **pr2**} *flag* **start** {*keyword*} *address data* **end**

- The priority level (**pr1**/**pr2)** determines the order in which the interrupt will occur.
- The *flag* parameter indicates the source number of the pin.
- **start** tells the simulator when to begin writing data to HET memory.
- **end** tells the simulator when to stop writing data to HET memory.
- The *key word* describes what will be written and from where. For example:
  - **w_word** indicates word is written to HET RAM.
  - **reg** indicates word is written to the control register.
  - **reg_h** indicates half word is written to the control register.
  - **reg_b** indicates a byte is written to the control register.

  In the following example, a data file named int_data is created:

```
pr1 0 start w_word 0x56 0x00008fff
           reg 0x10 0x00ff8870
           reg_b 0x8 0x8f
           end
pr2 5 start w_word 0x36 0x0000ffff end
pr1 0 start w_word 0x40 0x000000ff
            w_word 0x56 0x00000666 end
```

A priority one software interrupt occurs and the software interrupt flag is set to 0. 0x00008fff is written at 0x56, 0x00ff8870 is written at control register OFFREG2, and 0x8f is written at the lowest byte of control register HETADDR.

The second priority one software interrupt occurs, and the software interrupt flag is set to 0. 0x000000ff is written at 0x40, and 0x00000666 is written at 0x56 in HET memory.

The priority two software interrupt occurs, and the flag register bit is set to 5. 0x0000ffff is written at 0x36 in HET memory.

**Note:**

The 470 HET only allows 32-bit word access from memory. If a 16-bit read/write is performed on an odd address, the previous even address is accessed. If a 32-bit read/write is performed on an address whose two least significant bits are not 0, the previous word address is accessed.



### *Connecting your input file to the simulated pin*

To connect your input file to the simulated pin, use the following command:

**pinc** *pinname, filename*

- The *pinname* identifies the pin and must be one of the simulated pins: CC0-CC31, CPU, INT, MODE, or AINC.
- The *filename* is the name of your input file.

Example 4–1 shows you how to connect your input file to the external signal (CC0-CC31, CPU, AINC, MODE, and INT) using the PINC command.

*Example 4–1. Connecting the Input File With the PINC Command*

Suppose you want to generate an external signal on CC31 at the 12th, 34th, 56th, and 89th clock cycles.

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

```
[12,1] [34,0] [56,1] [89,0]
```

To connect the input file to the pin, enter:

```
pinc CC31 myfile                    Connects your data file
                                    to the specific interrupt pin
```

This command connects myfile to the CC31 pin. As a result, the simulator changes the input on CC31 to 1, 0, 1, and 0, respectively, at the 12th, 34th, 56th, and 89th clock cycles.

### Disconnecting your input file from the pin

To end the external signal simulation, use the **pind** command to disconnect the pin. The syntax for this command is:

**pind** *pinname*

The **pind** command detaches the file from the simulated pin. After executing this command, you can connect another file to the same pin.

### Listing the simulated pins and connecting input files

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

**pinl**

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

## 4.8 Running a Trace (simulator only)

The HET simulator produces a trace file whenever a program is loaded. The trace file contains information about the program loaded and traces the following items:

- ❑ Clock count
- ❑ Loop start
- ❑ HET registers
- ❑ Control registers
- ❑ Instruction field registers
- ❑ Status flags
- ❑ CC pins
- ❑ Instruction field row address
- ❑ Instruction address
- ❑ Instruction mnemonic

When you load a program, the simulator looks for an associated file with an .INI extension in the same directory as the .hbj file. If the associated .INI file does not exist, the simulator looks for the default .INI file, TRACE.INI, in the directory from which the simulator was invoked. The .INI file tells the simulator to write the trace results to a file it creates (with a .TRA extension). The .TRA file is formatted with the entries from the .INI file as column headings. The column headings are repeated every 40 lines to make browsing easy for the user. You will also see a “new loop start” line in the trace file before the cycle entry on which the loop starts. The extensions of the files discussed in this section (.INI and .TRA) are case sensitive.

For example, a program called sample.hbj is loaded on the simulator. The simulator finds a file called sample.INI in the same directory as the .hbj file. Using the entries from the .INI file, the simulator creates a sample.TRA file to which the trace results are written. Example 4–2 shows the sample.INI file and its associated sample.TRA output.

## Example 4–2. Running a Trace Using the sample.INI File

*(a) sample.INI (user created .INI file)*

```
CLOCK
HETADDR
#CF
```

*(b) sample.TRA (trace output file)*

```
      CLOCK          HETADDR        #CF
       1440               0f     110758
       1441               10     1117d8
       1442               11     112858
       1443               12     1138d8
       1444               13     114958
       1445               14     1159d8
       1446               15     116a58
       1447               16     117ad8
       1448               17     118b58
       1449               18     119bd8
       1450               19     11ac58
```



The above example uses the sample.INI file to trace the clock cycle, instruction address, and control field register of the sample.hbj program, and writes the results to the sample.TRA file.

If the sample.INI file is not found, the simulator looks for the default file, TRACE.INI in the directory from which the simulator was invoked. For example:

A program called sample.hbj is loaded on the simulator. The simulator looks for a file called sample.INI in the same directory as the .hbj file. The sample.INI file is not found, so the simulator finds the default TRACE.INI file in the directory from which the simulator was invoked. Using entries from the .INI file, the simulator creates a sample.TRA file to which the trace results are written. Example 4–3 shows the TRACE.INI file and its associated sample.TRA output.

*Example 4–3. Running a Trace Using the Default TRACE.INI File*

*(a) TRACE.INI (the default .INI file)*

```
CLOCK
HETADDR
;CCDIR CCDIN
;AREG BREG TREG
;#PF
#CF
#DF
;ZF
;CCPINS
CF 0x20
```

*(b) sample.TRA (trace output file)*

```
     CLOCK    HETADDR      #CF       #DF  CF00000020
      1440         0f    110758    00004e    00100fd8
      1441         10    1117d8    00004f    00100fd8
      1442         11    112858    000050    00100fd8
      1443         12    1138d8    000051    00100fd8
      1444         13    114958    000052    00100fd8
      1445         14    1159d8    000053    00100fd8
      1446         15    116a58    000054    00100fd8
      1447         16    117ad8    000055    00100fd8
      1448         17    118b58    000056    00100fd8
      1449         18    119bd8    000057    00100fd8
      1450         19    11ac58    000058    00100fd8
```

The above example uses the TRACE.INI file to trace the clock cycle, instruction address, control field register, data field register, and control field row address 0x20, of the sample.hbj program, and writes the results to the sample.TRA file. The remaining fields of the .INI file are commented out using a semicolon.

### *Creating a trace file (.INI)*

The .INI file contains the list of items that you want to trace. The simulator uses this list to create headers for the .TRA file. The headers are created in the same order that you list them in the .TRA file. The available items to trace are listed below:

- ❏ Clock count

  `CLOCK`
- ❏ Instruction field registers

  `#PF, #CF, #DF`
- ❏ Instruction field row address

  `{`**`PF`** `|` **`CF`** `|` **`DF`**`} {row address}`
- ❏ HET registers

  `AREG, BREG, TREG`
- ❏ Control registers

  `GCR, PFR, OFFREG1, OFFREG2, EXCCTRL1, EXCCTRL2, INTPRI, INTFLAG, HRSHARE, CCDIR, CCDIN, CCDWR, CCDSET, CCDCLR`
- ❏ Instruction mnemonic

  `INSTR`
- ❏ Instruction address

  `HETADDR`
- ❏ Status flags

  `ACF, DCF, NAF, GPF, ZF, XF`
- ❏ CC pins (all):

  `CCPINS`
- ❏ CC pins

  `CC0, CC1, CC2, CC3, CC4, CC5, CC6, CC7, CC8, CC9, CC10, CC11, CC12, CC13, CC14, CC15, CC16, CC17, CC18, CC19, CC20, CC21, CC22, CC23, CC24, CC25, CC26, CC27, CC28, CC29, CC30, CC31`

You can choose any combination of these items when you create or customize your .INI file. You must name your file with the case sensitive .INI extension. To customize the .INI file, make comments of the entries already in the file. To make a comment of an existing entry, simply preceed the entry with a semicolon.

**Note: Incomplete .TRA file**

The .TRA file will be incomplete if you do not first exit the simulator with the **quit** command, or load another .hbj file.

Chapter 5

# Loading, Displaying, and Running Code

The main purpose of a debugging system is to allow you to load and run your programs in a test environment. This chapter tells you how to load your programs into the debugging environment, run them on the target system, and view the associated source code. You can use the Load pulldown menu to execute many of the commands described in this chapter (see section 3.2, *Using the Menu Bar and the Pulldown Menus*, page 3-7).

Part II

**Topic** **Page**

## 5.1 Displaying Your Source Programs

The debugger displays *assembly language code* in the DISASSEMBLY window. The DISASSEMBLY window displays code that the current address register (HETADDR) points to.

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

### *Displaying assembly language code*

The assembly language code in the DISASSEMBLY window is the reverse assembly of program-memory contents. (This code does not come from the intermediate assembly files produced by the compiler.)



When you invoke the debugger, it comes up in assembly 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.

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

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

**dasm** *address*

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

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

**addr** *address*

ADDR works like the DASM command, positioning the code starting at *address* as the first line of code in the DISASSEMBLY window.

## 5.2 Loading Object Code

To debug a program, you must load the program's object code into memory. You can do this as you invoke the debugger, or you can do it after you invoke the debugger. (You create an object file by compiling and assembling your source files; see section 1.3, *Preparing Your Program for Debugging*, on page 1-6.)

### *Loading code while invoking the debugger*

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

If you want to load a file's symbol table only, use the –s option (this has the same effect as using the debugger's SLOAD command). To do this, enter the appropriate debugger-invocation command along with the name of the object file and specify –s (see page 1-10 for more information).

If you want to load an object file only, use the –v option (this has the same effect as using the debugger's RELOAD command). To do this, enter the appropriate debugger-invocation command along with the name of the object file and specify –v (see page 1-10 for more information).

### *Loading code after invoking the debugger*

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

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

**load** *object filename*

If you do not supply an extension, the debugger looks for *filename*.hnc.

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

**reload** [*object filename*]

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

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

**sload** *object filename*

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

## 5.3 Where the Debugger Looks for Source Files

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

You can tell the debugger where to search for a source file in these ways:

- When using LOAD, RELOAD, or SLOAD, you can specify the path as part of the filename. You have two choices for supplying the path information:
  - Specify the path as part of the filename.
  - Alternatively, if you are using Windows, you can use the CD command to change the current directory from within the debugger. The format for this command is:

    **cd** *directory name*

- Within the operating-system environment, you can name additional directories with the D_SRC environment variable. The format for doing this is:

  **SET D_SRC=***pathname;pathname* For Windows or OS/2™

  **setenv D_SRC "***pathname;pathname***"** For UNIX

  This allows you to name several directories that the debugger can search. The list of source directories that you create when you set the environment variable is valid until you change them.

- 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 ../../code. The debugger can recognize a cumulative total of 20 paths specified with D_SRC, –i, and the USE command.

## 5.4 Running Your Programs

To debug your programs, you must execute them on the simulator. The debugger provides two basic types of commands to help you run your code:

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

### *Defining the starting point for program execution*

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

- Finding its entry in the CPU window
- Finding the appropriately highlighted line in the DISASSEMBLY window. To do this, execute one of these commands:

  **dasm HETADDR**
  or
  **addr HETADDR**

### Running code (basic commands)

The debugger supports these basic commands for running all or part of a program.

**run** Use the RUN command to run an entire program. The format for this command is:

**run** [*expression*]

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

- ❏ If you do not supply an *expression*, the program executes until it encounters a breakpoint or until you press ESC or the left mouse button.
- ❏ If you supply a logical or relational *expression*, this becomes a conditional run (see page 5-11).
- ❏ If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger executes *count* instructions, halts, then updates the display.

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

**go** [*address*]

If you do not supply an *address* parameter, GO acts like a RUN command without an *expression* parameter—the program executes until it encounters a breakpoint or until you press ESC or the left mouse button. For more information about expressions, see chapter 10, Basic *Information About C Expressions*.


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

### Single-stepping through code

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

The debugger supports several commands for single-stepping through a program. The debugger ignores interrupts when you use the STEP command to single-step through assembly language code.

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

- ❑ If you 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 page 5-11).
- ❑ If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger single-steps *count* assembly language statements.

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

**step** [*expression*]

The debugger executes one assembly language statement at a time.

**next** The NEXT command is similar to the STEP command. It always steps to the next consecutive statement. The format for this command is:

**next** [*expression*]

For more information about expressions, see Chapter 10, Basic *Information About C Expressions*.


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

F8 Acts as a STEP command.

F10 Acts as a NEXT command.

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

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

To execute a STEP:

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

2) Press and release the left mouse button.

To execute a NEXT:

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

2) Press and release the left mouse button.

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



### *Resetting the simulator*

The debugger provides a command that allows you to reset the target system.

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

**reset**

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

### Running code conditionally

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

| | | |
|---|---|---|
| > | > = | < |
| < = | = = | ! = |
| && | \| \| | ! |

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

You must use software breakpoints with conditional runs; the expression is evaluated each time the debugger encounters a breakpoint. (Breakpoints are described in Chapter 7, *Using Software Breakpoints*.) Each time the debugger evaluates the conditional expression, it updates the screen. The debugger applies this algorithm:

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

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

## 5.5 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*). You can explicitly halt program execution in two ways:

Click the left mouse button.


ESC Press the escape key.

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

Chapter 6

# Managing Data

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

- ❏ Individual memory locations or a range of memory
- ❏ HET registers

Part II

**Topic** **Page**

## 6.1 Where Data Is Displayed

Three windows, referred to as *data-display windows*, display the various types of data.

| Type of Data | Window Name and Purpose |
|---|---|
| Memory locations | **MEMORY window**<br>Displays the contents of a range of data memory, program memory, or I/O space |
| Register values | **CPU window**<br>Displays the contents of HET registers |
| Specific memory locations or registers | **WATCH window**<br>Displays selected data |



## 6.2 Basic Commands for Managing Data

The debugger supports the following general-purpose commands that you can use to display or modify any type of data.

**?** 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. (See *Advanced editing—using expressions with side effects* on page 6-5 for more information about side effects.) If the result of *expression* is scalar, the debugger displays the result as a decimal value in the COMMAND window. If *expression* is a structure or array, ? displays the entire contents of the structure or array; you can halt long listings by pressing ESC.

Here are some examples that use the ? command.

| Command | Result Displayed in the COMMAND Window | Example |
|---|---|---|
| **? A** | Value of register A | `0xffff` |
| **? HETADDR** | Value of HETADDR | `0x0100` |
| **? *0x0100** | Content of memory address | `0100h` |

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

**eval** *expression*
or
**e** *expression*

The *expression* parameter is the same as for the ? command.

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

## 6.3 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 displayed in the WATCH window

There are two similar methods for overwriting displayed data:



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

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

2) Click the left button. The debugger highlights the selected field, and the window containing this field becomes active.

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

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


1) Select the window that contains the field you want to modify; make this the active window. (Use the mouse, the WIN command, or F6. For more details, see section 2.4, *The Active Window*, on page 2-15.)

2) Use arrow keys to move the cursor to the field you want to edit.

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

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

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

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

**Note:**

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

### *Advanced editing—using expressions with side effects*

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

This means that you can coerce many commands into changing values for you. Specifically, it 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 auxiliary register A, you can enter:

```
? A ↵
```

You can also use this type of command to modify the contents of auxiliary register A. Here are some examples of how you might do this:

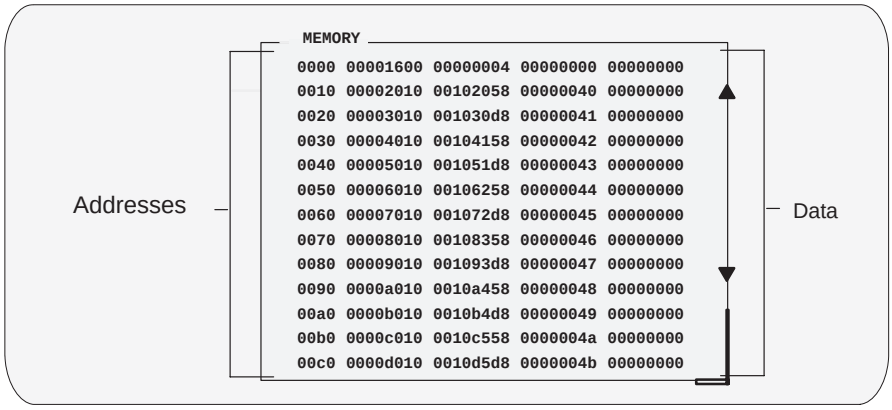
| | |
|---|---|
| `? A++ ↵` | *Side effect: increments the contents of A by 1* |
| `eval --A ↵` | *Side effect: decrements the contents of A by 1* |
| `? A = 8 ↵` | *Side effect: sets A to 8* |
| `eval A/=2 ↵` | *Side effect: divides contents of A by 2* |

Not all expressions have side effects. For example, if you enter **? A+4**, the debugger displays the result of adding 4 to the contents of A but does not modify the contents of A. 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:

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

## 6.4 Managing Data in Memory

In assembly mode, the debugger maintains a MEMORY window that displays the contents of memory. For details concerning the MEMORY window, see *MEMORY window* on page 2-7.




The debugger commands show the memory values at a specific location or display a different range of memory in the MEMORY window. The debugger allows you to change the values at individual locations; for more information, see section 6.3, *Basic Methods for Changing Data Values*.

### *Displaying memory contents*

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

The amount of memory that you can display in a MEMORY window is limited by the size of the window (which is limited only by the screen size).

**mem** You can use the MEM command to open an additional MEMORY window or to display a different memory range in a window. The syntax for this command is:

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

- The *expression* represents the address of the first entry in the MEMORY window. The end of the range is defined by the size of the window: to show more memory locations, make the window larger; to show fewer locations, make the window smaller. For more information, see *Resizing a window* on page 2-18.

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

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

  ```
  mem 0x00
  ```

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

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

  ```
  mem &SYM
  ```

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

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

  ```
  mem SP – A0 + label
  ```

- The *display format* parameter is optional. When used, the data is displayed in the selected format as shown in Table 6–1 on page 6-15.

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

You can also change the display of any data-display window—including the MEMORY window—by scrolling through the window's contents. For more details, see *Scrolling through a window's contents* on page 2-23.

### Saving memory values to a file

Sometimes it is useful to save a block of memory values to a file. The debugger provides a command for this.

**ms** If you want to save a block of memory values to a system file, use the MS (memory save) command. The files are saved in COFF format. The syntax for the MS command is:

**ms** *address, length, filename*

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

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

```
ms 0x0,0x10,memsave ↵
```

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

```
load memsave.hbj ↵
```

### Filling a block of memory

Sometimes it is useful to fill an entire block of memory at one time with a single value. The debugger provides a command for this.

**fill** If you want to fill a block of memory with a specified value, use the FILL command. The syntax for this command is:

**fill** *address, length, data*

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

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

```
fill 0x10ff,0xf,0xabcd ↵
```

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

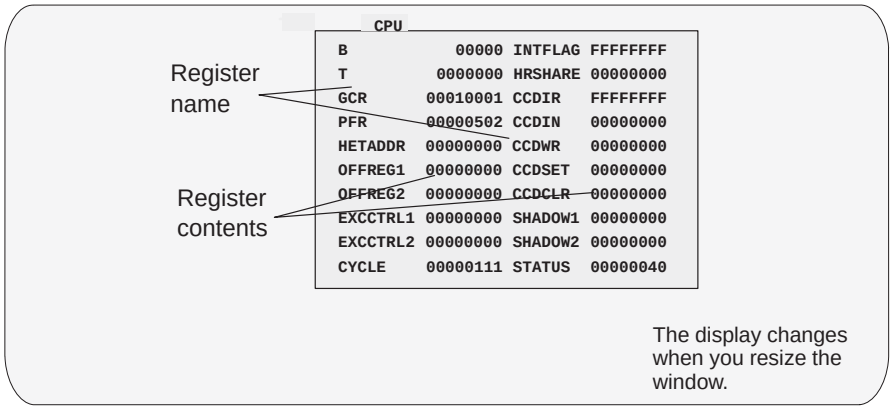
```
mem 0x10ff ↵
```

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

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

## 6.5 Managing Register Data

In assembly modes, the debugger maintains a CPU window that displays the contents of individual registers. For details, see *CPU window* on page 2-10.




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 6.3, *Basic Methods for Changing Data Values*.

### *Displaying register contents*

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

- 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 A, you could enter:

  **? A** ↵

  The debugger displays the current contents of register A in the display area of the COMMAND window.

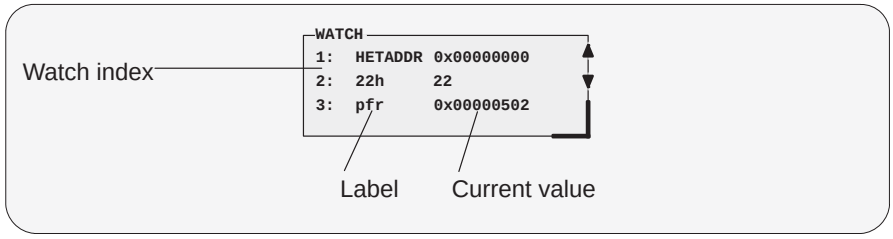
- If you want to observe a register over a longer period of time, you can use the WA command to display the register in a WATCH window. For example, if you want to observe the current address register (HETADDR), you could enter:

  **wa HETADDR, Register 0** ↵

  This adds HETADDR to the WATCH window and labels it as *Register 0*. The register's contents are continuously updated, just as if you were observing the register in the CPU window.

## 6.6 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 create a WATCH window that shows you how program execution affects specific expressions, variables, registers, or memory locations.

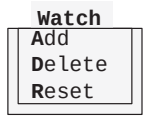



The debugger displays a WATCH window *only when you specifically request a WATCH window* with the WA command (described below). For additional details concerning the WATCH window, see *WATCH window* on page 2-12.

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 6.3, *Basic Methods for Changing Data Values,* on page 6-4.

**Note:**

All of the watch commands described can also be accessed from the Watch pulldown menu. For more information about using the the pulldown menus, see section 3.2, *Using the Menu Bar and the Pulldown Menus*, on page 3-7.



### *Displaying data in the WATCH window*

The debugger has one command for opening a WATCH window and displaying data.

**wa** To open a WATCH window and add items to it, use the WA (watch add) command. The syntax is:

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

When you first execute WA, the debugger opens a WATCH window. After that, executing WA adds additional values to the WATCH window, unless you open an additional watch window.

- The *expression* parameter can be any C expression, including an expression that has side effects. It is most useful to watch an expression whose value changes over time; constant expressions provide no useful function in the WATCH window.

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

  ```
  wa *0x26
  ```

- The *label* parameter is optional. When used, it provides a label for the watched entry. If you do not use a *label*, the debugger displays the *expression* in the label field.

- The *display format* parameter is optional. When used, the data is displayed in the selected format as shown in Table 6–1 on page 6-15.

- The *window name* parameter is optional. If you omit the *window name* parameter, the debugger displays the expression in the default WATCH window (labeled WATCH). You can open additional WATCH windows by using the *window name* parameter. When you open an additional WATCH window, the debugger appends the *window name* to the WATCH window label. You can create as many WATCH windows as you need.

### *Deleting watched values and closing the WATCH window*

The debugger supports two commands for deleting items from the WATCH window.

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

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

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

**wd** If you want to delete a specific item from a WATCH window, use the WD (watch delete) command. The syntax is:

**wd** *index number* [, *window name*]

Whenever you add an item to a WATCH window, the debugger assigns it an index number. (The illustration of the WATCH window on page 6-12 points to these watch indexes.) The WD command's *index number* parameter must correspond to one of the watch indexes in the named WATCH window.

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

**Note:**

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

## 6.7 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 6–1 lists the available formats and the corresponding characters that can be used as the *display format* parameter.

*Table 6–1. Display Formats for Debugger Data*

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

Table 6–2 lists the C data types that can be used for the *data type* parameter. Only a subset of the display formats applies to each data type, so Table 6–2 also shows valid combinations of data types and display formats.

*Table 6–2. Data Types for Displaying Debugger Data*

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

Here are some examples:

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

  **`setf short, u`** ↵

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

  **`setf short, *`** ↵

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

  **`setf`** ↵

  You see a display that looks something like this:

```
Type Format Defaults
char   : ASCII
uchar  : Unsigned decimal
int    : Decimal
uint   : Unsigned decimal
short  : Decimal
ushort : Unsigned decimal
long   : Decimal
ulong  : Unsigned decimal
float  : Exponential floating point
double : Exponential floating point
ptr    : Hexadecimal
```

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

  **`setf *`** ↵

### Changing the default format with ?, MEM, and WA

You can also use the ?, MEM, and WA commands to show data in alternative display formats. 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.

Here are some examples:

- To watch the GCR (global configuration register) in decimal, enter:

  ```
  wa gcr,,d ↵
  ```

- To display memory contents in octal, enter:

  ```
  mem 0x0,o ↵
  ```

- To display the default format representation of –3, enter:

  ```
  ? -3 ↵
  ```

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

Chapter 7

# Using Software Breakpoints

During the debugging process, you may want to halt execution temporarily so that you can examine the contents of selected variables, registers, and memory locations before continuing with program execution. You can do this by setting *software breakpoints* at critical points in your code. 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 on page 5-11).



**Topic** **Page**

## 7.1 Setting a Software Breakpoint

When you set a software breakpoint, the debugger highlights the breakpointed line in two ways:

- ❑ It prefixes the statement with the characters BP>.
- ❑ It shows the line in a bolder or brighter font. (You can use screen-customization commands to change this highlighting method.)

**Notes:**

1) On the HET, execution halts after completing the execution of the instruction on which the breakpoint is set. This is consistant with the built-in debuging capability of the HET architecture.
2) After a breakpoint halts execution, you can continue running a program by reissuing any of the run or single-step commands.
3) You can set up to 200 breakpoints.

There are several ways to set a software breakpoint:

1) Point to the line of assembly language code where you want to set a breakpoint.
2) Click the left button. The breakpoint is set.

*Repeating this action clears the breakpoint.*


1) Make the DISASSEMBLY window the active window.
2) Use the arrow keys to move the cursor to the line of code where you would like to set a breakpoint.
3) Press the F9 key. The breakpoint is set.

*Repeating this action clears the breakpoint.*

**ba** If you know the address where you would like to set a software breakpoint, you can use the BA (breakpoint add) command. This command does not require

you to search through code to find the desired line. The syntax for the BA command is:

**ba** *address*

This command sets a breakpoint at *address*. This parameter can be an absolute address, any C expression, or the name of an assembly language label. You cannot set multiple breakpoints at the same statement.

## 7.2 Clearing a Software Breakpoint

There are several ways to clear a software breakpoint.

1) Point to a breakpointed assembly language statement.

2) Click the left button. The breakpoint is cleared.


1) Use the arrow keys or the DASM command to move the cursor to a breakpointed assembly language statement.

2) Press the F9 key. The breakpoint is cleared.

**br** If you want to clear *all* the software breakpoints that are set, use the BR (breakpoint reset) command. This command does not require you to search through code to find the desired line. The syntax for the BR command is:

**br**

**bd** If you would like to clear one specific software breakpoint and you know the address of this breakpoint, you can use the BD (breakpoint delete) command. The syntax for the BD command is:

**bd** *address*

This command clears the breakpoint at *address*. This parameter can be an absolute address, any C expression, or the name of an assembly language label. If no breakpoint is set at *address*, the debugger ignores the command.

## 7.3 Finding the Software Breakpoints That Are Set

Sometimes you may need to know where software breakpoints are set. For example, the BD command's *address* parameter must correspond to the address of a breakpoint that is set. The debugger provides a command for listing software breakpoints.

**bl** The BL (breakpoint list) command provides an easy way to get a complete listing of all the software breakpoints that are currently set in your program. The syntax for this command is:

**bl**

The BL command displays a table of software breakpoints in the display area of the COMMAND window. BL lists all the software breakpoints that are set, in the order in which you set them. Here is an example of this type of list:

```
 Address      Symbolic Information
00100         start
00108
00120         bottom
```

The address is the memory address of the breakpoint. If the statement where you set a breakpoint has a label, the debugger also displays the label name. You see only an address unless the statement defines a symbol.

Chapter 8

# Customizing the Debugger Display

The debugger display is completely configurable; you can create the interface that is best suited for your use. Besides being able to size and position individual windows, you can change the appearance of many of the display features, such as window borders, how the current statement is highlighted, etc. In addition, if you are using a color display, you can change the colors of any area on the screen. Once you have customized the display, you can save the custom configuration for use in future debugging sessions.



**Topic** **Page**

## 8.1 Changing the Colors of the Debugger Display

You can use the debugger with a color or a monochrome display; the commands described in this section are most useful if you have a color display. If you are using a monochrome display, these commands change the shades on your display. For example, if you are using a black-and-white display, these commands change the shades of gray that are used.

**color**
**scolor**

You can use the COLOR or SCOLOR command to change the colors of areas in the debugger display. The format for these commands is:

**color** *area name*, *attribute*$_1$ [, *attribute*$_2$ [, *attribute*$_3$ [, *attribute*$_4$ ] ] ]
**scolor** *area name*, *attribute*$_1$ [, *attribute*$_2$ [, *attribute*$_3$ [, *attribute*$_4$ ] ] ]

Although these commands are similar, SCOLOR updates the screen immediately, and COLOR does not update the screen (the new colors/attributes take effect as soon as the debugger executes another command that updates the screen). You might use the COLOR command several times, followed by an SCOLOR command to put all of the changes into effect at once.

The *attributes* identify how the areas of the display are affected. Table 8–1 lists the valid values for the *attribute* parameters. The *area name* parameter identifies the area of the display that is affected. Table 8–2 lists valid values for the *area name* parameters. The subsections following Table 8–2 further identify these areas.

*Table 8–1. Colors and Other Attributes for the COLOR and SCOLOR Commands*

| Colors | | Other Attributes |
|---|---|---|
| black | blue | bright |
| green | cyan | blink |
| red | magenta | |
| yellow | white | |

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

*Table 8–2. Summary of Area Names for the COLOR and SCOLOR Commands*

| Area Names | | | |
|---|---|---|---|
| menu_bar | menu_border | menu_entry | menu_cmd |
| menu_hilite | menu_hicmd | win_border | win_hiborder |
| win_resize | field_text | field_hilite | field_edit |
| field_label | field_error | cmd_prompt | cmd_input |
| cmd_cursor | cmd_echo | asm_data | asm_label |
| background | blanks | error_msg | file_text |
| file_brk | file_pc | file_pc_brk | |

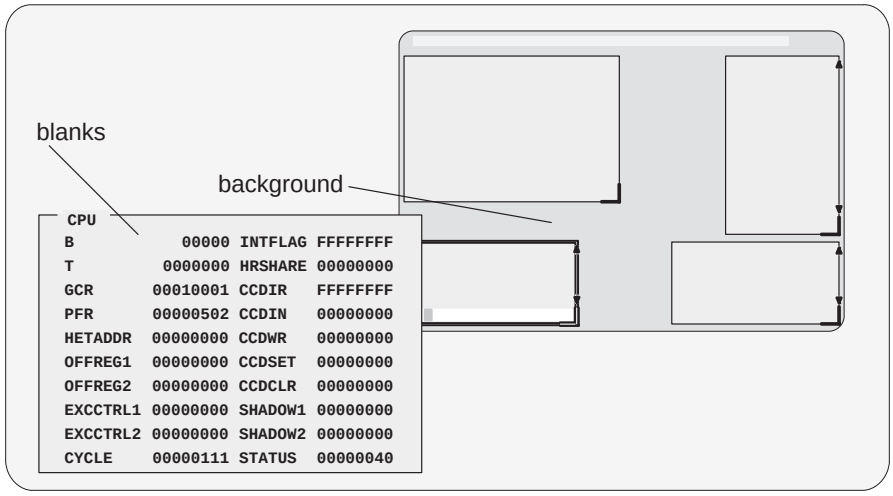
**Note:** Listing order is left to right, top to bottom.

You do not have to type an entire *attribute* or *area name*; you need to type only enough letters to uniquely identify either parameter. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they are listed in Table 8–2 (left to right, top to bottom).

The remainder of this section identifies these areas.

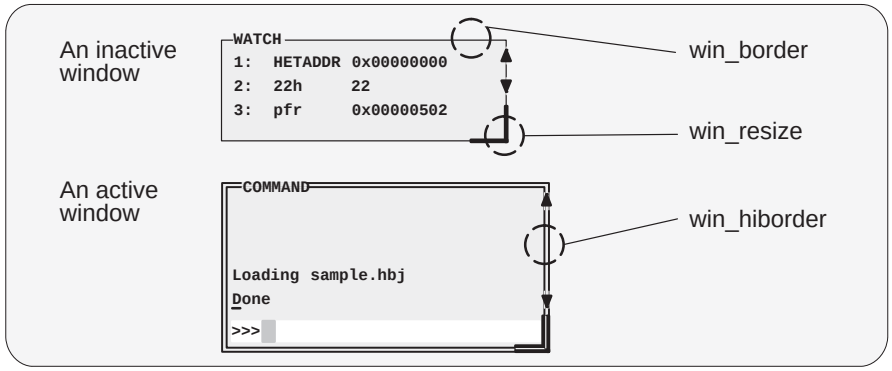


### *Area names: common display areas*



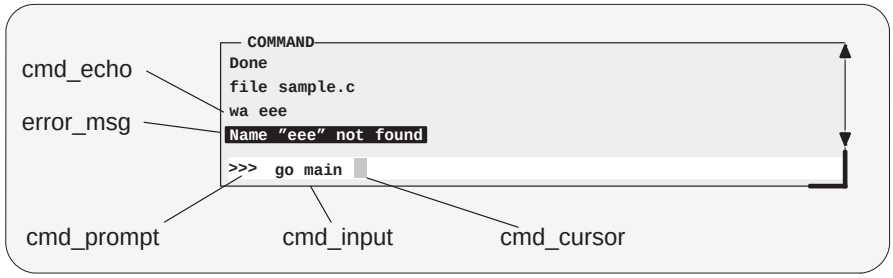

| Area Identification | Parameter Name |
|---|---|
| Screen background (behind all windows) | background |
| Window background (inside windows) | blanks |

### Area names: window borders



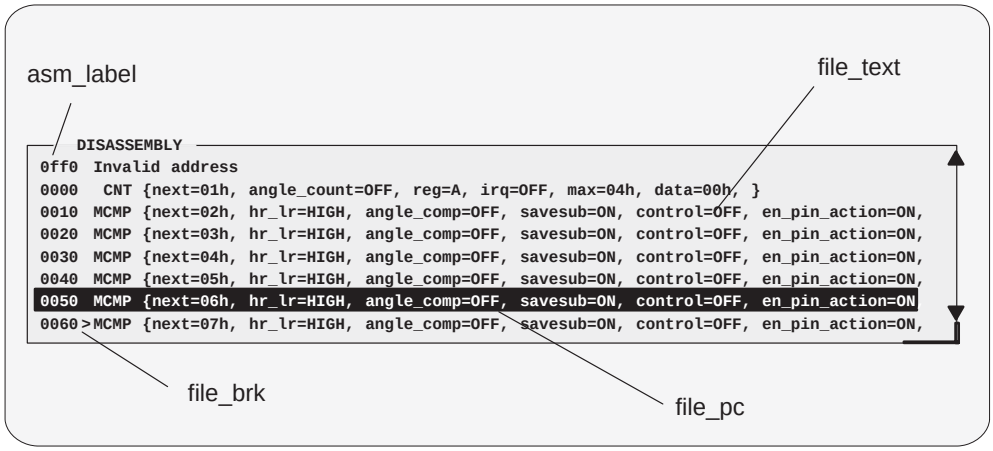

| Area Identification | Parameter Name |
|---|---|
| Window border for any window that is not active | win_border |
| The reversed L in the lower right corner of a resizable window | win_resize |
| Window border of the active window | win_hiborder |

### Area names: COMMAND window



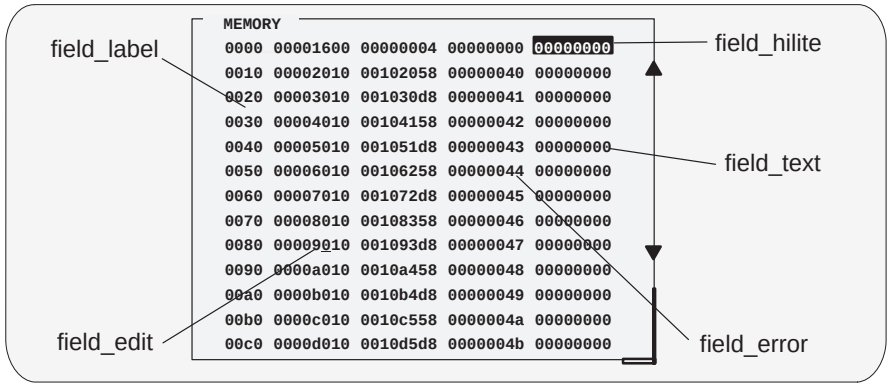

| Area Identification | Parameter Name |
|---|---|
| Echoed commands in display area | cmd_echo |
| Errors shown in display area | error_msg |
| Command-line prompt | cmd_prompt |
| Text that you enter on the command line | cmd_input |
| Command-line cursor | cmd_cursor |

### *Area names: DISASSEMBLY window*



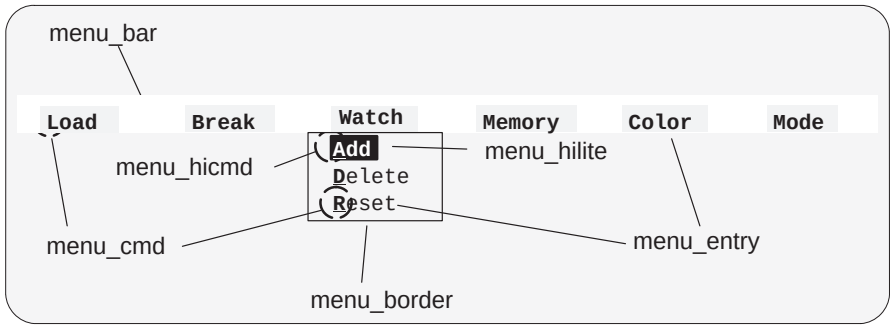

| Area Identification | Parameter Name |
|---|---|
| Addresses in DISASSEMBLY window | asm_label |
| Text in DISASSEMBLY window | file_text |
| Breakpointed text in DISASSEMBLY window | file_brk |
| HETADDR in DISASSEMBLY window | file_pc |

### Area names: data-display windows




| Area Identification | Parameter Name |
|---|---|
| Label of a window field (includes register names in CPU window, addresses in MEMORY window, index numbers and labels in WATCH window) | field_label |
| Text of a window field (includes data values for all data-display windows) and of most command output messages in command window | field_text |
| Text of a highlighted field | field_hilite |
| Text of a field that has an error (such as an invalid memory location) | field_error |
| Text of a field being edited (includes data values for all data-display windows) | field_edit |

### Area names: menu bar and pulldown menus




| Area Identification | Parameter Name |
|---|---|
| Top line of display screen; background to main menu choices | menu_bar |
| Border of any pulldown menu | menu_border |
| Text of a menu entry | menu_entry |
| Invocation key for a menu or menu entry | menu_cmd |
| Text for current (selected) menu entry | menu_hilite |
| Invocation key for current (selected) menu entry | menu_hicmd |

## 8.2 Changing the Border Styles of the Windows

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

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

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

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

| Index | Style |
|---|---|
| 0 | Double-lined box |
| 1 | Single-lined box |
| 2 | Solid 1/2-tone top, double-lined sides and bottom |
| 3 | Solid 1/4-tone top, double-lined sides and bottom |
| 4 | Solid box, thin border |
| 5 | Solid box, heavy sides, thin top and bottom |
| 6 | Solid box, heavy borders |
| 7 | Solid 1/2-tone box |
| 8 | Solid 1/4-tone box |

Here are some examples of the BORDER command. You can skip parameters, if desired.

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

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

## 8.3 Saving and Using Custom Displays

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

When you invoke the debugger, it looks for a screen configuration file called init.clr. The screen configuration file defines how various areas of the display will appear. If the debugger does not find this file, it uses the default screen configuration. Initially, init.clr defines screen configurations that exactly match the default configuration.

The debugger supports two commands (SSAVE and SCONFIG) for saving and restoring custom screen configurations into files. The filenames that you use for restoring configurations must correspond to the filenames that you used for saving configurations. These are binary files, not text files, so you cannot edit the files with a text editor.



### *Changing the default display for monochrome monitors*

The default display is most useful with color monitors. The debugger highlights changed values, messages, and other information with color; this may not be particularly helpful if you are using a monochrome monitor.

The debugger package includes another screen configuration file named mono.clr, which defines a screen configuration that can be used with monochrome monitors. The best way to use this configuration is to rename the file:

1) Rename the original init.clr file—you might want to call it color.clr.

2) Next, rename the mono.clr file. Call it init.clr. Now, whenever you invoke the debugger, it automatically comes up with a customized screen configuration for monochrome monitors.

If you are not happy with the way that this file defines the screen configuration, you can customize it.

### Saving a custom display

Once you customize the debugger display, you can save the new screen configuration using this command:

**ssave** Use the SSAVE command to save the customized screen configuration to a file. The format for this command is:

**ssave** [*filename*]

This saves the current screen resolution, border styles, colors, window positions, window sizes, and (on PCs) video mode (EGA, VGA, etc.) for all debugging modes.

The *filename* parameter names the new screen configuration file. You can include path information (including relative pathnames); if you do not specify 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.

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

**Note:**

The file created by the SSAVE command in this version of the debugger saves positional, screen size, and video mode information that was not saved by SSAVE in previous versions of the debugger. The format of this new information is not compatible with the old format. If you attempt to load an earlier version's SCONFIG file, the debugger issues an error message and stops the load.



### Loading a custom display

The debugger provides a command to load a custom display.

**sconfig** You can use the SCONFIG command to restore the display to a particular configuration. The format for this command is:

**sconfig** [*filename*]

This restores the screen resolution, colors, window positions, window sizes, border styles, and (on PCs) video mode (EGA, CGA, MDA, etc.) saved in *filename*. Screen resolution and video mode are restored either by changing the mode (on video cards with switchable modes) or by resizing the debugger screen (on other hosts).

If you do not supply a *filename*, the debugger looks for init.clr. The debugger searches for the file in the current directory and then in directories named with the D_DIR environment variable. (For information about setting the D_DIR environment variable, see the CD-ROM insert.)

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

### Invoking the debugger with a custom display

If you customize the screen and always want to invoke the debugger with this screen configuration, you can accomplish this in two ways:

- ❏ Save the configuration in init.clr.
- ❏ Add a line to the batch file that the debugger executes at invocation time (init.cmd). This line should use the SCONFIG command to load the custom configuration.



### Returning to the default display

If you saved a custom configuration into init.clr but do not want the debugger to come up in that configuration, rename the file or delete it. If you are in the debugger, have changed the configuration, and want to revert to the default, just execute the SCONFIG command without a filename.

## 8.4 Changing the Prompt

Another way that you can customize your display is to change the command-line prompt.

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

**prompt** *new prompt*

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

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

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

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

*Part I*

# *Hands-On Information*

*Part II*

# *Debugger Description*

*Part III*

# *Reference Material*

# Part III

Chapter 9

# Summary of Commands and Special Keys

This chapter summarizes the basic debugger commands and the debugger's special key sequences.

**Topic** **Page**

## 9.1 Functional Summary of Debugger Commands

This section summarizes the debugger commands according to these categories:

- **Changing modes.** These commands (listed on page 9-3) enable you to switch freely between the debugging modes (minimal and assembly).
- **Managing windows.** These commands (listed on page 9-3) enable you to select the active window and move or resize the active window.
- **Displaying and changing data.** These commands (listed on page 9-3) enable you to display and evaluate a variety of data items.
- **Performing system tasks.** These commands (listed on page 9-4) enable you to perform several DOS-like functions and provide you with some control over the target system.
- **Managing breakpoints.** These commands (listed on page 9-4) provide you with a command line method for controlling software breakpoints.
- **Displaying files and loading programs.** These commands (listed on page 9-5) enable you to change the display in the DISASSEMBLY window and to load object files into memory.
- **Customizing the screen.** These commands (listed on page 9-5) allow you to customize the debugger display, then save and later reuse the customized displays.
- **Memory mapping.** These commands (listed on page 9-5) enable you to define the areas of target memory that the debugger can access.
- **Running programs.** These commands (listed on page 9-6) provide you with a variety of methods for running your programs in the debugger environment.

### *Changing modes*

| To put the debugger in... | Use this command... | See page... |
|---|---|---|
| Assembly mode | asm | 9-10 |
| Minimal mode | minimal | 9-22 |

### *Managing windows*

| To do this... | Use this command... | See page... |
|---|---|---|
| Reposition the active window | move | 9-23 |
| Resize the active window | size | 9-31 |
| Select the active window | win | 9-36 |
| Make the active window as large as possible | zoom | 9-37 |

### *Displaying and changing data*

| To do this... | Use this command... | See page... |
|---|---|---|
| Evaluate and display the result of a C expression | ? | 9-9 |
| Evaluate a C expression without displaying the results | eval | 9-17 |
| Display a different range of memory in the MEMORY window or display an additional MEMORY window | mem | 9-21 |
| Change the default format for displaying data values | setf | 9-30 |
| Continuously display the value of a variable, register, or memory location within the WATCH window | wa | 9-35 |
| Delete a data item from the WATCH window | wd | 9-36 |
| Delete all data items from the WATCH window and close the WATCH window | wr | 9-36 |

## *Performing system tasks*

| To do this... | Use this command... | See page... |
|---|---|---|
| Define your own command string | alias | 9-10 |
| Change the current working directory from within the debugger environment | cd/chdir | 9-13 |
| Clear all displayed information from the display area of the COMMAND window | cls | 9-13 |
| List the contents of the current directory or any other directory | dir | 9-15 |
| Record the information shown in the display area of the COMMAND window | dlog | 9-15 |
| Display a string to the COMMAND window while executing a batch file | echo | 9-16 |
| Conditionally execute debugger commands in a batch file | if/else/endif | 9-18 |
| Loop debugger commands in a batch file | loop/endloop | 9-19 |
| Exit the debugger | quit | 9-26 |
| Reset the target system | reset | 9-26 |
| Associate a beeping sound with the display of error messages | sound | 9-32 |
| Execute commands from a batch file | take | 9-33 |
| Delete an alias definition | unalias | 9-33 |
| Name additional directories that can be searched when you load source files | use | 9-34 |

## *Managing breakpoints*

| To do this... | Use this command... | See page... |
|---|---|---|
| Add a software breakpoint | ba | 9-11 |
| Delete a software breakpoint | bd | 9-11 |
| Display a list of all the software breakpoints that are set | bl | 9-11 |
| Reset (delete) all software breakpoints | br | 9-12 |

### *Displaying files and loading programs*

| To do this... | Use this command... | See page... |
|---|---|---|
| Display assembly language code at a specific point | addr | 9-10 |
| Display assembly language code at a specific address | dasm | 9-14 |
| Load an object file | load | 9-18 |
| Load only the object-code portion of an object file | reload | 9-26 |
| Load only the symbol-table portion of an object file | sload | 9-31 |

### *Customizing the screen*

| To do this... | Use this command... | See page... |
|---|---|---|
| Change the border style of any window | border | 9-12 |
| Change the screen colors, but do not update the screen immediately | color | 9-14 |
| Change the command-line prompt | prompt | 9-25 |
| Change the screen colors and update the screen immediately | scolor | 9-28 |
| Load and use a previously saved custom screen configuration | sconfig | 9-29 |
| Save a custom screen configuration | ssave | 9-32 |

### *Memory mapping*

| To do this... | Use this command... | See page... |
|---|---|---|
| Initialize a block of memory | fill | 9-17 |
| Add an address range to the memory map | ma | 9-20 |
| Enable or disable memory mapping | map | 9-20 |
| Delete an address range from the memory map | md | 9-21 |
| Display a list of the current memory map settings | ml | 9-22 |
| Reset the memory map (delete all ranges) | mr | 9-23 |
| Save a block of memory to a system file | ms | 9-24 |
| Connect an input file to the pin | pinc | 9-25 |
| Disconnect the input file from the pin | pind | 9-25 |
| List the pins that are connected to the input files | pinl | 9-25 |

## *Running programs*

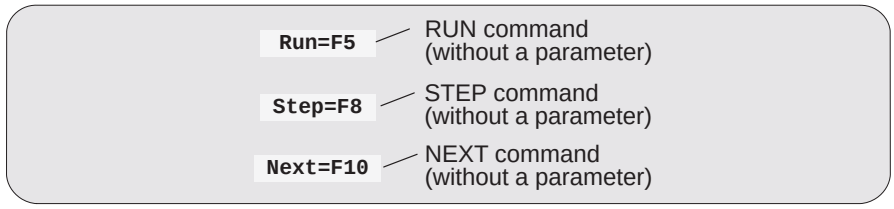
| To do this... | Use this command... | See page... |
|---|---|---|
| Run a program up to a certain point | go | 9-17 |
| Single-step through assembly language | next | 9-24 |
| Reset the target system | reset | 9-26 |
| Run a program | run | 9-27 |
| Single-step through assembly language | step | 9-32 |
| Execute commands from a batch file | take | 9-33 |

## 9.2 How the Menu Selections Correspond to Commands

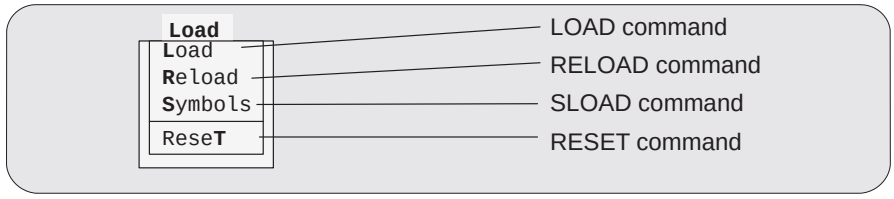
The following sample screens illustrate the relationship of the basic debugger commands to the menu bar and pulldown menus.

You can use the menus with or without a mouse. To access a menu from the keyboard, press the ALT key and the letter that is highlighted in the menu name. (For example, to display the Load menu, press ALT L.) Then, to make a selection from the menu, press the letter that is highlighted in the command you have selected. (For example, on the Load menu, to execute Reload, press R.) If you do not want to execute a command, press ESC to close the menu.

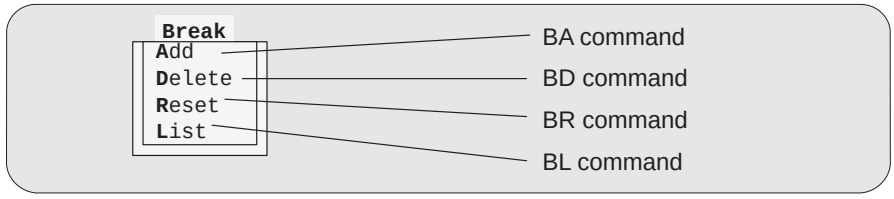
### *Program-execution commands*



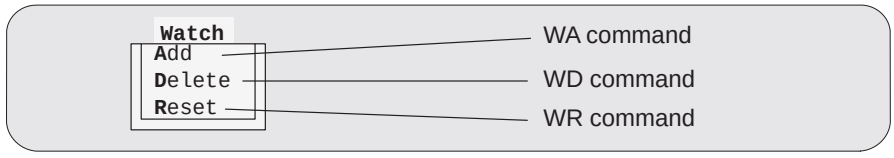

### *File/load commands*



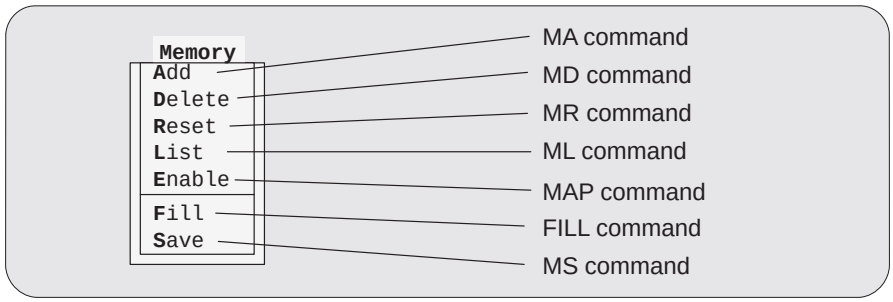

### *Breakpoint commands*



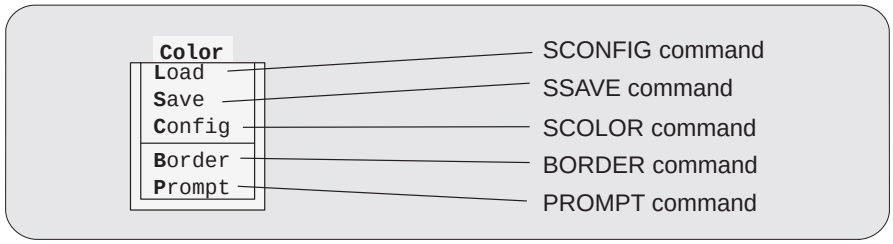

### *Watch commands*



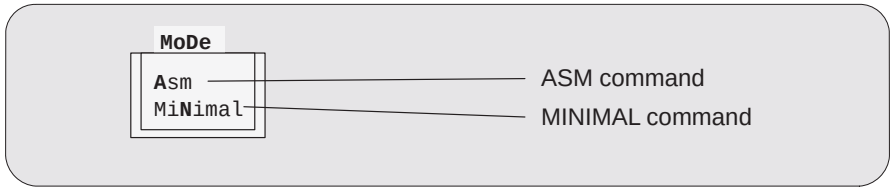
### *Memory commands*



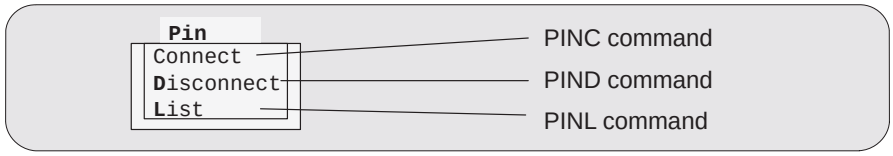

### *Screen-configuration commands*




### *Mode commands*




### *Interrupt-simulation commands*



## 9.3 Alphabetical Summary of Debugger Commands

Commands are not case sensitive; to emphasize this, command names are shown in both uppercase and lowercase throughout this book.

### ? — *Evaluate Expression*

**Syntax** **?** *expression* [*, display format*]

**Menu selection** none

**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.*

When you use the optional *display format* parameter, data is displayed in one of the following formats:

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

## addr — *Display Code at Specified Address*

**Syntax** **addr** *address*

**Menu selection** none

**Description** Use the ADDR command to display the disassembly at a specific point. 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.

## alias — *Define Custom Command String*

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

**Menu selection** none

**Description** You use the ALIAS command to associate one or more debugger commands with a single *alias name.*

You can include as many commands in the *command string* as you like, as long you separate them with semicolons and enclose the entire string of commands in quotation marks. You can also identify command parameters by a percent sign followed by a number (%1, %2, etc.). The total number of characters for an individual command (expanded to include parameter values) is limited to 132 (this restriction applies to the debugger version of the ALIAS command only).

Previously defined alias names can be included as part of the definition for a new alias.

To find the current definition of an alias, enter the ALIAS command with the *alias name* only. To see a list of all defined aliases, enter the ALIAS command with no parameters.

## asm — *Enter Assembly Mode*

**Syntax** **asm**

**Menu selection** Mo**D**e→**A**sm

**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** **B**reak→**A**dd

**Description** The BA command sets a software breakpoint at a specific *address*. This command does not require you to search through code to find the desired line. The *address* can be an absolute address, any C expression, or the name of an assembly language label.

Breakpoints can be set in program memory (RAM) only; the *address* parameter is treated as a program-memory address.

## bd — *Delete Software Breakpoint*

**Syntax** **bd** *address*

**Menu selection** **B**reak→ **D**elete

**Description** The BD command clears a software breakpoint at a specific *address*. The *address* can be an absolute address, any C expression, or the name of an assembly language label.

## bl — *List Software Breakpoints*

**Syntax** **bl**

**Menu selection** **B**reak→**L**ist

**Description** The BL command provides an easy way to get a complete listing of all the software breakpoints that are currently set in your program. It displays a table of breakpoints in the display area of the COMMAND window. BL lists all the breakpoints that are set in the order in which you set them.

## border — *Change Style of Window Border*

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

**Menu selection** **C**olor→**B**order

**Description** The BORDER command changes the border style of the active window, the inactive windows, and the border style of any window that you are resizing. The debugger supports nine border styles. Each parameter for the BORDER command must be one of the numbers that identify these styles:

| Index | Style |
|---|---|
| 0 | Double-lined box |
| 1 | Single-lined box |
| 2 | Solid 1/2-tone top, double-lined sides/bottom |
| 3 | Solid 1/4-tone top, double-lined sides/bottom |
| 4 | Solid box, thin border |
| 5 | Solid box, heavy sides, thin top/bottom |
| 6 | Solid box, heavy borders |
| 7 | Solid 1/2-tone box |
| 8 | Solid 1/4-tone box |

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

## br — *Reset Software Breakpoint*

**Syntax** **br**

**Menu selection** **B**reak→**R**eset

**Description** The BR command clears all software breakpoints that are set.

## cd, chdir — *Change Directory*

**Syntax**

**cd** [*directory name*]
**chdir** [*directory name*]

**Menu selection**

none

**Description**

This command is valid only when using the debugger with Windows. 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 *pathname*, the CD command displays the name of the current directory. This command can affect any other command whose parameter is a filename, such as the LOAD and TAKE commands, when used with the USE command. You can also use the CD command to change the current drive. For example,

```
cd c:
cd d:\source
cd c:\hethll
```

## cls — *Clear Screen*

**Syntax**

**cls**

**Menu selection**

none

**Description**

The CLS command clears all displayed information from the display area of the COMMAND window.

## color — *Change Screen Colors*

**Syntax** **color** *area name, attribute*$_1$ [,*attribute*$_2$ [,*attribute*$_3$ [,*attribute*$_4$] ] ]

**Menu selection** none

**Description** The COLOR command changes the color of specified areas of the debugger display. COLOR does not update the display; the changes take effect when another command, such as SCOLOR, updates the display. The *area name* parameter identifies the areas of the display that are affected. The *attributes* identify how the areas are affected. The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Valid values for the *attribute* parameters include:

| | | | |
|---|---|---|---|
| black | blue | green | cyan |
| red | magenta | yellow | white |
| bright | | blink | |

Valid values for the *area name* parameters include:

| | | | |
|---|---|---|---|
| menu_bar | menu_border | menu_entry | menu_cmd |
| menu_hilite | menu_hicmd | win_border | win_hiborder |
| win_resize | field_text | field_hilite | field_edit |
| field_label | field_error | cmd_prompt | cmd_input |
| cmd_cursor | cmd_echo | asm_data | asm_label |
| background | blanks | error_msg | file_text |
| file_brk | file_pc | | |

You do not have to type an entire *attribute* or *area name*; you need to type only enough letters to uniquely identify the attribute. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they are listed above (left to right, top to bottom).

## dasm — *Display Disassembly at Specific Address*

**Syntax** **dasm** *address*

**Menu selection** none

**Description** The DASM command displays code beginning at a specific point within the DISASSEMBLY window.

## dir — *List Directory Contents*

**Syntax** **dir** [*directory name*]

**Menu selection** none

**Description** This command is valid only when using the debugger with Windows. 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.

## dlog — *Record Display Window*

**Syntax** **dlog** *filename* [,{**a** | **w**}]
or
**dlog close**

**Menu selection** none

**Description** The DLOG command allows you to record the information displayed in the COMMAND window into a log file.

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

  **dlog** *filename*

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

- To end the recording session, enter:

  **dlog close** ↵

If necessary, you can write over existing log files or append additional information to existing files. The optional parameters of the DLOG command control how existing log files are used:

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

## echo — *Echo String to Display Area*

**Syntax** **echo** *string*

**Menu selection** none

**Description** The ECHO command displays a command *string* in the display area of the COMMAND window. You cannot use quotation marks around the *string,* and any leading blanks in your command string are removed when the ECHO command is executed. You can execute the ECHO command only in a batch file.

## else — *Execute Alternative Commands*

**Description** ELSE provides an alternative list of commands in the IF/ELSE/ENDIF command sequence. See page 9-18 for more information about these commands.

## endif — *Terminate Conditional Sequence*

**Description** ENDIF identifies the end of a conditional-execution command sequence begun with an IF command. See page 9-18 for more information about these commands.

## endloop — *Terminate Looping Sequence*

**Description** ENDLOOP identifies the end of the LOOP/ENDLOOP command sequence. See page 9-19 for more information about the LOOP/ENDLOOP commands.

## eval — *Evaluate Expression*

**Syntax**

**eval** *expression*
**e** *expression*

**Menu selection**

none

**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).

## fill — *Fill Memory*

**Syntax**

**fill** *address, length, data*

**Menu selection**

**M**emory→**F**ill

**Description**

The FILL command fills a block of memory with a specified value.

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

## go — *Run to Specified Address*

**Syntax**

**go** [*address*]

**Menu selection**

none

**Description**

The GO command executes code up to a specific point in your program. If you do not supply an *address*, then GO acts like a RUN command without an *expression* parameter—the program executes until it encounters a breakpoint or until you press ESC or the left mouse button.

## if/else/endif — *Conditionally Execute Debugger Commands*

**Syntax**

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

**Menu selection**

none

**Description**

These commands allow you to execute debugger commands conditionally in a batch file. If the *expression* is nonzero, the debugger executes the commands between the IF and the ELSE or ENDIF. The ELSE portion of the command sequence is optional.

The conditional commands work with the following provisions:

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

## load — *Load Executable Object File*

**Syntax**

**load** *object filename*

**Menu selection**

**L**oad→ **L**oad

**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*.hnc. The LOAD command clears the old symbol table and closes the WATCH window.

## loop/endloop — *Loop Through Debugger Commands*

**Syntax**

**loop** *expression*
*debugger commands*
**endloop**

**Menu selection**

none

**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, length, type*

**Menu selection** **M**emory→**A**dd

**Description** The MA command identifies valid ranges of target memory. A new memory map must not overlap an existing entry; if you define a range that overlaps an existing range, the debugger ignores the new range.

- The *address* parameter defines the starting address of a range in memory. This parameter can be an absolute address, any C expression, or an assembly language label.
- The *length* parameter defines the length of the range. This parameter can be any C expression.
- The *type* parameter identifies the read/write characteristics of the memory range. The *type* must be one of these keywords:

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

## map — *Enable Memory Mapping*

**Syntax** **map** {**on** | **off**}

**Menu selection** **M**emory→**E**nable

**Description** The MAP command enables or disables memory mapping. In some instances, you may want to explicitly enable or disable memory. Disabling memory mapping can cause bus-fault problems in the target system because the debugger may attempt to access nonexistent memory.

## md — *Delete Block From Memory Map*

**Syntax** **md** *address*

**Menu selection** **M**emory→**D**elete

**Description** The MD command deletes a range of memory from the debugger's memory map.

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

```
Specified map not found
```

## mem — *Modify MEMORY Window Display*

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

**Menu selection** none

**Description** The MEM command identifies a new starting address for the block of memory displayed in the MEMORY window. The optional *window name* parameter opens an additional MEMORY window, allowing you to view a separate block of memory. The debugger displays the contents of memory at *expression* in the first data position in the MEMORY window. The end of the range is defined by the size of the window. The *expression* can be an absolute address, a symbolic address, or any C expression.

When you use the optional *display format* parameter, memory is displayed in one of the following formats:

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

## minimal — *Enter Minimal Mode*

**Syntax** **minimal**

**Menu selection** Mo**D**e→Mi**N**imal

**Description** The MINIMAL command changes from the current debugging mode to minimal mode. If you are already in minimal mode, the MINIMAL command has no effect.

## ml — *List Memory Map*

**Syntax** **ml**

**Menu selection** **M**emory→**L**ist

**Description** The ML command lists the memory ranges that are defined for the debugger's memory map. The ML command lists the starting address, ending address, and read/write characteristics of each defined memory range.

## move — *Move Active Window*

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

**Menu selection** none

**Description** The MOVE command moves the active window to the specified XY position. If you choose, you can resize the window while you move it (see the SIZE command for valid *width* and *length* values). You can use the MOVE command in one of two ways:

- By supplying a specific *X position* and *Y position*
- By omitting the *X position* and *Y position* parameters and using function keys to interactively move the window

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

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

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

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

When you are finished using the arrow keys, you must press ESC or ↵.

## mr — *Reset Memory Map*

**Syntax** **mr**

**Menu selection** **M**emory→**R**eset

**Description** The MR command resets the debugger's memory map by deleting all defined memory ranges from the map.

## ms — *Save Memory Block to File*

**Syntax** **ms** *address, length, filename*

**Menu selection** **M**emory→**S**ave

**Description** The MS command saves the values in a block of memory to a system file; files are saved in COFF format.

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

## next — *Single-Step, Next Statement*

**Syntax** **next** [*expression*]

**Menu selection** Next=**F10** (in disassembly)

**Description** The NEXT command is similar to the STEP command. The debugger executes one assembly language statement at a time. NEXT always steps to the next consecutive statement, ignoring calls to functions.

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page 5-11, discusses this in detail).

## pause — *Pause Execution*

**Syntax** **pause**

**Menu selection** none

**Description** The PAUSE command allows you to pause the debugger while running a batch file. Pausing is especially helpful in debugging the commands in a batch file.

When the debugger reads this command in a batch file, the debugger stops execution and displays the following message:

```
<< pause – type return >>
```

To continue processing, press ↵.

## pinc — *Connect Pin*

**Syntax** **pinc** *pinname, filename*

**Menu selection** **P**in→**C**onnect

**Description** The PINC command connects an input file to a pin.

- ❏ The *pinname* parameter identifies the pin name and must be one of the following pins (AINC, CC0–CC31, CPU, INT, or MODE).
- ❏ The *filename* parameter is the name of your input file.

## pind — *Disconnect Pin*

**Syntax** **pind** *pinname*

**Menu selection** **P**in→**D**isconnect

**Description** The PIND command disconnects an input file from a pin. The *pinname* parameter identifies the pin name and must be one of the following pins (AINC, CC0–CC31, CPU, INT, or MODE).

## pinl — *List Pin*

**Syntax** **pinl**

**Menu selection** **P**in→**L**ist

**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.

## prompt — *Change Command-Line Prompt*

**Syntax** **prompt** *new prompt*

**Menu selection** **C**olor→**P**rompt

**Description** The PROMPT command changes the command-line prompt. The *new prompt* can be any string of characters (note that a semicolon or comma ends the string).

## quit — *Exit Debugger*

**Syntax** **quit**

**Menu selection** none

**Description** The QUIT command exits the debugger and returns to the operating system.

## reload — *Reload Object Code*

**Syntax** **reload** [*object filename*]

**Menu selection** **L**oad→**R**eload

**Description** The RELOAD command loads only an object file *without* loading its associated symbol table. This is useful for reloading a program when target memory has been corrupted. If you enter the RELOAD command without specifying a filename, the debugger reloads the file that you loaded last.

## reset — *Reset Target System*

**Syntax** **reset**

**Menu selection** **L**oad→Rese**T**

**Description** The RESET command resets the simulator and reloads the monitor. This is a *software* reset.

If you execute the RESET command, the simulator simulates the TMS470 HET and peripheral-reset operation, putting the processor in a known state.

## run — *Run Code*

**Syntax** **run** [*expression*]

**Menu selection** Run=**F5**

**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 press the left mouse button or press ESC.
- If you supply a logical or relational *expression*, the run becomes conditional (*Running code conditionally*, page 5-11, discusses this in detail).
- If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger executes *count* instructions, halts, and updates the display.

## safehalt — *Toggle Safehalt Mode*

**Syntax** **safehalt** {**on** | **off**}

**Menu selection** none

**Description** The SAFEHALT command places the debugger in safehalt mode. When safehalt mode is off (the default), you can halt a running target device either by pressing ESC or by clicking a mouse button. When safehalt mode is on, you can halt a running target device only by pressing ESC; mouse clicks are ignored.

## scolor — *Change Screen Colors*

**Syntax**

**scolor** *area name*, *attribute*$_1$ [, *attribute*$_2$ [, *attribute*$_3$ [, *attribute*$_4$ ] ] ]

**Menu selection**

**C**olor→**C**onfig

**Description**

The SCOLOR command changes the color of specified areas of the debugger display and updates the display immediately. The *area name* parameter identifies the area of the display that is affected. The *attributes* identify how the area is affected. The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Valid values for the *attribute* parameters include:

| | | | |
|---|---|---|---|
| black | blue | green | cyan |
| red | magenta | yellow | white |
| bright | | blink | |

Valid values for the *area name* parameters include:

| | | | |
|---|---|---|---|
| menu_bar | menu_border | menu_entry | menu_cmd |
| menu_hilite | menu_hicmd | win_border | win_hiborder |
| win_resize | field_text | field_hilite | field_edit |
| field_label | field_error | cmd_prompt | cmd_input |
| cmd_cursor | cmd_echo | asm_data | asm_label |
| background | blanks | error_msg | file_text |
| file_brk | file_pc | file_pc_brk | |

You do not have to type an entire *attribute* or *area name*; you need to type only enough letters to uniquely identify the attribute. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they are listed above (left to right, top to bottom).

## sconfig — *Load Screen Configuration*

**Syntax** **sconfig** [*filename*]

**Menu selection** **C**olor→**L**oad

**Description** The SCONFIG command restores the display to a specified configuration. This restores the screen colors, window positions, window sizes, and border styles that were saved with the SSAVE command into *filename*. If you do not supply a *filename*, the debugger looks for init.clr. The debugger searches for the specified file in the current directory and then in directories named with the D_DIR environment variable.

When you use SCONFIG to restore a configuration that includes multiple WATCH or MEMORY windows, the additional windows are not launched automatically. However, when you launch an additional window using the same name as before you saved the configuration, the window is placed in the correct location.

## setf — *Set Default Data-Display Format*

**Syntax**

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

**Menu selection**

none

**Description**

The SETF command changes the display format for a specific data type. If you enter SETF with no parameters, the debugger lists the current display format for each data type.

❏ The *data type* parameter can be any of the following C data types:

| | | | | |
|---|---|---|---|---|
| char | short | uint | ulong | double |
| uchar | int | long | float | ptr |

❏ The *display format* parameter can be any of the following characters:

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

Only a subset of the display formats can be used for each data type. Listed below are the valid combinations of data types and display formats.

| | Valid Display Formats | | | | | | | | | | Valid Display Formats | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Data Type** | **c** | **d** | **o** | **x** | **e** | **f** | **p** | **s** | **u** | **Data Type** | **c** | **d** | **o** | **x** | **e** | **f** | **p** | **s** | **u** |
| char (c) | √ | √ | √ | √ | | | | | √ | long (d) | √ | √ | √ | √ | | | | | √ |
| uchar (d) | √ | √ | √ | √ | | | | | √ | ulong (d) | √ | √ | √ | √ | | | | | √ |
| short (d) | √ | √ | √ | √ | | | | | √ | float (e) | | | √ | √ | √ | √ | | | |
| int (d) | √ | √ | √ | √ | | | | | √ | double (e) | | | √ | √ | √ | √ | | | |
| uint (d) | √ | √ | √ | √ | | | | | √ | ptr (p) | | | √ | √ | | | √ | √ | |

To return all data types to their default display format, enter:

```
setf * ↵
```

## size — *Size Active Window*

**Syntax** **size** [*width*, *length* ]

**Menu selection** none

**Description** The SIZE command changes the size of the active window. You can use the SIZE command in one of two ways:

- By supplying a specific *width* and *length* or
- By omitting the *width* and *length* parameters and using function keys to interactively resize the window.

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

If a window is in the middle of the display, you cannot size it to the maximum height and width—you can size it only to the right and bottom screen borders. The easiest way to make a window as large as possible is to zoom it; see *Zooming a window* on page 2-20.

If you enter the SIZE command without *width* and *length* parameters, you can use arrow keys to size the window.

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

When you are finished using the arrow keys, you must press ESC or ↵.

## sload — *Load Symbol Table*

**Syntax** **sload** *object filename*

**Menu selection** **L**oad→**S**ymbols

**Description** The SLOAD command loads the symbol table of the specified object file. SLOAD is useful in a debugging environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). SLOAD clears the existing symbol table before loading the new one but does not modify memory or set the program entry point. Note that SLOAD closes the WATCH window.

## sound — *Enable Error Beeping*

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

**Menu selection** none

**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.

## ssave — *Save Screen Configuration*

**Syntax** **ssave** [*filename*]

**Menu selection** **C**olor→**S**ave

**Description** The SSAVE command saves the current screen configuration to a file. This saves the screen colors, window positions, window sizes, and border styles. SSAVE also saves the location of multiple WATCH and MEMORY windows. The *filename* parameter names the new screen configuration file. You can include path information (including relative pathnames); if you 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.

## step — *Single-Step*

**Syntax** **step** [*expression*]

**Menu selection** Step=**F8** (in disassembly)

**Description** The STEP command single-steps through assembly language code. The debugger executes one assembly language statement at a time.

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page 5-11, discusses this in detail).

## take — *Execute Batch File*

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

**Menu selection** none

**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. For the debugger, you can change this behavior:

- If you do not use the *suppress echo flag* parameter, or if you use it but supply a nonzero value, then the debugger behaves in the default manner.
- If you would like to suppress the echoing and updating, use the value 0 for the *suppress echo flag* parameter.

## unalias — *Delete Alias Definition*

**Syntax** **unalias** *alias name*
**unalias** *****

**Menu selection** none

**Description** The UNALIAS command deletes defined aliases.

- To delete a *single alias*, enter the UNALIAS command with an alias name. For example, to delete an alias named NEWMAP, enter:

  ```
  unalias NEWMAP ↵
  ```

- To delete *all aliases*, enter an asterisk instead of an alias name:

  ```
  unalias * ↵
  ```

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

## use — *Use New Directory*

**Syntax** **use** [*directory name*]

**Menu selection** none

**Description** The USE command allows you to name an additional directory that the debugger can search when looking for source files. You can specify only one directory at a time.

If you enter the USE command without specifying a directory name, the debugger lists all of the current directories.

## version — *Display the Current Debugger Version*

**Syntax** **version**

**Menu selection** none

**Description** The VERSION command displays the debugger's copyright date and the current version number of the debugger, silicon, etc.

## wa — *Add Item to WATCH Window*

**Syntax**

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

**Menu selection**

**W**atch→**A**dd

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

If you want to use a *display format* parameter without a *label* parameter, just insert an extra comma. For example:

```
wa PFR,,d ↵
```

You can open additional WATCH windows by using the *window name* parameter. When you open an additional WATCH window, the debugger appends the *window name* to the WATCH window label. You can create as many WATCH windows as you need.

If you omit the *window name* parameter, the debugger displays the expression in the default WATCH window (labeled WATCH).

## wd — *Delete Item From WATCH Window*

**Syntax** **wd** *index number* [, *window name*]

**Menu selection** **W**atch→**D**elete

**Description** The WD command deletes a specific item from the WATCH window. The WD command's *index number* parameter must correspond to one of the watch indexes listed in the WATCH window. The optional *window name* parameter is used to specify a particular WATCH window.

## win — *Select Active Window*

**Syntax** **win** *WINDOW NAME*

**Menu selection** none

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

If several windows of the same type are visible on the screen, do not use the WIN command to select one of them. If you supply an ambiguous name, 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** **W**atch→**R**eset

**Description** The WR command deletes all items from a WATCH window and closes the window.

- To close the default WATCH window, enter:

  **wr** ↵

- To close one of the additional WATCH windows, use this syntax:

  **wr** *windowname*

- To close all WATCH windows, enter:

  **wr *** ↵

## zoom

*Zoom Active Window*

**Syntax** **zoom**

**Menu selection** none

**Description** The ZOOM command makes the active window as large as possible. To unzoom a window, enter the ZOOM command a second time; this returns the window to its prezoom size and position.

## 9.4 Summary of Special Keys

The debugger provides function key, cursor key, and command key sequences for performing a variety of actions:

- ❑ Editing text on the command line
- ❑ Using the command history
- ❑ Switching modes
- ❑ Halting or escaping from an action
- ❑ Displaying the pulldown menus
- ❑ Running code
- ❑ Selecting or closing a window
- ❑ Moving or sizing a window
- ❑ Scrolling through a window's contents
- ❑ Editing data or selecting the active field

### *Editing text on the command line*

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

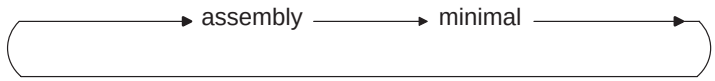
† You can use the arrow keys only when the COMMAND window is selected.

### *Using the command history*

| To... | Press... |
|---|---|
| Repeat the last command that you entered | F2 |
| Move backward, one command at a time, through the command history | TAB |
| Move forward, one command at a time, through the command history | SHIFT TAB |

### *Switching modes*

| To... | Press... |
|---|---|
| Switch debugging modes in this order: | F3 |




### *Halting or escaping from an action*

The escape key acts as an end or undo key in several situations.

| To... | Press... |
|---|---|
| Halt program execution | ESC |
| Close a pulldown menu | |
| Undo an edit of the active field in a data-display window (pressing this key leaves the field unchanged) | |
| Halt the display of a long list of data in the display area of the COMMAND window | |

## *Displaying pulldown menus*

| To... | Press... |
|---|---|
| Display the Load menu | ALT L |
| Display the Break menu | ALT B |
| Display the Watch menu | ALT W |
| Display the Memory menu | ALT M |
| Display the Color menu | ALT C |
| Display the MoDe menu | ALT D |
| Display the Pin menu | ALT P |
| Display an adjacent menu | ← or → |
| Execute any of the choices from a displayed pulldown menu | The highlighted letter corresponding to your choice |

## *Running code*

| To... | Press... |
|---|---|
| Run code from the HETADDR (equivalent to the RUN command without an *expression* parameter) | F5 |
| Single-step code from the HETADDR (equivalent to the STEP command without an *expression* parameter) | F8 |
| Single-step code from the HETADDR (equivalent to the NEXT command without an *expression* parameter) | F10 |

### *Selecting or closing a window*

| To... | Press... |
|---|---|
| Select the active window (pressing this key makes each window active in turn; stop pressing the key when the desired window becomes active) | F6 |
| Close the WATCH or additional MEMORY window (the window must be active before you can close it) | F4 |

### *Moving or sizing a window*

You can use the arrow keys to interactively move a window after entering the MOVE or SIZE command without parameters.

| To... | Press... |
|---|---|
| Move the window down one line<br>Make the window one line longer | ↓ |
| Move the window up one line<br>Make the window one line shorter | ↑ |
| Move the window left one character position<br>Make the window one character narrower | ← |
| Move the window right one character position<br>Make the window one character wider | → |

### *Scrolling through a window's contents*

These descriptions and instructions for scrolling apply to the active window.

| To... | Press... |
|---|---|
| Scroll up through the window contents, one window length at a time | PAGE UP |
| Scroll down through the window contents, one window length at a time | PAGE DOWN |
| Move the field cursor up, one line at a time | ↑ |
| Move the field cursor down, one line at a time | ↓ |
| Move the field cursor left one field; at the first field on a line, wrap back to the last fully displayed field on the previous line | ← |
| Move the field cursor right one field; at the last field on a line, wrap around to the first field on the next line | → |

### *Editing data or selecting the active field*

The F9 function key makes the current field (the field that the cursor is pointing to) active. This has various effects, depending on the field.

| To... | Press... |
|---|---|
| Set or clear a breakpoint | F9 |
| Edit the contents of the current field in any data-display window | F9 |

Chapter 10

# Basic Information About C Expressions

Many of the debugger commands take C expressions as parameters. This allows the debugger to have a relatively small, yet powerful, instruction set. Because C expressions can have side effects—that is, the evaluation of some types of expressions can affect existing values—you can use the same command to display or to change a value. This reduces the number of commands in the command set.

This chapter contains basic information that you need to use C expressions as debugger command parameters.

**Topic** **Page**

## 10.1 C Expressions for Assembly Language Programmers

It is not necessary for you to be an experienced C programmer in order to use the debugger. However, in order to use the debugger's full capabilities, you should be familiar with the rules governing C expressions. You should obtain a copy of *The C Programming Language* (first or second edition) by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, 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*) | typecast | | |

❑ **Relational and logical operators**

| | | | |
|---|---|---|---|
| > | greater than | >= | greater than or equal to |
| < | less than | <= | less than or equal to |
| == | is equal to | != | is not equal to |
| && | logical AND | \|\| | logical OR |
| ! | logical NOT (unary) | | |

### ❑ Increment and decrement operators

| | | | |
|---|---|---|---|
| ++ | increment | – – | decrement |

These unary operators can precede or follow a symbol. When the operator precedes a symbol, the symbol value is incremented/decremented before it is used in the expression; when the operator follows a symbol, the symbol value is incremented/decremented after it is used in the expression. Because these operators affect the symbol's final value, they have side effects.

### ❑ Bitwise operators

| | | | |
|---|---|---|---|
| & | bitwise AND | \| | bitwise OR |
| ^ | bitwise exclusive-OR | << | left shift |
| >> | right shift | ~ | 1s complement (unary) |

### ❑ Assignment operators

| | | | |
|---|---|---|---|
| = | assignment | += | assignment with addition |
| –= | assignment with subtraction | /= | assignment with division |
| %= | assignment with modulo | &= | assignment with bitwise AND |
| ^= | assignment with bitwise XOR | \|= | assignment with bitwise OR |
| <<= | assignment with left shift | >>= | assignment with right shift |
| *= | assignment with multiplication | | |

These operators support a shorthand version of the familiar binary expressions; for example, X = X + Y can be written in C as X += Y. Because these operators affect a symbol's final value, they have side effects.

## 10.2 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*

These 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).
- ❏ 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 *****s in a cast.



### *Additional features*

These are the additional features of the debugger's expression analysis.

- ❏ 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 HET's auxiliary registers are treated as integers and/or pointers.
- ❏ Void expressions are legal (treated like integers).
- ❏ The specification of variables and functions can be qualified with context information. Local variables (including local statics) can be referenced with the expression form:

  *function name***.***local name*

This expression format is useful for examining the automatic variables of a function that is not currently being executed. Unless the variable is static, however, the function must be somewhere in the current call stack. Note that if you want to see local variables from the currently executing function, you need not use this form; you can simply specify the variable name (just as in your C source).

File-scoped variables (such as statics or functions) can be referenced with the following expression form:

*filename***.***function name*
or *filename***.***variable name*

This expression format is useful for accessing a file-scoped static variable (or function) that may share its name with variables in other files.

In this expression, *filename* **does not include** the file extension; the debugger searches the object symbol table for any source filename that matches the input name, disregarding any extension. Thus, if the variable *ABC* is in file source.c, you can specify it as source.ABC.

These expression forms can be combined into an expression of the form:

*filename***.***function name***.***variable name*

- Any integral or void expression can be treated as a pointer and used with the indirection operator (*). Here are several examples of valid use of a pointer in an expression:

```
*123
*AR5
*(AR2 + 123)
*(I*J)
```

By default, the values are treated as integers (that is, these expressions point to integer values).

- Any expression can be typecast to a pointer to a specific type (overriding the default of pointing to an integer, as described above).

**Hint:** You can use casting with the WA command 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.

The first expression differs from the expression:

```
(float)*10
```

In this case, the debugger fetches an integer from address 10 and converts the integer to a floating-point value.

You can also typecast to user-defined types such as structures. For example, in the expression:

```
((struct STR *)10)–>field
```

the debugger treats memory location 10 as a pointer to a structure of type STR (assuming that a structure is at address 10) and accesses a field from that structure.

Appendix A

# What the Debugger Does During Invocation

In some circumstances, you may find it helpful to know the steps that the debugger performs during the invocation process. (For more information about setting the environment variables mentioned below, see the CD-ROM insert.)

The debugger:

1) Reads options from the command line

2) Reads any information specified with the D_OPTIONS environment variable

3) Reads information from the D_DIR and D_SRC environment variables

4) Looks for the init.clr screen configuration file

   (The debugger searches for the screen configuration file in directories named with D_DIR.)

5) Initializes the debugger screen and windows but initially displays only the COMMAND window

6) Finds the batch file that defines your memory map by searching in directories named with D_DIR. The debugger expects this file to set up the memory map and follows these steps to look for the batch file:

   - ❏ When you invoke the debugger, it checks to see if you have used the –t debugger option. If it finds the –t option, the debugger reads and executes the specified file.
   - ❏ If you have not used the –t option, the debugger looks for the default initialization batch file.

   If the debugger finds the file, it reads and executes the file.

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

7) Loads any object filenames specified with D_OPTIONS or specified on the command line during invocation

8) Determines the initial mode (assembly or minimal) and displays the appropriate windows on the screen

At this point, the debugger is ready to process any commands that you enter.

Appendix B

# Debugger Messages

This appendix contains an alphabetical listing of the progress and error messages that the debugger might display in the display area of the COMMAND window. Each message contains both a description of the situation that causes the message and an action to take if the message indicates a problem or error.

**Topic** **Page**

## B.1 Associating Sound With Error Messages

You can associate a beeping sound with the display of error messages. To do this, use the SOUND command. The format for this command is:

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

By default, no beep is associated with error messages (SOUND OFF). The beep is helpful if the COMMAND window is hidden behind other windows.

## B.2 Alphabetical Summary of Debugger Messages

# Symbol

### ']' expected

*Description* This is an expression error—it means that the parameter contained an opening bracket symbol "**[**" but did not contain a closing bracket symbol "**]**".

*Action* See section B.3, *Additional Instructions for Expression Errors*, page B-14.

### ')' 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 B.3, *Additional Instructions for Expression Errors*, page B-14.

# A

### Aborted by user

*Description* The debugger halted a long COMMAND display listing (from WHATIS, DIR, ML, or BL) because you pressed the ESC key.

*Action* None required; this is normal debugger behavior.

# B

### Breakpoint table full

*Description* The maximum limit of 200 breakpoints is already set, and there was an attempt to set another. The limit includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.

*Action* Enter a BL command to see where breakpoints are set in your program. Use the BR command to delete all software breakpoints, or use the BD command to delete individual software breakpoints.

# C

### Cannot allocate host memory

*Description* This is a fatal error—it means that the debugger is running out of memory.

*Action* You might try invoking the debugger with the –v option so that fewer symbols may be loaded. Or you might want to relink your program and link in fewer modules at a time.

### Cannot allocate system memory

*Description* This is a fatal error—it means that the debugger is running out of memory.

*Action* You might try invoking the debugger with the –v option so that fewer symbols may be loaded. Or you might want to relink your program and link in fewer modules at a time.

### Cannot connect pin

*Description* The debugger was unable to execute the pinc command issued.

*Action* Make sure that the pin you are trying to connect to is legal. Specifically, you cannot connect a file to a cc pin that is configured as output. (See section 4.7, *Simulating external signals*, page 4-10.)

### Cannot edit field

*Description* Expressions that are displayed in the WATCH window cannot be edited.

*Action* If you attempted to edit an expression in the WATCH window, you may have actually wanted to change the value of a symbol or register used in the expression. Use the ? or EVAL command to edit the actual symbol or register. The expression value will be updated automatically.

### Cannot find/open initialization file

*Description* The debugger 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 CD-ROM. 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 instructions for setting the D_DIR environment variable on the CD-ROM insert.

### 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 or initdb.bat file; then invoke the debugger again.

### Cannot map into reserved memory: ?

*Description* The debugger tried to access unconfigured/reserved/nonexistent memory.

*Action* Remap the reserved memory accesses.

### Cannot open config file

*Description* The SCONFIG command cannot find the screen-customization file that you specified.

*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.

### Cannot open new window

*Description* The maximum limit of 127 windows is already open. The last request to open a window exceeded this limit.

*Action* Close any unnecessary windows. Windows that can be closed include WATCH and additional MEMORY windows. To close any of these windows, make the desired window active and press F4. To close the WATCH window, enter WD.

### 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.

### 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 or initdb.bat file, then invoke the debugger again.

### Cannot reset the processor

*Description* This is a fatal error—for some reason, pressing ESC did not halt program execution.

*Action* Exit the debugger. Invoke the autoexec or initdb.bat file, then invoke the debugger again.

### Cannot set/verify breakpoint at *address*

*Description* Either you attempted to set a breakpoint in read-only or protected memory, or there are hardware problems with the target system. This may also happen when you enable or disable on-chip memory while using breakpoints.

*Action* Check your memory map.

### Cannot take address of register

*Description* This is an expression error. C does not allow you to take the address of a register.

*Action* See section B.3, *Additional Instructions for Expression Errors*, page B-14.

### Command "*cmd*" not found

*Description* The debugger did not recognize the command that you typed.

*Action* Reenter the correct command. Refer to Chapter 9, *Summary of Commands and Special Keys*.

### Conflicting map range

*Description* A block of memory specified with the MA command overlaps an existing memory map entry. Blocks cannot overlap.

*Action* Use the ML command to list the existing memory map; this will help you find the existing block that the new block would overlap. If the existing block is not necessary, delete it with the MD command and reenter the MA command. If the existing block is necessary, reenter the MA command with parameters that will not overlap the existing block.

## E

### Error in expression

*Description* This is an expression error.

*Action* See section B.3, *Additional Instructions for Expression Errors*, page B-14.

## F

### 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.

### Float not allowed

*Description* This is an expression error—a floating-point value was used incorrectly.

*Action* See section B.3, *Additional Instructions for Expression Errors*, page B-14.

# 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 B.3, *Additional Instructions for Expression Errors*, page B-14.

### Illegal control transfer instruction

*Description* The instruction following a delayed branch/call instruction was modifying the current address register (CAR).

*Action* Modify your source code.

### Illegal left hand side of assignment

*Description* This is an expression error—the left-hand side of an assignment expression does not meet C language assignment rules.

*Action* See section B.3, *Additional Instructions for Expression Errors*, page B-14.

### Illegal memory access

*Description* Your program tried to access unmapped memory.

*Action* Modify your source code.

### Illegal opcode

*Description* An invalid HET instruction was encountered.

*Action* Modify your source code.

### Illegal operand of &

*Description* This is an expression error—the expression attempts to take the address of an item that does not have an address.

*Action* See section B.3, *Additional Instructions for Expression Errors*, page B-14.

### Illegal pointer math

*Description* This is an expression error—some types of pointer math are not valid in C expressions.

*Action* See section B.3, *Additional Instructions for Expression Errors*, page B-14.

### Illegal pointer subtraction

*Description* This is an expression error—the expression attempts to use pointers in a way that is not valid.

*Action* See section B.3, *Additional Instructions for Expression Errors*, page B-14.

### Illegal syntax for this type of pin

*Description* The syntax of the connected file is not legal for this pin.

*Action* Varify that you have used the correct syntax of the input file for that pin.

### Illegal use of void expression

*Description* This is an expression error—the expression parameter does not meet the C language rules.

*Action* See section B.3, *Additional Instructions for Expression Errors*, page B-14.

### Input number too big at line number

*Description* The number used in the specified line is larger than the allowed maximum.

*Action* Try a smaller value.

### Integer not allowed

*Description* This is an expression error—the command did not accept an integer as a parameter.

*Action* See section B.3, *Additional Instructions for Expression Errors*, page B-14.

### 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 9, *Summary of Commands and Special Keys*.

### Invalid attribute name

*Description* The COLOR and SCOLOR commands accept a specific set of area names for their first parameter. The parameter entered did not match one of the valid attributes.

*Action* Reenter the COLOR or SCOLOR command with a valid area name parameter. Valid area names are listed in Table 8–2 (page 8-3).

### Invalid color name

*Description* The COLOR and SCOLOR commands accept a specific set of color attributes as parameters. The parameter entered did not match one of the valid attributes.

*Action* Reenter the COLOR or SCOLOR command with a valid color parameter. Valid color attributes are listed in Table 8–1 (page 8-2).

### Invalid memory attribute

*Description* The third parameter of the MA command specifies the type, or attribute, of the block of memory that MA adds to the memory map. The parameter entered did not match one of the valid attributes.

*Action* Reenter the MA command. Use one of the following valid parameters to identify the memory type:

| | |
|---|---|
| R, ROM | (read-only memory) |
| W, WOM | (write-only memory) |
| R\|W, RAM | (read/write memory) |
| PROTECT | (no-access memory) |
| OUTPORT, P\|W | (output port) |
| INPORT, P\|R | (input port) |
| IOPORT, P\|R\|W | (input/output port) |

### Invalid object file

*Description* Either the file specified with the LOAD, SLOAD, or RELOAD command is not an object file that the debugger can load, or it has been corrupted.

*Action* Be sure that you are loading an actual object file. If the file you attempted to load was a valid executable object file, then it was probably corrupted; recompile and assemble the file.

### Invalid watch delete

*Description* The debugger cannot delete the parameter supplied with the WD command. Usually, this is because the watch index does not exist or because a symbol name was typed instead of a watch index.

*Action* Reenter the WD command. Be sure to specify the watch index that matches the item you would like to delete (this is the number in the left column of the WATCH window). Remember, you cannot delete items symbolically—you must delete them by number.

### Invalid window position

*Description* The debugger cannot move the active window to the XY position specified with the MOVE command. Either the XY parameters are not within the screen limits, or the active window may be too large to move to the desired position.

*Action* You can use the mouse to move the window.

- ❏ If you do not have a mouse, enter the MOVE command without parameters; then use the arrow keys to move the window. When you are finished, you *must* press ESC or ↵.
- ❏ If you prefer to use the MOVE command with parameters, the minimum XY position is 0,1; the maximum position depends on which screen size you are using.

### Invalid window size

*Description* The width and length specified with the SIZE or MOVE command may be too large or too small. If a valid width and length were specified, then the active window is already at the far right or bottom of the screen and so cannot be made larger.

*Action* You can use the mouse to size the window.

- ❏ If you do not have a mouse, enter the SIZE command without parameters; then use the arrow keys to move the window. When you are finished, you *must* press ESC or ↵.
- ❏ If you prefer to use the SIZE command with parameters, the minimum size is 4 by 3; the maximum size depends on which screen size you are using.

## L

### Large cycle number; maximum is 0xffffff

*Description* The cycle number used in this line is too big

*Action* Make sure the cycle number is within the allowed range.

### Load aborted

*Description* This message always follows another message.

*Action* Refer to the message that preceded *Load aborted*.

### Lval required

*Description* This is an expression error—an assignment expression was entered that requires a legal left-hand side.

*Action* See section B.3, *Additional Instructions for Expression Errors*, page B-14.

## M

### 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* Too many levels of nesting in the usage of the repeat.

*Action* You will have to reduce the number of nested repeats.

### Non-repeatable instruction

*Description* The instruction following the RPT instruction is not a repeatable instruction.

*Action* Modify your code.

## P

### Pointer not allowed

*Description* This is an expression error.

*Action* See section B.3, *Additional Instructions for Expression Errors*, page B-14.

### Processor is already running

*Description* One of the RUN commands was entered while the debugger was running free from the target system.

*Action* Enter the HALT command to stop the free run, then reenter the desired RUN command.

## S

### Specified map not found

*Description* The MD command was entered with an address or block that is not in the memory map.

*Action* Use the ML command to verify the current memory map. When using MD, you can specify only the first address of a defined block.

### Syntax error at line number

*Description* The parser came across an illegal syntax in the file it connected to.

*Action* Check the file content with the syntax description in section 4.7, *Simulating External Signals*, page 4-10.

T

### Take file stack too deep

*Description* Batch files can be nested up to ten levels deep. Batch files can call other batch files, which can call other batch files, and so on. Apparently, the batch file that you are TAKEing calls batch files that are nested more than ten levels deep.

*Action* Edit the batch file that caused the error. Instead of calling another batch file from within the offending file, you may want to copy the contents of the second file into the first. This will remove a level of nesting.

### Too many breakpoints

*Description* The maximum limit of 200 breakpoints is already set, and there was an attempt to set another. The limit includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.

*Action* Enter a BL command to see where breakpoints are set in your program. Use the BR command to delete all software breakpoints or use the BD command to delete individual software breakpoints.

### Too many paths

*Description* More than 20 paths have been specified cumulatively with the USE command, D_SRC environment variable, and –i debugger option.

*Action* 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

### User halt

*Description* The debugger halted program execution because you pressed the ESC key.

*Action* None required; this is normal debugger behavior.

## W

### Window not found

*Description* The parameter supplied for the WIN command is not a valid window name.

*Action* Reenter the WIN command. Remember that window names must be typed in uppercase letters. Here are the valid window names; the bold letters show the smallest acceptable abbreviations:

| | | |
|---|---|---|
| **CP**U | **CO**MMAND | **DISA**SSEMBLY |
| **M**EMORY | **W**ATCH | |

## B.3 Additional Instructions for Expression Errors

Whenever you receive an expression error, you should reenter the command and edit the expression so that it follows the C language expression rules. If necessary, refer to a C language manual such as *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

Appendix C

# Glossary

## A

**active window:** The window that is currently selected for moving, sizing, editing, closing, or some other function.

**aliasing:** A method of customizing debugger commands; aliasing provides a shorthand method for entering often-used command strings.

**assembly mode:** A debugging mode that shows assembly language code in the DISASSEMBLY window.

**autoexec.bat:** A batch file that contains DOS commands for initializing your PC.

## B

**batch file:** One of two different types of files. One type contains DOS commands for the PC to execute. A second type of batch file contains debugger commands for the debugger to execute. The PC does not execute debugger batch files, and the debugger does not execute PC batch files.

**breakpoint:** A point within your program where execution halts because of a previous request from you.

## C

**C compiler:** A program that translates C source statements into assembly language source statements.

**casting:** A feature of C expressions that allows you to use one type of data as if it were a different type of data.

**click:** To press and release a mouse button without moving the mouse.

**COFF:** *Common object file format*. An implementation of the object file format of the same name developed by AT&T.

**command line:** The portion of the COMMAND window where you can enter commands.

**command-line cursor:** A block-shaped cursor that identifies the current character position on the command line.

**COMMAND window:** A window that provides an area for you to enter commands and for the debugger to echo command entry, show command output, and list progress or error messages.

**CPU window:** A window that displays the contents of HET on-chip registers, including the current address register, status register, A-file registers, and B-file registers.

**current-field cursor:** A screen icon that identifies the current field in the active window.

**cursor:** An icon on the screen (such as a rectangle or a horizontal line) that is used as a pointing device. The cursor is usually under mouse or keyboard control.

## D

**data-display windows:** Windows for observing and modifying various types of data. This category includes the MEMORY, CPU, and WATCH windows.

**D_DIR:** An environment variable that identifies the directory containing the commands and files necessary for running the debugger.

**debugger:** A window-oriented software interface that helps you to debug HET programs running on an HET simulator.

**disassembly:** Assembly language code formed from the reverse-assembly of the contents of memory.

**DISASSEMBLY window:** A window that displays the disassembly of memory contents.

**discontinuity:** A state in which the addresses fetched by the debugger become nonsequential as a result of instructions that load the CAR with new values, such as branches, and returns.

**display area:** The portion of the COMMAND window where the debugger echoes command entry, shows command output, and lists progress or error messages.

**D_OPTIONS:** An environment variable that you can use for identifying often-used debugger options.

**drag:** To move the mouse while pressing one of the mouse buttons.

**D_SRC:** An environment variable that identifies directories containing program source files.

## E

**EGA:** *Enhanced graphics adaptor.* An industry standard for video cards.

**EISA:** *Extended industry standard architecture.* A standard for PC buses.

**environment variable:** A special system symbol that the debugger uses for finding directories or obtaining debugger options.

## H

**HETADDR:** *Current address register.*

**HET:** *TMS470 high-end timer.* A module that provides sophisticated timing functions for complex, real-time applications, such as automobile engine management or power-train management. These applications require the measurement of information from multiple sensors and drive actuators with complex and accurate time pulses.

## 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.

**ISA:** *Industry standard architecture*. A subset of the EISA standard.

## M

**memory map:** A map of memory space that tells the debugger which areas of memory can and 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.

**minimal mode:** A debugging mode that displays the COMMAND window and WATCH window only.

**mouse cursor:** A block-shaped cursor that tracks mouse movements over the entire display.

## P

**PC:** *Personal computer.*

**point:** To move the mouse cursor until it overlays the desired object on the screen.

**pulldown menu:** A command menu that is accessed by name or with the mouse from the menu bar at the top of the debugger display.

## S

**scalar type:** A C type in which the variable is a single variable, not composed of other variables.

**scrolling:** A method of moving the contents of a window up, down, left, or right to view contents that were not originally shown.

**side effects:** A feature of C expressions in which using an assignment operator in an expression affects the value of one of the components used in the expression.

**simulator:** A development tool that simulates the operation of the HET and lets you execute and debug applications programs by using the HET debugger.

**single-step:** A form of program execution that allows you to see the effects of each statement. The program is executed statement by statement; the debugger pauses after each statement to update the data-display windows.

**symbol table:** A file that contains the names of all variables in your HET program.

## V

**VGA:** *Video Graphics Array.* An industry standard for video cards.

# W

**WATCH window:** A window that displays the values of selected expressions, symbols, addresses, and registers.

**window:** A defined rectangular area of virtual space on the display.

# Index



## A

## B

# C

## D

## E

## F

## G



## H



## I



## K

## N



## O



## P

## Q



## R



## S

## T

## U



## V



## W

## X



## Z