Texas Instruments

# TMS320C80 (MVP) C Source Debugger

## User's Guide

1995

Microprocessor Development Systems

TEXAS INSTRUMENTS

Printed in U.S.A., March 1995
D418004-9761 revision*

SPRU107A

User's
Guide

# TMS320C80 (MVP)
# C Source Debugger

1995

# TMS320C80 (MVP)
# C Source Debugger
# User's Guide

SPRU107
March 1995

PRINTED WITH SOY INK ™

TEXAS INSTRUMENTS

Printed on Recycled Paper

## IMPORTANT NOTICE

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

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

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

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

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

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

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

## Preface

# Read This First

## *About This Manual*

The TMS320C80 MVP (multimedia video processor) is Texas Instruments first single-chip multiprocessor DSP (digital signal processor) device. The MVP contains five powerful, fully programmable processors: a master processor (MP) and four parallel processors (PPs). The MP is a 32-bit RISC (reduced instruction set computer) with an integral, high-performance IEEE-754 floating-point unit. Each PP is an advanced 32-bit DSP; thus, in addition to having similar processing capabilities as conventional DSPs, each PP has advanced features to accelerate operation on a variety of data types.

The MVP supports a variety of parallel-processing configurations, which facilitates a wide range of multimedia and other applications that require high processing speeds. Applications include image processing, two- and three-dimensional and virtual reality graphics, audio/video digital compression, and telecommunications.

This manual describes the MVP C source debuggers. The C source debuggers are an advanced programmer's interface to develop, test, and refine MVP C programs and assembly language programs. The debuggers are the interface to the MVP simulator and emulator. There are two types of debuggers: one for the MP and one for the PPs. You can run multiple debuggers at one time under the control of the parallel debug manager. This manual provides information about the features and operation of the debuggers and the parallel debug manager; it also includes basic information about C expressions and a description of progress and error messages.

## Notational Conventions

This document uses the following conventions.

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

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

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

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

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

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

  | Command | Result displayed in the COMMAND window |
  |---|---|
  | **`whatis giant`** | `struct zzz giant[100];` |
  | **`whatis xxx`** | `struct xxx   {`<br>`   int a;`<br>`   int b;`<br>`   int c;`<br>`   int f1 : 2;`<br>`   int f2 : 4;`<br>`   struct xxx *f3;`<br>`   int f4[10];`<br>`}` |

  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 COMMAND window display area.

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

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

  **mem** is the command. This command has two parameters, indicated by *expression* and *display format*. The first parameter must be an actual C expression; the second parameter, which identifies a specific display format, is optional.

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

  run [*expression*]

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

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

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

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

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

## *Information About Cautions*

> **This is an example of a caution statement.**
>
> **A caution statement describes a situation that could potentially damage your software or equipment.**

Please read each caution statement carefully.

## *Related Documentation From Texas Instruments*

The following books describe the TMS320C80 MVP 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.

***TMS320C80 Multimedia Video Processor Data Sheet*** (literature number SPRS023) describes the features of the ’C80 device and provides pinouts, electrical specifications, and timings for the device.

***TMS320C80 Multimedia Video Processor (MVP) Technical Brief*** (literature number SPRU106) provides an overview of the ’C80 features, development environment, architecture, and memory organization.

***TMS320C80 (MVP) Code Generation Tools User’s Guide*** (literature number SPRU108) describes the ’C80 code generation tools. This manual provides information about the features and operation of the linker and the master processor (MP) and parallel processor (PP) C compilers and assemblers. It also includes a description of the common object file format (COFF) and shows you how to link MP and PP code.

***TMS320C80 (MVP) Master Processor User's Guide*** (literature number SPRU109) describes the 'C80 master processor (MP). This manual provides information about the MP features, architecture, operation, and assembly language instruction set; it also includes sample applications that illustrate various MP operations.

***TMS320C80 (MVP) Multitasking Executive User's Guide*** (literature number SPRU112) describes the 'C80 multitasking executive software. This manual provides information about the multitasking executive software features, operation, and interprocessor communications; it also includes a list of task error codes.

***TMS320C80 (MVP) Parallel Processor User's Guide*** (literature number SPRU110) describes the 'C80 parallel processor (PP). This manual provides information about the PP features, architecture, operation, and assembly language instruction set; it also includes software applications and optimizations.

***TMS320C80 (MVP) System-Level Synopsis*** (literature number SPRU113) contains the 'C80 system-level synopsis, which describes the 'C80 features, development environment, architecture, memory organization, and communication network (the crossbar).

***TMS320C80 (MVP) Transfer Controller User's Guide*** (literature number SPRU105) describes the 'C80 transfer controller (TC).This manual provides information about the TC features, functional blocks, and operation; it also includes examples of block write operations for big- and little-endian modes.

***TMS320C80 (MVP) Video Controller User's Guide*** (literature number SPRU111) describes the 'C80 video controller (VC). This manual provides information about the VC features, architecture, and operation; it also includes procedures and examples for programming the serial register transfer (SRT) controller and the frame timer registers.

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

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

## *If You Need Assistance. . .*

| **If you want to. . .** | **Do this. . .** |
|---|---|
| Request more information about Texas Instruments Digital Signal Processing (DSP) products | Write to:<br>Texas Instruments Incorporated<br>Market Communications Manager, MS 736<br>P.O. Box 1443<br>Houston, Texas 77251–1443 |
| Order Texas Instruments documentation | Call the TI Literature Response Center:<br>**(800) 477–8924** |
| Ask questions about product operation or report suspected problems | Call the DSP hotline:<br>**(713) 274–2320**<br>**FAX: (713) 274–2324** |
| Report mistakes in this document or any other TI documentation | Fill out and return the reader response card at the end of this book, or send your comments to:<br>Texas Instruments Incorporated<br>Technical Publications Manager, MS 702<br>P.O. Box 1443<br>Houston, Texas 77251–1443<br>Electronic mail: **comments@books.sc.ti.com** |

## *FCC Warning*

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

## *Trademarks*

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

SPARC is a trademark of SPARC International, Inc.

SPARCstation is licensed exclusively to Sun Microsystems, Inc.

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

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

# Contents

Provides an introduction to the parallel debug manager and demonstrates how the parallel debug manager can be used to support a multiprocessing debugging environment (for use with the simulator only).

Describes the parallel debug manager and the commands that you can use to control multiple debuggers.

Describes the default displays, tells you how to switch between assembly language and C debugging, describes the various types of windows on the display, and tells you how to move and size the windows.

Describes the rules for entering commands from the command line, tells you how to use the pulldown menus and dialog boxes (for entering parameter values), and provides general information about entering commands from a batch file.

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

# Figures

# Tables

# Examples

Chapter 1

# Overview of a Code Development and Debugging System

The C source debugger is an advanced programmer’s interface that helps you develop, test, and refine TMS320C80 (MVP) C programs and assembly language programs. The debugger is the interface to the ’C80 simulator and ’C80 emulator.

There are two types of debuggers: one for the master processor (MP) and one for the advanced DSPs (referred to as PPs). Both types of debuggers use the same interface. Throughout this book, the term **debugger** refers to that interface and describes features that apply to both the MP and PP debuggers. You can run multiple debuggers at one time: one debugger for the MP and one for each PP in your system. Multiple debuggers run under the control of the parallel debug manager.

This chapter gives an overview of the debugger interface and the parallel debug manager.

**Topics**

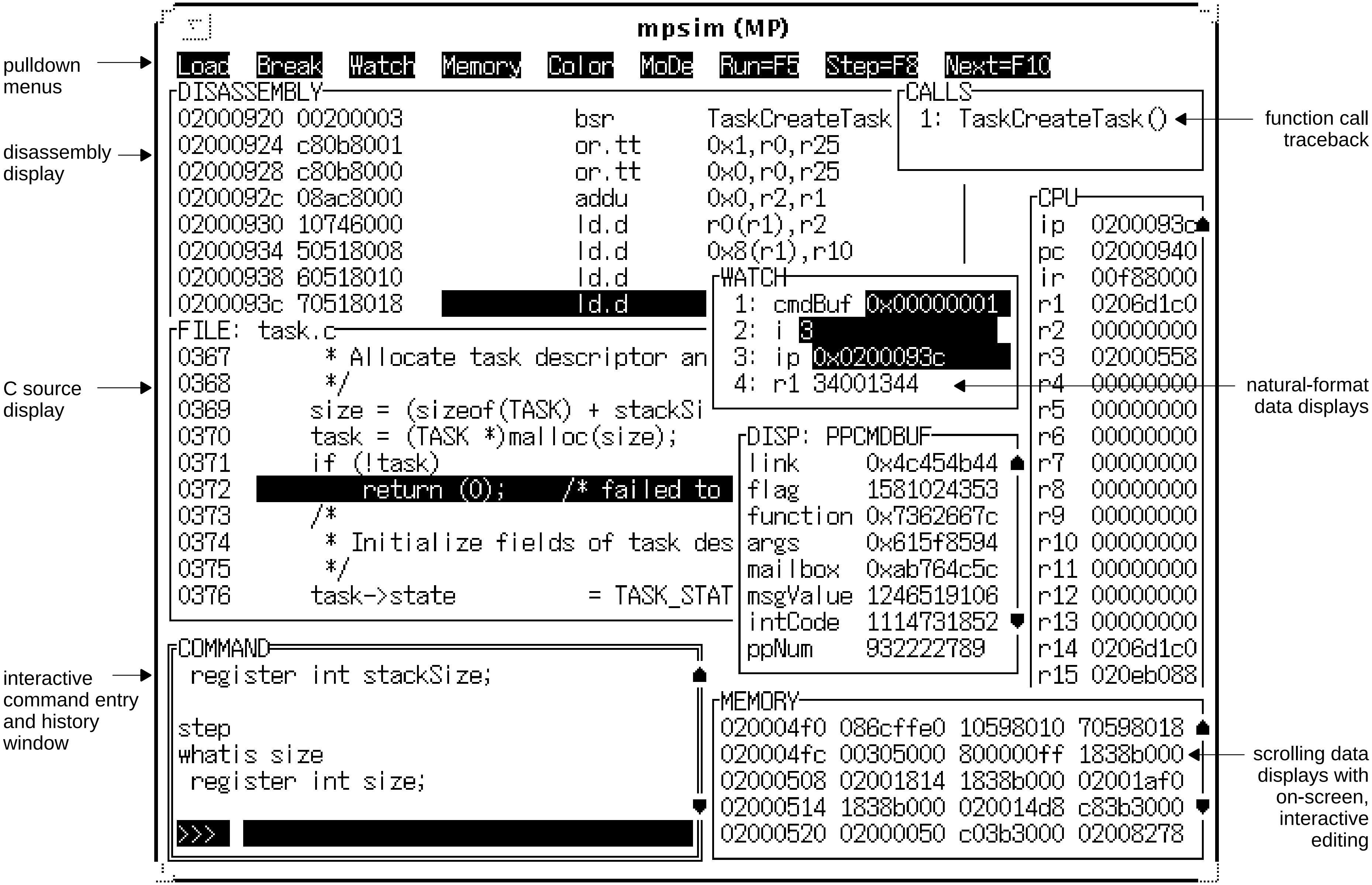
## 1.1 Description of the MVP C Source Debugger

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

The Texas Instruments advanced programmer's interface is easy to learn and use. The friendly window-, mouse-, and menu-oriented interface reduces learning time and eliminates the need to memorize complex commands. The debugger's customizable displays and flexible command entry let you develop a debugging environment that suits your needs—you won't be locked into a rigid environment. A shortened learning curve and increased productivity reduce the software development cycle, so you'll get your product to market faster.

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

Figure 1–1. The Basic Debugger Display



```
DISASSEMBLY
02000920 00200003          bsr      TaskCreateTask
02000924 c80b8001          or.tt    0x1,r0,r25
02000928 c80b8000          or.tt    0x0,r0,r25
0200092c 08ac8000          addu     0x0,r2,r1
02000930 10746000          ld.d     r0(r1),r2
02000934 50518008          ld.d     0x8(r1),r10
02000938 60518010          ld.d
0200093c 70518018          ld.d
```

```
CALLS
 1: TaskCreateTask()
```

```
CPU
ip   0200093c
pc   02000940
ir   00f88000
r1   0206d1c0
r2   00000000
r3   02000558
r4   00000000
r5   00000000
r6   00000000
r7   00000000
r8   00000000
r9   00000000
r10  00000000
r11  00000000
r12  00000000
r13  00000000
r14  0206d1c0
r15  020eb088
```

```
WATCH
 1: cmdBuf 0x00000001
 2: i 3
 3: ip 0x0200093c
 4: r1 34001344
```

```
FILE: task.c
0367        * Allocate task descriptor an
0368        */
0369       size = (sizeof(TASK) + stackSi
0370       task = (TASK *)malloc(size);
0371       if (!task)
0372           return (0);     /* failed to
0373       /*
0374        * Initialize fields of task des
0375        */
0376       task->state          = TASK_STAT
```

```
DISP: PPCMDBUF
link      0x4c454b44
flag      1581024353
function  0x7362667c
args      0x615f8594
mailbox   0xab764c5c
msgValue  1246519106
intCode   1114731852
ppNum     932222789
```

```
COMMAND
 register int stackSize;

step
whatis size
 register int size;

>>>
```

```
MEMORY
020004f0 086cffe0 10598010 70598018
020004fc 00305000 800000ff 1838b000
02000508 02001814 1838b000 02001af0
02000514 1838b000 020014d8 c83b3000
02000520 02000050 c03b3000 02008278
```

## Key features of the debugger

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

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

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

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

❑ **Continuous update.** The debugger continuously updates information on the screen, highlighting changed values.

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

❑ **Flexible command entry.** There are 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. Want to re-enter a command? No need to retype it—simply use the command history.

🖱 key 👆

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

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

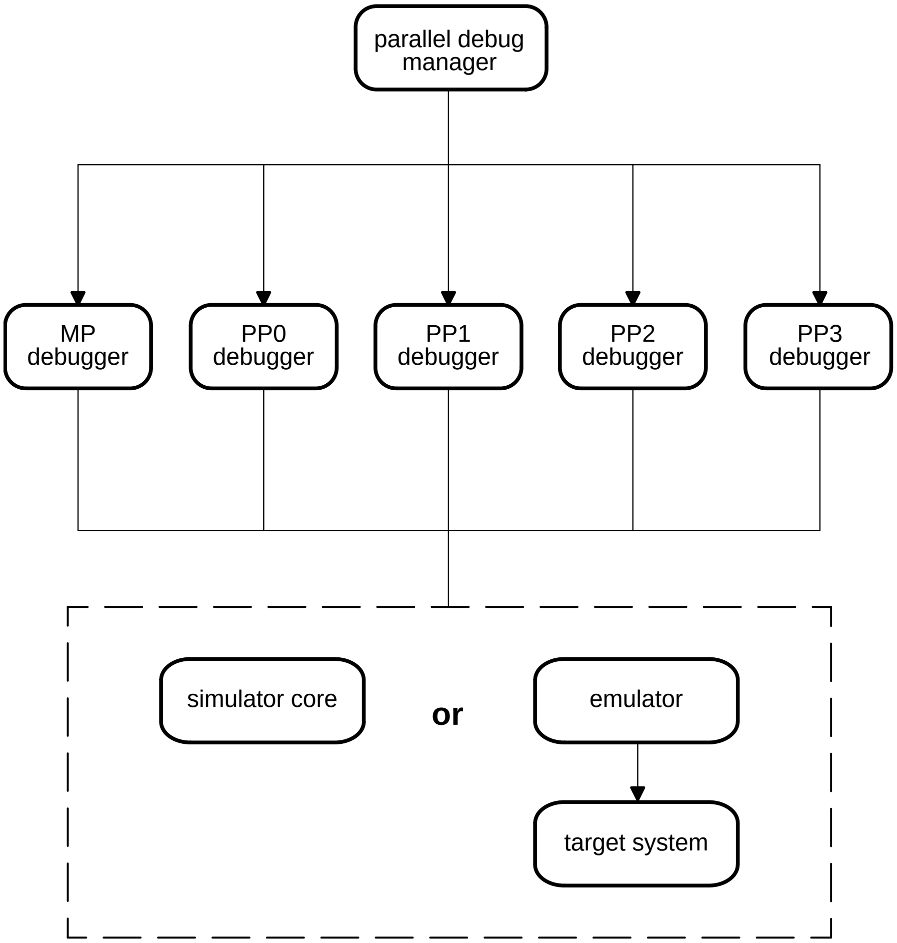
Create and save as many custom configurations as you like, or use the defaults. A color display is preferable; the various types of information on the display are easier to distinguish when they are highlighted with color.

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

## 1.2 Description of the Parallel Debug Manager

The parallel debug manager is a command shell from which you can invoke and control multiple debuggers. This allows you to debug code in a multiprocessing environment. As Figure 1–2 shows, you can run one MP debugger and up to four PP debuggers under the parallel debug manager.

Figure 1–2. The Parallel Debug Manager Environment



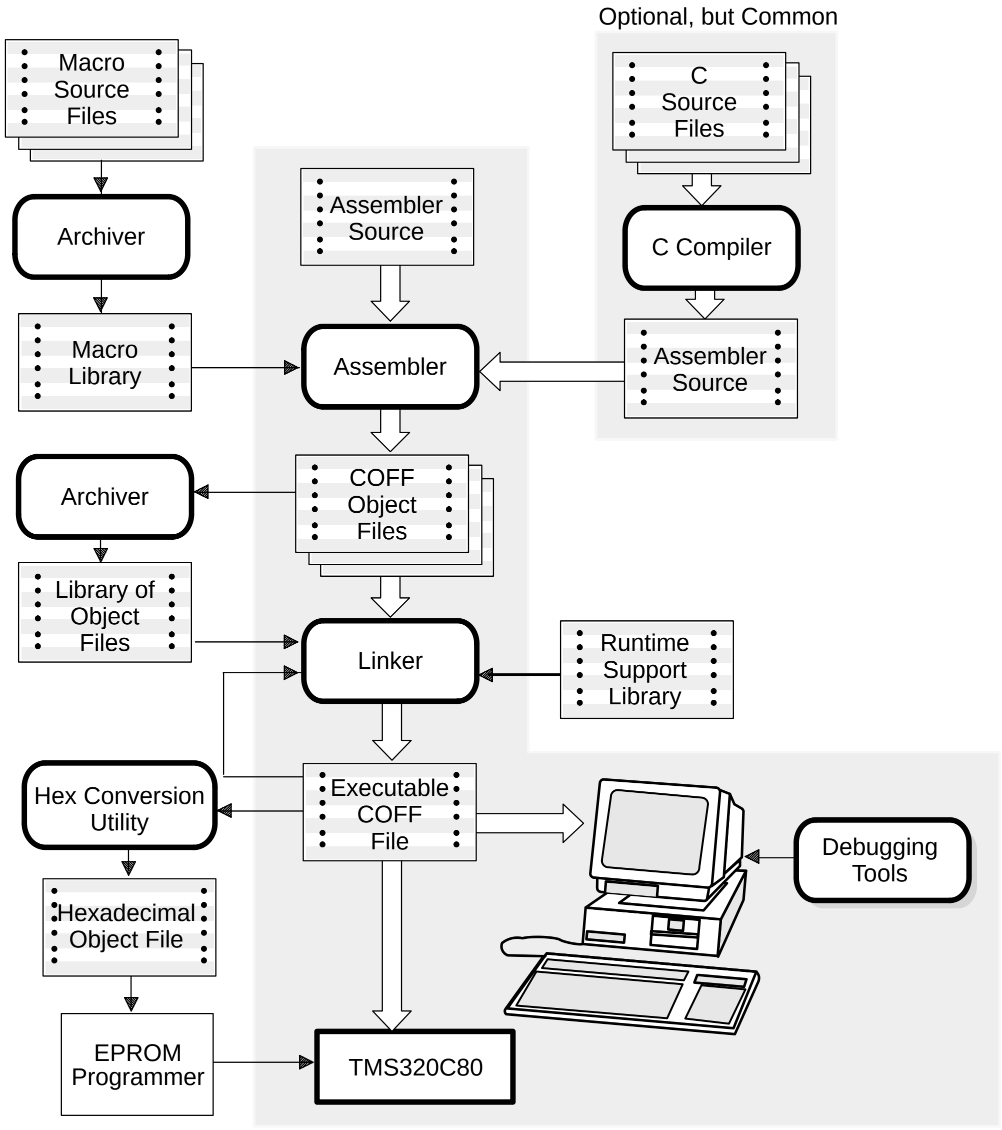
From the parallel debug manager, you can:

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

## 1.3 Developing Code for the MVP

The MVP is supported by a complete set of hardware and software development tools, including a C compiler, an assembler, and a linker. Figure 1–3 illustrates the MVP code development flow. The most common paths of development are shaded with gray; the other portions are optional.

Figure 1–3. MVP Software Development Flow



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

The following list describes the tools shown in Figure 1–3.

C Compiler

The MP and PP **optimizing ANSI C compilers** are full-featured optimizing compilers that translate standard ANSI C programs into MP and PP assembly language source, respectively. Key characteristics include:

□ **Standard ANSI C.** The ANSI standard is a precise definition of the C language, agreed upon by the C community. The standard encompasses most of the recent extensions to C. To an increasing degree, ANSI conformance is a requirement for C compilers in the DSP community.

□ **Optimization.** The compilers use several advanced techniques for generating efficient, compact code from C source.

□ **Assembly language output.** The MP and PP compilers generate MP and PP assembly language source, respectively, that you can inspect (and modify, if desired).

□ **ANSI standard runtime support.** The compiler package comes with a complete runtime library that conforms to the ANSI C library standard. The library includes functions for string manipulation, dynamic memory allocation, data conversion, timekeeping, trigonometry, exponential operations, and hyperbolic operations. Functions for I/O and signal handling are not included, because they are application specific.

□ **Flexible assembly language interface.** The compiler has straightforward calling conventions, allowing you to easily write assembly and C functions that call each other.

□ **Shell program.** The compiler package includes a shell program that enables you to compile, assemble, and link programs in a single step.

□ **Source interlist utility.** The compiler package includes a utility that interlists your original C source statements into the assembly language output of the compiler. This utility provides you with an easy method for inspecting the assembly code generated for each C statement.

Linker

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

Assembler

The **assemblers** translate assembly language source files into machine language object files in common object file format (COFF). There are two separate assemblers and two separate assembly languages for the MVP (MP and PP); however, once MP and PP assembly source files are assembled into COFF, they are linked into common memory.

Debugging Tools

The main purpose of the development process is to produce a module that can be executed in an **MVP target system**. You can use one of two **debugging tools** to refine and correct your code. Available products include:

❏ A realtime in-circuit **emulator**
❏ A software **simulator**

Both of these tools use the MVP debugger as a software interface.

Hex Conversior Utility

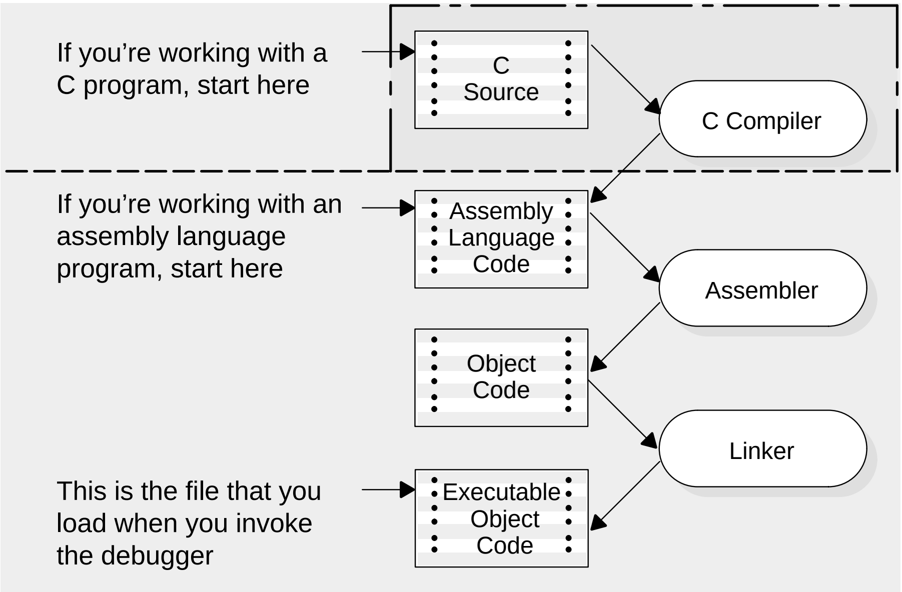
A hex conversion utility is available to convert a COFF object file into an ASCII-Hex, Intel, Motorola-S, Tektronix, or TI-tagged object-format file that can be downloaded to an EPROM programmer.

## 1.4 Preparing Your Program for Debugging

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

You can compile, assemble, and link a program by invoking the compiler, assembler, and linker in separate steps; or you can perform all three actions in a single step by using the MPCL shell program (for the MP) or the PPCL shell program (for the PP). Subsection 1.1.1, *Invoking the C Compilers*, in the *MVP Code Generation Tools User's Guide* contains complete instructions for invoking the tools individually and for using the shell programs.

Figure 1–4. Steps in Preparing a Program


If you're working with a C program, start here
C Source
C Compiler
If you're working with an assembly language program, start here
Assembly Language Code
Assembler
Object Code
Linker
This is the file that you load when you invoke the debugger
Executable Object Code

| | |
|---|---|
| If you're preparing to debug a C program. . . | 1) Compile the program; use the –g option.<br>2) Assemble the resulting assembly language program.<br>3) Link the resulting object file.<br>This produces an object file that you can load into the debugger. |
| If you're preparing to debug an assembly language program. . . | 1) Assemble the assembly language source file. If you want to include a local symbol table, assemble the program with the –s option.<br>2) Link the resulting object file.<br>This produces an object file that you can load into the debugger. |

## 1.5 Invoking the Debuggers and the Parallel Debug Manager

There are two ways to invoke the debugger:

- You can invoke a standalone debugger that cannot communicate with other debuggers.
- You can invoke several debuggers that are under control of the parallel debug manager.

### *Invoking a standalone debugger*

Here's the basic format for the commands that invoke a standalone debugger:

```
Emulator:
   MP:      mpemu   –n processor name   [filename]   [–options]
   PP:      ppemu   –n processor name   [filename]   [–options]

Simulator:
   MP:      mpsim   [filename]   [–options]
   PP:      ppsim   [–n processor name]   [filename]   [–options]
```

- **mpemu, ppemu, mpsim,** and **ppsim** are the commands that invoke the debugger. Enter one of these commands from the operating-system command line.
- **–n** *processor name* supplies a processor name that corresponds to a processor in your system. The processor name can consist of up to eight alphanumeric characters and must begin with an alphabetic character. Note that the name is not case sensitive.
  - **If you're invoking the simulator version of the MP debugger,** you don't need the –n option; the *processor name* defaults to MP.
  - **If you're invoking the simulator version of the PP debugger,** the processor name that you use **must** end in a 0, 1, 2, or 3 that corresponds to the actual PP in your system. For example, to name the debugger for the PP1, you can use the name DSP1. The *processor name* defaults to PP0 if you are invoking a PP version of the debugger and you don't use –n. However, you **must** use the –n option to specify a processor name if you want to invoke a debugger for PP1, PP2, or PP3 in a multiple-PP system.

   - **If you’re invoking the emulator version of either the MP or PP debugger,** the processor name that you supply must be a combination of the MVP device name that you defined in your board.dat file and the processor name (MP, PP0, PP1, etc.), separated by an underscore character. (The board.dat file is described in Appendix E, *Describing Your Target System to the Debugger*.) If you’re naming a PP, the processor name must end in a 0, 1, 2, or 3 that corresponds to the actual PP in your system.

     For example, if you had an MVP in your device chain that was called MVP1 and you wanted to invoke a debugger for PP1 of that device, you’d enter:

     ```
     ppemu –n MVP1_PP1 ↵
     ```

- *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 isn’t in the current directory, you must supply the entire pathname. If you don’t supply an extension for the filename, the debugger assumes that the extension is .out, unless you’re using multiple extensions; you must specify the **entire** filename if the filename has more than one extension.

- *–options* supply the debugger with additional information. For a complete list of debugger options, see Section 1.6.

**Note:**

Before you invoke the emulator version of the debugger (mpemu or ppemu) for the first time, you must perform a hardware reset (drive the MVP reset pin active). If you don’t perform a reset before you invoke the emulator version of the debugger, the MVP endian selection and MP halt state are not set up properly. These functions are selected only by a hardware reset.

## Invoking multiple debuggers

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

**pdm**

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

When you invoke the parallel debug manager, it looks for a file called init.pdm. This file contains initialization commands for the parallel debug manager. The parallel debug manager searches for the init.pdm file in the current directory and in the directories you specify with the D_DIR environment variable. If the parallel debug manager can't find the initialization file, you see this message: Cannot open take file.

Once you have invoked the parallel debug manager, you can invoke and control debuggers for each of the processors in your multiprocessing system. You can invoke one MP debugger and up to four PP debuggers in either the emulator or simulator versions; you cannot mix the emulator and simulator versions.

To invoke an emulator or simulator version of the debugger from the parallel debug manager, use the SPAWN command. Here's the basic format for this command:

Emulator:
MP: **spawn mpemu –n** *processor name* [*filename*] [–*options*]
PP: **spawn ppemu –n** *processor name* [*filename*] [–*options*]

Simulator:
MP: **spawn mpsim** [*filename*] [–*options*]
PP: **spawn ppsim** [**–n** *processor name*] [*filename*] [–*options*]

- **mpemu, ppemu, mpsim,** and **ppsim** are the executables that invoke the MP or PP version of the debugger. The parallel debug manager associates the *processor name* with the actual processor according to which executable you use. For example, to invoke a simulator version of the PP debugger and associate the name PP1 with it, you would enter:

  ```
  spawn ppsim –n PP1 ⏎
  ```

  To invoke a debugger, the parallel debug manager must be able to find the executable file for that debugger. The parallel debug manager first searches the current directory and then searches the directories listed with the path shell variable.

- **–n** *processor name* supplies a processor name. The parallel debug manager uses processor names to identify the debuggers that are running. The processor name can consist of up to eight alphanumeric characters and must begin with an alphabetic character. Note that the name is not case sensitive.

  - **If you're invoking the simulator version of the MP debugger,** you don't need to use the –n option; the *processor name* defaults to MP.
  - **If you're invoking the simulator version of the PP debugger,** the processor name that you use *must* end in a 0, 1, 2, or 3 that corresponds to the actual PP in your system. For example, to name the debugger for the PP1, you can use the name DSP1. The *processor name* defaults to PP0 if you are invoking a PP version of the debugger and you don't use –n. However, you **must** use the –n option to specify a processor name if you want to invoke a debugger for the PP1, PP2, or PP3.

  - **If you’re invoking the emulator version of either the MP or PP debugger,** the processor name that you supply must be a combination of the MVP device name that you defined in your board.dat file and the processor name (MP, PP0, PP1, etc.), separated by an underscore character. (The board.dat file is described in Appendix E, *Describing Your Target System to the Debugger*.) If you’re naming a PP, the processor name must end in a 0, 1, 2, or 3 that corresponds to the actual PP in your system.

    For example, if you had an MVP in your device chain that was called “MVP1” and you wanted to invoke a debugger for PP1 of that device, you’d enter:

    ```
    spawn ppemu –n MVP1_PP1 ↵
    ```

- *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 isn’t in the current directory, you must supply the entire pathname. If you don’t supply an extension for the filename, the debugger assumes that the extension is .out, unless you’re using multiple extensions; you must specify the **entire** filename if the filename has more than one extension.

- *–options* supply the debugger with additional information. For a complete list of debugger options, see Section 1.6.

**Note:**

Before you invoke the emulator version of the debugger (mpemu or ppemu) for the first time, you must perform a hardware reset (drive the MVP $\overline{\text{RESET}}$ pin low). If you don’t perform a reset before you invoke the emulator version of the debugger, the MVP endian selection and MP halt state may not set up properly. These functions are selected only by a hardware reset.

## 1.6 Debugger Options

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

Table 1–1. Summary of Debugger Options

| Option | Description | Debugger Tools | See Page |
|---|---|---|---|
| –b[b] | Select the screen size | All | DB:1-17 |
| –d *machinename* | Display debugger on different machine (X Windows only) | All | DB:1-17 |
| –f *filename* | Identify a new board configuration file | Emulator | DB:1-17 |
| –i *pathname* | Identify additional directories | All | DB:1-18 |
| –o | Enable C I/O | All | DB:1-18 |
| –p *driver number* | Identify the device driver | Emulator | DB:1-19 |
| –s | Load the symbol table only | All | DB:1-19 |
| –t *filename* | Identify a new initialization file | All | DB:1-19 |
| –v | Load without the symbol table | All | DB:1-19 |
| –x | Ignore D_OPTIONS | All | DB:1-19 |

## Selecting the screen size (–b option)

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 easily fit up to five default-size debuggers on your screen. However, you can change the default screen size by using one of the –b options, which provides a preset screen size, or by resizing the screen at runtime.

❑ **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.

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

If you are using OpenWindows, you can display the debugger on a different machine than the one the program is running on by using the –d option. For example, if you are running the MP 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:

```
spawn mpsim –d barney:0 ↵
```

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

## Identifying a new board configuration file (–f option)

This option is valid only when you're using the emulator. The –f option allows you to specify a board configuration file that will be used instead of board.dat. The format for this option is:

**–f** *filename*

## *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:

**spawn mpemu –n** *name* **–i** *pathname*$_1$ **–i** *pathname*$_2$ **–i** *pathname*$_3$ . . .

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

## *Enabling C I/O functions (–o option)*

The –o option enables the use of C I/O functions. The C I/O library of functions makes it possible for the MVP to access the host's operating system to perform I/O (via the debugger). When used in conjunction with the MVP code generation tools, the capability to perform I/O on the host provides you with more options when debugging and testing code written for the MVP. See Section 5, *The C I/O Library*, in the *MVP Code Generation Tools User's Guide*, for a description of the standard input/output functions.

**Note:**

The interface between the debugger and the host operating system and the common data structures used by C I/O make it impossible for you to use C I/O on more than one processor per application.

## *Identifying the device driver (–p option)*

The –p option is valid only when you're using the emulator. When you invoke the debugger, you must identify the number of the device driver file that you're using. You set up the device driver and the associated file when you install the emulator (see *Step 2: Setting Up Your Workstation to Recognize the Emulator,* in the *TMS320C8x Workstation Emulator Installation Guide*). The format for the –p option is:

**–p** *driver number*

For example, if you set up your device driver file to be *rsd5a* for device driver *sd5*, you would enter the following:

```
mpemu –n MVP1_MP –p 5 ⏎
```

You can omit the –p option if the default device driver file (*rsd4a*) is the device driver file for the emulator.

## *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 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 init.cmd. The format for this option is:

**–t** *filename*

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

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

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

## *Ignoring D_OPTIONS (–x option)*

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

## 1.7 Exiting a Debugger or the Parallel Debug Manager

To exit any version of the debugger, enter the following command from the COMMAND window of the debugger you want to close:

```
quit ⏎
```

You can also enter QUIT from the command line of the parallel debug manager to quit **all** of the debuggers (and also close the parallel debug manager).

An alternative method of exiting a debugger is moving your mouse cursor on the border of the window that contains the debugger and executing Quit or Close from the window-manager menu.

Chapter 2

# An Introductory Tutorial to the C Source Debugger

This chapter provides a step-by-step, hands-on demonstration of the MP C source debugger's basic features. The features shown for the MP debugger are the same for the PP debugger. However, you can perform the tutorial only on the simulator version of the MP debugger. You cannot perform the tutorial on the emulator version of the MP debugger nor on either the emulator or simulator version of the PP debugger.

This is not the kind of tutorial that you can take home to read—the tutorial is effective only if you're sitting at your workstation, performing the lessons in the order that they're presented. The tutorial contains two sets of lessons (11 in the first, 13 in the second) and takes about one hour to complete.

## *How to use this tutorial*

This tutorial contains three basic types of information:

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

> Make the CPU window the active window:
>
> **`win CPU`** ⏎

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

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

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

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

**Important!** This tutorial assumes that you have correctly and completely installed your debugger and software tools (including invoking any files or system commands as instructed in the *TMS320C8x (MVP) Software Tools Getting Started*).

## *A note about entering commands*

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

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

## *An escape route (just in case)*

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

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

## *Invoke the debugger and load the sample program’s object code*

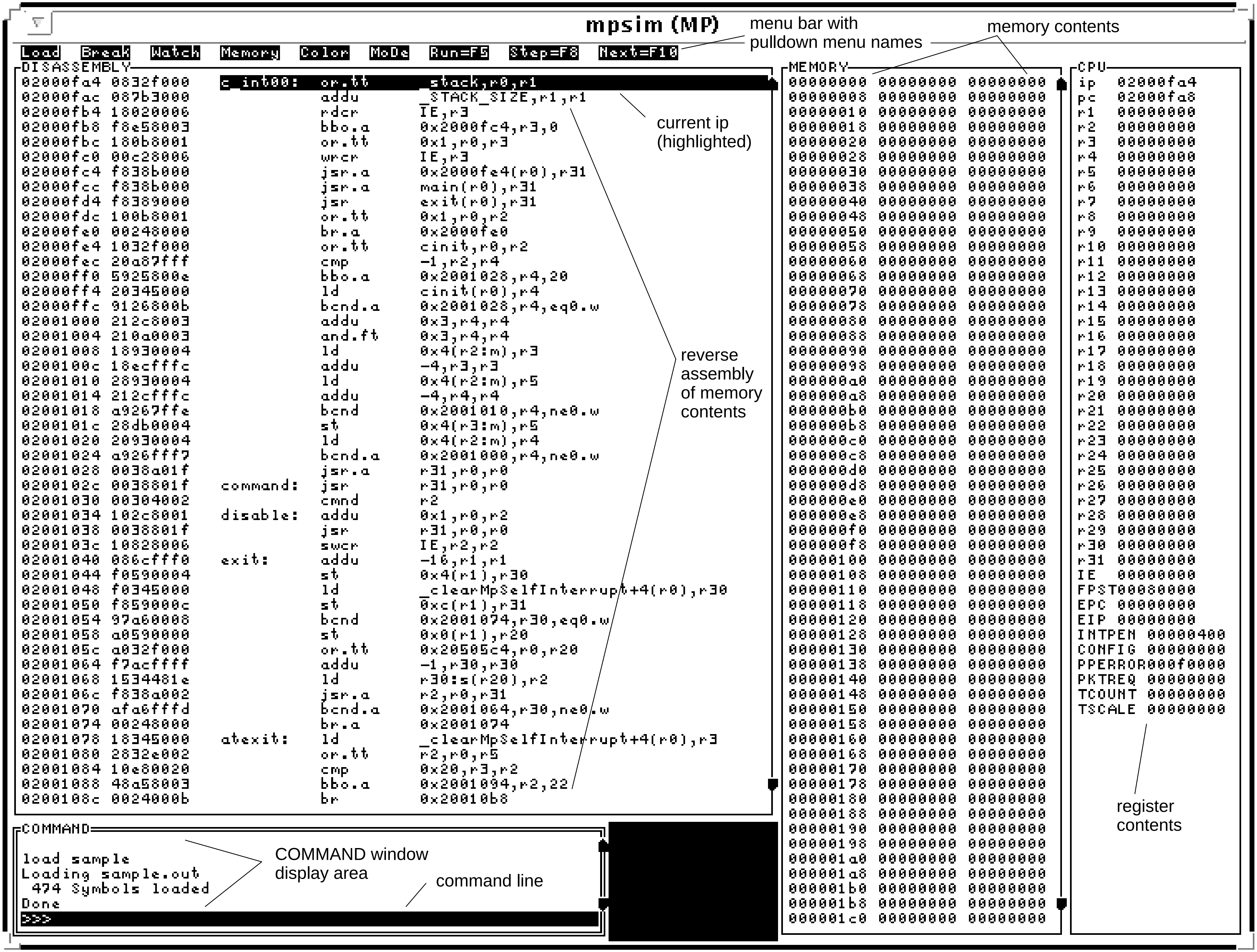
Included with the debugger is a demonstration program named *sample*. This lesson shows you how to invoke the MP debugger and load the sample program.

From a command shell window, invoke the MP debugger and load the sample program:

```
mpsim /your_pathname/mvp/demo/sample ↵
```

## *Take a look at the display. . .*

Now you should see a display similar to the figure on the following page (it may not be exactly the same, but it should be close).



```
DISASSEMBLY
02000fa4 0832f000   c_int00:  or.tt      _stack,r0,r1
02000fac 087b3000             addu       _STACK_SIZE,r1,r1
02000fb4 18020006             rdcr       IE,r3
02000fb8 f8e58003             bbo.a      0x2000fc4,r3,0
02000fbc 180b8001             or.tt      0x1,r0,r3
02000fc0 00c28006             wrcr       IE,r3
02000fc4 f838b000             jsr.a      0x2000fe4(r0),r31
02000fcc f838b000             jsr.a      main(r0),r31
02000fd4 f8389000             jsr        exit(r0),r31
02000fdc 100b8001             or.tt      0x1,r0,r2
02000fe0 00248000             br.a       0x2000fe0
02000fe4 1032f000             or.tt      cinit,r0,r2
02000fec 20a87fff             cmp        -1,r2,r4
02000ff0 5925800e             bbo.a      0x2001028,r4,20
02000ff4 20345000             ld         cinit(r0),r4
02000ffc 9126800b             bcnd.a     0x2001028,r4,eq0.w
02001000 212c8003             addu       0x3,r4,r4
02001004 210a0003             and.ft     0x3,r4,r4
02001008 18930004             ld         0x4(r2:m),r3
0200100c 18ecfffc             addu       -4,r3,r3
02001010 28930004             ld         0x4(r2:m),r5
02001014 212cfffc             addu       -4,r4,r4
02001018 a9267ffe             bcnd       0x2001010,r4,ne0.w
0200101c 28db0004             st         0x4(r3:m),r5
02001020 20930004             ld         0x4(r2:m),r4
02001024 a926fff7             bcnd.a     0x2001000,r4,ne0.w
02001028 0038a01f             jsr.a      r31,r0,r0
0200102c 0038801f   command:  jsr        r31,r0,r0
02001030 00304002             cmnd       r2
02001034 102c8001   disable:  addu       0x1,r0,r2
02001038 0038801f             jsr        r31,r0,r0
0200103c 10828006             swcr       IE,r2,r2
02001040 086cfff0   exit:     addu       -16,r1,r1
02001044 f0590004             st         0x4(r1),r30
02001048 f0345000             ld         _clearMpSelfInterrupt+4(r0),r30
02001050 f859000c             st         0xc(r1),r31
02001054 97a60008             bcnd       0x2001074,r30,eq0.w
02001058 a0590000             st         0x0(r1),r20
0200105c a032f000             or.tt      0x20505c4,r0,r20
02001064 f7acffff             addu       -1,r30,r30
02001068 1534481e             ld         r30:s(r20),r2
0200106c f838a002             jsr.a      r2,r0,r31
02001070 afa6fffd             bcnd.a     0x2001064,r30,ne0.w
02001074 00248000             br.a       0x2001074
02001078 18345000   atexit:   ld         _clearMpSelfInterrupt+4(r0),r3
02001080 2832e002             or.tt      r2,r0,r5
02001084 10e80020             cmp        0x20,r3,r2
02001088 48a58003             bbo.a      0x2001094,r2,22
0200108c 0024000b             br         0x20010b8
```

```
MEMORY
00000000 00000000 00000000
00000008 00000000 00000000
00000010 00000000 00000000
00000018 00000000 00000000
00000020 00000000 00000000
00000028 00000000 00000000
00000030 00000000 00000000
00000038 00000000 00000000
00000040 00000000 00000000
00000048 00000000 00000000
00000050 00000000 00000000
00000058 00000000 00000000
00000060 00000000 00000000
00000068 00000000 00000000
00000070 00000000 00000000
00000078 00000000 00000000
00000080 00000000 00000000
00000088 00000000 00000000
00000090 00000000 00000000
00000098 00000000 00000000
000000a0 00000000 00000000
000000a8 00000000 00000000
000000b0 00000000 00000000
000000b8 00000000 00000000
000000c0 00000000 00000000
000000c8 00000000 00000000
000000d0 00000000 00000000
000000d8 00000000 00000000
000000e0 00000000 00000000
000000e8 00000000 00000000
000000f0 00000000 00000000
000000f8 00000000 00000000
00000100 00000000 00000000
00000108 00000000 00000000
00000110 00000000 00000000
00000118 00000000 00000000
00000120 00000000 00000000
00000128 00000000 00000000
00000130 00000000 00000000
00000138 00000000 00000000
00000140 00000000 00000000
00000148 00000000 00000000
00000150 00000000 00000000
00000158 00000000 00000000
00000160 00000000 00000000
00000168 00000000 00000000
00000170 00000000 00000000
00000178 00000000 00000000
00000180 00000000 00000000
00000188 00000000 00000000
00000190 00000000 00000000
00000198 00000000 00000000
000001a0 00000000 00000000
000001a8 00000000 00000000
000001b0 00000000 00000000
000001b8 00000000 00000000
000001c0 00000000 00000000
```

```
CPU
ip   02000fa4
pc   02000fa8
r1   00000000
r2   00000000
r3   00000000
r4   00000000
r5   00000000
r6   00000000
r7   00000000
r8   00000000
r9   00000000
r10  00000000
r11  00000000
r12  00000000
r13  00000000
r14  00000000
r15  00000000
r16  00000000
r17  00000000
r18  00000000
r19  00000000
r20  00000000
r21  00000000
r22  00000000
r23  00000000
r24  00000000
r25  00000000
r26  00000000
r27  00000000
r28  00000000
r29  00000000
r30  00000000
r31  00000000
IE   00000000
FPST00080000
EPC  00000000
EIP  00000000
INTPEN 00000400
CONFIG 00000000
PPERROR000f0000
PKTREQ 00000000
TCOUNT 00000000
TSCALE 00000000
```

```
COMMAND

load sample
Loading sample.out
 474 Symbols loaded
Done
>>>
```

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

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

1) Reset the MP:

   **`reset`** ↵

2) Load the sample program again:

   **`load sample`** ↵

## What’s in the DISASSEMBLY window?

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

This tutorial step demonstrates that the code shown in the DISASSEMBLY window corresponds to memory contents. The MEMORY window displays the current contents of memory. Initially, memory is displayed starting at address 0; if you look at the first line of the DISASSEMBLY window, you’ll see that its display starts with the line containing c_int00.

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

```
mem c_int00 ⏎
```

Notice that the first column in the DISASSEMBLY window corresponds to the addresses in the MEMORY window; the second column in the DISASSEMBLY window corresponds to the memory contents displayed in the MEMORY window.

## *Select the active window*

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

Make the CPU window the active window:

```
win CPU ⏎
```

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

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

**Try This:** Press the F6 key to "cycle" through the windows in the display, making each one active in turn. Press F6 as many times as necessary until the CPU window becomes the active window.

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

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

2) Click the left mouse button.

lesson continues on the next page →

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

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

  Press ESC to get out of this.

- ❑ If you're pointing inside the DISASSEMBLY or FILE window, you'll set a breakpoint on the statement that you are pointing to. Point to the same statement (it's highlighted with a ">" symbol); press the button again to delete the breakpoint.

## *Resize the active window*

This lesson shows you how to resize the active window.

**Important!** Be sure the CPU window is still active.

Make the CPU window as small as possible:

**`size 4,3`** ↵

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

Make the CPU window larger:

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

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

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

↖ 1) If you examine any inactive window, you'll see a highlighted, backwards L in the lower right corner. Point to the lower right corner of the CPU window.

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

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

## *Zoom the active window*

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

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

Make the active window as large as possible:

```
zoom ⏎
```

The window should now be as large as possible, taking up the entire display (except for the menu bar) and hiding all the other windows.

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

```
zoom ⏎
```

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

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

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

Zoom the active window:

↖ 1) Point to the upper left corner of the active window.

2) Click the left mouse button.

Return the window to its previous size by repeating these steps.

## *Move the active window*

This lesson shows you how to move the active window.

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

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

```
move 0,1 ⏎
```

The debugger doesn't let you move the window to the very top—that would hide the menu bar

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

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

```
move ⏎
→→→→
ESC
```

Press → until the CPU window is back where it was (it may seem like only the border is moving—this is normal)

Press ESC when you finish moving the window

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

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

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

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

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

## *Scroll through a window’s contents*

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

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

Scroll through the contents of the DISASSEMBLY window:

↖ 1) Point to the up or down scroll arrow.

▮▯ 2) Press the left mouse button; continue pressing it until the display has scrolled several lines.

▯▯ 3) Release the button.

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

Make the MEMORY window the active window:

**`win MEMORY`** ⏎

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

↓ ↑ PAGE DOWN PAGE UP

These keys don’t work the same for all windows; Section 14.4, *Summary of Special Keys*, summarizes the functions of all the special keys and key sequences and how they affect different windows.

## *Display the C source version of the sample file*

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

Display the contents of a C source file:

```
file shrinksv.c ↵
```

This opens a FILE window that displays the contents of the file shrinksv.c (shrinksv.c was one of the files that contributed to making the sample object file). You can always tell which file you're displaying by the label in the FILE window. Right now, the label should say FILE: shrinksv.c.

## *Execute some code*

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

Execute a portion of the sample program:

```
go main ↵
```

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

- The current IP and PC are highlighted in the DISASSEMBLY window.
- The object code of the first statement in the DISASSEMBLY window is highlighted because these statements is associated with the current C statement (which is highlighted in the FILE window).
- The CALLS window, which tracks functions as they're called, now points to main().
- The values of the IP and PC (and possibly some additional registers) are highlighted in the CPU window because they were changed by program execution.

## *Become familiar with the three debugging modes*

The debugger has three basic debugging modes:

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

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

The following steps show you how to switch debugging modes.

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

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

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

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

▯▯ 4) Release the button.

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

Switch to auto mode:

↖ 1) Point to the word MoDe on the menu bar.

▮▯ 2) Click the left mouse button. This displays and freezes the MoDe menu.

3) Now select C(auto). Choose one of these methods:

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

lesson continues on the next page →

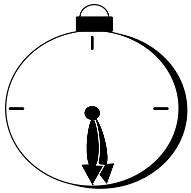
You should be in auto mode now, and you should see the FILE window but not the DISASSEMBLY window (because your program is in C code). Auto mode automatically switches between an assembly and a C display, depending on where you are in your program.

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

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

Switch back to mixed mode before continuing:

```
mix ↵
```



## Halfway Point

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

If you want to close the debugger, just type QUIT ↵. When you come back, reinvoke the debugger and load the sample program (page DB:2-4). Then turn to page DB:2-17 and continue with the second set of lessons.

## *Open another text file, then redisplay a C source file*

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

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

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

```
file init.cmd ↵
```

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

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

Redisplay another C source file (main.c):

```
func main ↵
```

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

## *Use the basic RUN command*

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

Run your entire program:

**`run`** ↵

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

This isn’t very exciting; halt program execution:

ESC

## *Set some breakpoints*

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

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

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

```
file main.c ↵
```

Set a software breakpoint and run your program:

1) With the FILE window active, scroll to statement in the FILE window that contains the **TaskInitTasking()** function and set a software breakpoint at that line:

   ↖ a) Point the mouse cursor at the **TaskInitTasking()** statement.

   🖱 b) Click the left mouse button. Notice how the line is prefixed with the > character; this identifies a software breakpointed statement.

2) Reset the program entry point:

   ```
   restart ↵
   ```

3) Enter the run command:

   ```
   run ↵
   ```

   Program execution halts at the breakpoint

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

Clear the breakpoint:

↖ 1) Point the mouse cursor at the **TaskInitTasking()** statement. (The line should still have a > prefix from setting the breakpoint.)

🖱 2) Click the left mouse button. The line is no longer prefixed.

## *Benchmark a section of code*

You can use breakpoints to help you benchmark a section of code. This means that you'll ask the debugger to count the number of CPU clock cycles that are consumed by a certain portion of code.

Benchmark some code:

1) In main.c (displayed in the FILE window), set two software breakpoints: one at the statement in the FILE window that contains the **TaskInitTasking()** function and one at the statement that contains the **TaskInstallMalloc(myMalloc, my Free)** function.

2) Reset the program entry point:

```
restart ↵
```

3) Enter the run command:

```
run ↵
```

This runs to the first breakpoint

4) Enter the runb command:

```
runb ↵
```

This runs to the second breakpoint
(this may take several seconds)

5) Now use the ? command to examine the contents of the CLK pseudo-register:

```
? clk ↵
```

The debugger now shows a number in the display area; this is the number of CPU clock cycles consumed by the portion of code between the two breakpointed C statements.

**Important!** The value in the CLK pseudoregister is valid **only** when you execute the RUNB command and when that execution is halted on breakpointed statements.

Delete both software breakpoints:

```
br ↵
```

The BR (breakpoint reset) command deletes
all software breakpoints that were set

## *Watch some values and single-step through code*

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

Set up for the single-step example:

1) Display the contents of shrinksv.c in the FILE window:

```
file shrinksv.c   ↵
```

2) Set a breakpoint at the first line that contains the **for (i=0; i<4; i++)** statement.

3) Restart and run to the breakpoint:

```
restart ↵
run ↵
```

4) Delete the breakpoint:

```
br ↵
```

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

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

```
wa ip, Instruction Pointer, x ↵
wa pp ↵
wa cmdBuf ↵
wa i ↵
```

You may have noticed that the WA (watch add) command can have one, two, or three parameters. The first parameter is the item that you're watching. The second parameter is an optional label. The third parameter is the display format.

lesson continues on the next page →

If the WATCH window isn’t wide enough to display the IP value, resize the window. Now try out the single-step commands.

Single-step through the sample program:

```
step 20 ↵
```

**Try This:** Notice that the STEP command single-stepped each assembly language statement (in fact, you single-stepped through 20 assembly language statements). Did you also notice that the FILE window displayed the lines of code for the **PP = &(((SRVARG *)arg)–>pp[i])** function when it was called? The debugger supports additional single-step commands that have a slightly different flavor.

□ For example, if you enter:

```
cstep 20 ↵
```

you’ll single-step 20 **C statements**, not assembly language statements (notice how the IP “jumps” in the DISASSEMBLY window).

□ Now enter the NEXT command, as shown below. You’ll be single-stepping 20 assembly language statements, but the FILE window won’t display the code line(s) for a function when one is executed.

```
next 20 ↵
```

(There’s also a CNEXT command that “nexts” in terms of C statements.)

## Run code conditionally

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

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

```
file shrinksv.c ⏎
```

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

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

```
wd 3 ⏎
wd 1 ⏎
wd 1 ⏎
```

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

Set up for the conditional run examples:

1) Set software breakpoints at the first line that contains the **for (i=0; i<4; i++)** statement and at the line that contains the **cmdBuf[i] = cmdBuf** statement.

   Did you notice that the WATCH window disappeared when you made the FILE window active to set the breakpoints? It didn't really disappear—it's just hidden behind the FILE window.

2) Bring the WATCH window to the top. Press F6 enough times to make the WATCH window active.

3) Reset the program entry point:

   ```
   restart ⏎
   ```

4) Run to the first breakpoint:

   ```
   run ⏎
   ```

5) Reset the value of i:

   ```
   ?i=0 ⏎
   ```

lesson continues on the next page →

Now initiate the conditional run:

```
run i<3 ↵
```

This causes the debugger to run through the loop as long as the value of i is less than 3. Each time the debugger encounters the breakpoint in the loop, it updates the value of i in the WATCH window.

When the conditional run completes, close the WATCH window.

Close the WATCH window:

```
wr ↵
```

## *WHATIS that?*

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

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

```
whatis i ⏎
      long i;                           i is a long unsigned integer
whatis pp ⏎
      struct {                                   pp is a structure
          enum PPNUM ppNum;
          long semaId;
          void (*program)();
      } *pp;
whatis portId
      long portId;               portId is a long unsigned integer
whatis SRVARG ⏎
      struct {                SRVARG is a long, nested structure
          struct {
              enum PPNUM ppNum
              long semaId
              void (*program)();
          } pp[4];
          long portId;
      } SRVARG;
```

Press ESC to halt long listings. You can also scroll the COMMAND window or use the ZOOM command to view the longer listings.

## *Clear the COMMAND window display area*

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

Clear the COMMAND window display area:

```
cls ⏎
```

## *Change some values*

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

Change a value in memory:

1) Display memory beginning with address 0x8040:

   **`mem 0x8040`** ↵

↖ 2) Point to the contents of memory location 0x8040.

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

4) Type ffffffff.

5) Press ↵ to enter the new value.

6) Press ESC to conclude editing.

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

1) Make the CPU window the active window:

   **`win CPU`** ↵

↑ ↓ 2) When you make the CPU window the active window, the field cursor ( _ ) should be pointing to the IP contents. If it's not, press the arrow keys until the cursor points to the IP contents.

F9 3) Press F9 .

4) Type 020001fc.

↓ 5) Press ↓ enough times to point at the contents of register R5.

6) Type ffffffff.

↵ 7) Press ↵ to enter the new value.

ESC 8) Press ESC to conclude editing.

## *Display a data-flow plot*

The MVP simulator allows you to observe the running session’s memory access patterns and status information. The **data-flow plot** feature provides a cycle-by-cycle visual representation of the processor status and memory accesses.

Set up for the data-flow plot example:

1) Remove all software breakpoints:

   **`br`** ↵

2) Reset the program entry point:

   **`restart`** ↵

3) Execute a portion of the sample program:

   **`go main`** ↵

Bring up a data-flow plot:

**`dataplot`** ↵

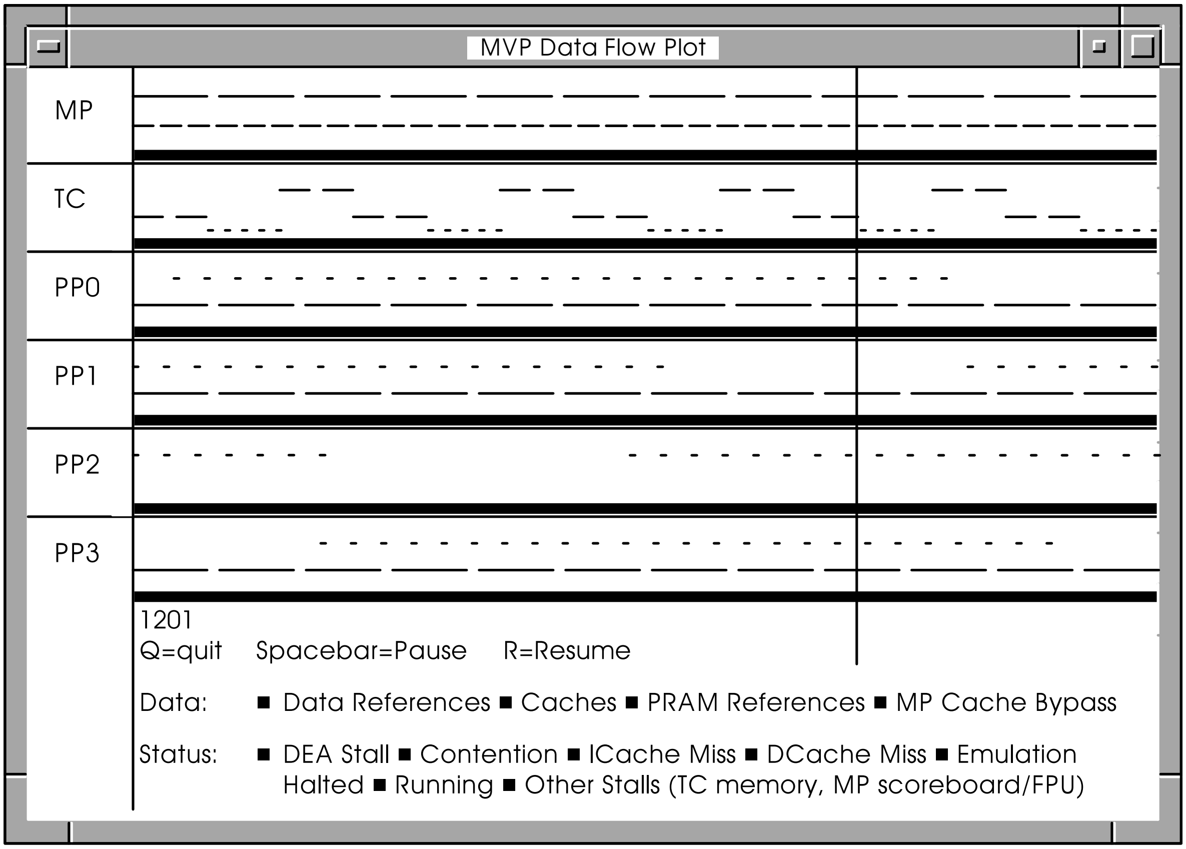
This command causes the debugger to display the MVP Data Flow Plot window. Right now, the window is blank. That’s because code must be executing for there to be anything meaningful shown in the window.

Execute some code. Move your cursor over the MP debugger window and press F5.

Pressing F5 has the same effect as entering RUN from the command line.

The data-flow plot provides a cycle-by-cycle display of processor status and memory accesses. The data-flow plot that you see should look similar to the following:

lesson continues on the next page →



1201
Q=quit Spacebar=Pause R=Resume

Data: ■ Data References ■ Caches ■ PRAM References ■ MP Cache Bypass

Status: ■ DEA Stall ■ Contention ■ ICache Miss ■ DCache Miss ■ Emulation
Halted ■ Running ■ Other Stalls (TC memory, MP scoreboard/FPU)

**Notes:** 1) Although this example is shown in black and white, the data-flow plot is designed for use on a color monitor.

2) Your window label and borders may look different from the illustration, depending on which window manager you're using under the X Window System.

Watch the data-flow plot as it shows the memory access patterns of the on-chip memory.

To close the MVP Data Flow Plot window, move the cursor over the window and press Q.

## *Define a memory map*

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

View the default memory map settings:

```
ml ↵
```

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

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

Change the memory map:

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

   ```
   md 0x0 ↵
   ```

   This deletes the block of memory beginning at address 0.

2) Use the MA (memory add) command to define two new blocks of memory:

   ```
   ma 0x0,0x800,ROM ↵
   ma 0x800,0x2000,RAM ↵
   ```

## *Define your own command string*

You will find it helpful to have a shorthand method for entering:

- ❑ A command that often has the same parameters
- ❑ The same commands in a particular sequence

The debugger provides an **aliasing** feature that allows you to do this.

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

Define an alias for setting up the memory map:

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

```
alias mymap,”md 0x0;md 0x800;ma 0x0,0x800,ROM;
             ma 0x800,0x2000,RAM;ml” ↵
```

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

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

```
mymap ↵
```

This is equivalent to entering the following five commands:

```
md 0x0
md 0x800
ma 0x0,0x800,ROM
ma 0x800,0x2000,RAM
ml
```

## *Close the debugger*

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

Close the debugger and return to the shell:

```
quit ↵
```

Chapter 3

# Advanced Tutorial: Using the MVP Multiprocessing Features

This tutorial demonstrates of the MVP's multiprocessing capabilities. You can perform the tutorial only on the simulator version of the debuggers. Before you use this tutorial, you should complete the basic debugger tutorial in Chapter 2, *An Introductory Tutorial to the C Source Debugger*.

## *A note about entering commands*

In this tutorial, you'll be invoking the parallel debug manager and multiple debuggers. Unless otherwise noted, enter all commands from the parallel debug manager command shell window.

## *Invoke the parallel debug manager and individual debuggers*

Before you can invoke multiple debuggers in a multiprocessing environment, you must first invoke the parallel debug manager.

**Important!** This tutorial assumes that you have correctly and completely installed your parallel debug manager, debugger, and debugging tools (including invoking any files or system commands as instructed in the installation guide). However, if you set the D_DIR environment variable, you must unset it for this tutorial:

```
unsetenv D_DIR  ⏎
```

From a command shell window, enter these commands:

1) Change to the demo directory:

```
cd /your_pathname/mvp/demo  ⏎
```

2) Upon invocation, the parallel debug manager looks for a file called init.pdm. For installation purposes, the init.pdm file in the demo directory contains special scripts that automatically invoke a debugger with the parallel debug manager. Since you don't want to do this for the tutorial, rename the init.pdm file:

```
mv init.pdm oldinit.pdm  ⏎
```

3) Invoke the parallel debug manager:

```
pdm  ⏎
```

After you invoke the parallel debug manager, you'll see messages like these in the command shell window:

```
Parallel Debug Manager  Version 1.1
Copyright (c) 1992,1994  Texas Instruments Incorporated

Cannot open take file

PDM:1>>
```

The prompt, PDM:1>>, tells you that the parallel debug manager is ready to accept commands. The message **Cannot open take file** means that the parallel debug manager could not find the init.pdm file (because you renamed the file).

lesson continues on the next page →

Once you have invoked the parallel debug manager, you can invoke multiple debuggers that will be controlled by the parallel debug manager.

When you first begin an MVP debugging session, the PPs are in a halted mode and are not ready to accept commands. They are, in effect, “asleep” until the MP executes a portion of code that “awakens” them. The first step of this lesson brings up the MP debugger and executes the portion of code that wakes up the PPs.

1) Invoke a debugger for the MP by using the SPAWN command:

```
spawn mpsim ↵
```

   You should see an additional command shell that contains the MP debugger.

2) Load the sample program’s object code into the the MP:

```
send –g MP load sample ↵
```

   The SEND command allows you to send actual debugger commands to a processor or group of processors.

3) Set a breakpoint in the MP code past the point at which the PPs are “awakened”:

```
send –g MP ba DummyClient ↵
```

4) Run code until the breakpoint:

```
send –g MP run ↵
```

   It will take several seconds to reach the breakpoint.

lesson continues on the next page →

Once the RUN command has completed, the PPs are “awakened” and ready to accept commands, and you can invoke a debugger for each PP.

1) Invoke a debugger for PP0:

   ```
   spawn ppsim –n PP0 ↵
   ```

2) Invoke a debugger for PP1:

   ```
   spawn ppsim –n PP1 ↵
   ```

3) Invoke a debugger for PP2:

   ```
   spawn ppsim –n PP2 ↵
   ```

4) Invoke a debugger for PP3:

   ```
   spawn ppsim –n PP3 ↵
   ```

The –n option allows you to provide a name that corresponds to the name of a processor. Notice that you didn’t need to specify a name for the MP debugger in the previous step; when you invoke the MP debugger, the name defaults to MP.

You should now see six command shell windows: one that contains the parallel debug manager, one that contains the MP debugger, and four that contain each of the PP debuggers that you invoked.

## *Create groups of processors*

Now that you've invoked the debuggers, you're ready to start sending commands to the processors. But what if you want to send commands to more than one processor at a time? To do so, you'll need to set up groups of processors that you can send commands to.

You're going to create three groups:

- ❑ Group ONE will contain the MP and PP0.
- ❑ Group PPS will contain the four PPs: PP0, PP1, PP2, and PP3.
- ❑ Group ALL will contain the MP and all four PPs.

Create the ONE and PPS groups by entering the following commands:

```
set ONE = MP PP0 ↵
set PPS = PP0 PP1 PP2 PP3 ↵
```

Now that you've created these groups, you can make the ALL group by basing it on the PPS group:

```
set ALL = MP $PPS ↵
```

The $ tells the parallel debug manager to use the processors listed previously in the PPS group (PP0, PP1, PP2, and PP3) as part of the new list of processors.

Now that you've created these groups, you may want to send numerous commands to a particular group. Most of the commands that you can send to groups require a group name, or they use the default group. By setting a default group, you can avoid having to type a group name numerous times.

Make the ALL group the default group:

```
set dgroup = $ALL ↵
```

*dgroup* is a special group name that the parallel debug manager recognizes. The parallel debug manager looks for this group if you don't specify a group name when you enter a command.

lesson continues on the next page →

**Important!** Did you include the dollar sign ($) when you set up the default group? If you omitted it, you've just set up a group that contains ALL instead of the processors that are contained in the ALL group. To correctly set up the default group, re-enter the SET command as listed above.

**Try This:** To verify that you've correctly created these groups, use the SET command without any parameters:

**`set`** ↵

You should see the following in the display area of the parallel debug manager:

```
ALL    ″MP PP0 PP1 PP2 PP3″
ONE    ″MP PP0″
PPS    ″PP0 PP1 PP2 PP3″
dgroup ″MP PP0 PP1 PP2 PP3″
```

## *Load the sample program’s symbol table*

In the first lesson, you loaded the sample program’s object code into the simulator core when you performed the LOAD command for the MP. However, the PPs must know what the symbol table looks like before they can execute code. One way that you can load the symbol table is from within each individual debugger. However, this method can be time-consuming, especially if you’re loading the same code into more than one debugger.

From the parallel debug manager, you can send actual debugger commands to a processor or group of processors. This allows you to send an SLOAD (load symbol table) command to a group of processors.

Load the sample program’s symbol table into the debuggers contained in the PPS group:

```
send –g PPS sload sample ↵
```

Notice that the messages that are displayed in the COMMAND window of the individual debuggers are also shown in the display area of the parallel debug manager.

## *Execute some code in parallel*

The most powerful aspect of a multiprocessing system is the ability to execute code in parallel. In this step, you'll execute some code on the five processors.

**Important!** Be sure that the default group contains all five processors (MP, PP0, PP1, PP2, and PP3). To verify that the default group (dgroup) is set properly, use the SET command:

```
set dgroup ↵
```

Recall from the basic tutorial in Chapter 2 that you can view the memory access patterns and status information of processors that are running code. In this lesson, you'll bring up a data-flow plot to watch the processors execute in parallel.

Bring up a data-flow plot:

```
send –g MP dataplot ↵
```

This command causes the MP debugger to display the MVP Data Flow Plot window. Right now, the window is blank. That's because code must be executing for there to be anything meaningful shown in the window.

You need to do one more thing before you start to execute in parallel: set a common breakpoint in the PP code. To do so, you must set the breakpoint for each individual PP.

Set a breakpoint in common PP code:

```
send –g PPS ba _ShrinkImage↵
```

For each PP, this command sets a breakpoint on the line that contains the ShrinkImage function. Now that you've set a stopping point, run some code.

Use the global run command:

```
prun –r ↵
```

The –r option returns the parallel debug manager prompt immediately after you enter the command, even though the debuggers are still running.

lesson continues on the next page →

The PRUN command synchronizes the debuggers to cause the processors to begin execution at the same real time. Watch the data flow plot window. Notice that the PPs are no longer idle—they're quite active. Also watch to see the "Running" line for the PPs turn from black to gray—that means that the PP has reached the breakpoint and emulation has halted.

Close the data flow plot window before continuing by moving the cursor over the window and pressing Q .

Although the PPs halted when they reached the common breakpoint, the MP continues to run.

Halt the MP:

**phalt** ↵

The PHALT command halts execution of each processor at the same real time.

You can see where the processors are in the code by checking the execution status of the processors:

**stat** ↵

The STAT command tells you whether a processor is running or halted. If a processor is halted when you execute this command, then the parallel debug manager also lists a PC value. This value is the address of the next instruction to be executed (current IP value for the MP or current IPE value for the PP). When you executed STAT after the PHALT command, you should have seen something like this in the display area of the parallel debug manager:

```
[MP] Halted PC=2001568
[PP0] Halted PC=2003078 at breakpoint
[PP1] Halted PC=2003078 at breakpoint
[PP2] Halted PC=2003078 at breakpoint
[PP3] Halted PC=2003078 at breakpoint
```

**Important!** Do not confuse the PC value shown in the parallel debug manager display with the PC registers in either the MP or PP CPU window. The PC value is a parallel debug manager variable that holds the current IP value (for the MP) or the current IPE value (for a PP).

## *Execute code and watch an image develop*

This lesson shows you how to view an image that develops as you run common PP code.

Before continuing, make the PPS group the default group:

**`set dgroup = $PPS`** ↵

Click on the border of the MP window to make it the top window. From the MP window, enter:

**`image`** ↵

In the dialog box that appears in the MP debugger window, enter the appropriate parameters:

↖ 1) Point to the parameter field next to “Start Address”.

🖱 2) Click the left mouse button. This selects the field to identify it as the field that will be edited.

3) Type DrillOut.

TAB 4) Press TAB to move to the next field.

5) Type 256.

TAB 6) Press TAB to move to the next field.

7) Type 240.

TAB 8) Press TAB to move to the next field.

9) Type 8.

🖱 10) Point to the <OK> field and click the left mouse button. This executes your choice and closes the dialog box.

lesson continues on the next page →

As the dialog box disappears, you will see the MVP Image Display window. The window is empty since you’re not running any PP code.

Run PP code in parallel:

```
prun –r ↵
```

Watch the Image Display window as the PPs execute the code. You’ll see the image begin to develop. Move the cursor into the MVP Image Display window. The screen will change colors when you move the mouse cursor into the window, and the partially completed grayscale image will be visible.

Before the image fully develops (or when you’re tired of watching the image), close the window that contains the image by moving the cursor over the window and pressing Q .

Halt execution:

```
phalt ↵
```

## Single-step in parallel

In addition to running in parallel, the parallel debug manager allows you to single-step in parallel through assembly language code with interrupts disabled.

**Important!** Be sure that the default group contains PP0, PP1, PP2, and PP3.

Start single-stepping in parallel:

```
pstep 20 ↵
```

The PSTEP command causes the processors to synchronously single-step 20 times.

**Try This:** The parallel debug manager supports a command history that is similar to the UNIX command history. Re-execute the PSTEP command that you just entered:

```
!! ↵
```

As in UNIX, the !! command tells the parallel debug manager to repeat the last command that you entered. Notice that the parallel debug manager prompt has a number appended to it. You can use the !*number* command to repeat the command that you entered at parallel debug manager prompt *number.*

## *Gather information from the PPs*

The parallel debug manager is useful in helping you gather information from the individual processors. You’ve been running quite a bit of code on the PPs. You might like to know how much code is left for the PPs to execute. The PP register d6 tells you how many lines of code are left for the PP to execute. You can use the SEND command to ask the PPs for the d6 register value.

**Important!** Be sure that the default group contains the PP0, PP1, PP2, and PP3.

Enter the SEND command:

```
send ? d6 ↵
```

The results of the command are shown in the COMMAND window of each individual debugger. The results are also shown by processor in the display area of the parallel debug manager. You should see something like this:

```
[PP0] 27
[PP1] 26
[PP2] 26
[PP3] 27
```

## *Define your own command string*

Recall from the basic debugger tutorial in Chapter 2 that the debugger has a shorthand method for entering frequently used command sequences. This method is called aliasing.

The parallel debug manager has an aliasing feature that is similar to that of the debugger. Aliasing is useful for creating parallel debug manager versions of debugger commands. For example, you may find it useful to set up a parallel debug manager version of the ML command to list the memory map of a group of processors.

Use the ALIAS command to define a parallel debug manager alias for the ML (memory list) command:

```
alias ml, ”send –g %1 ml” ↵
```

This command tells the parallel debug manager to create an alias called ML that, upon execution, will send the debugger ML command to a group you specify.

Test out your new alias:

```
ml ONE ↵
```

You should see the memory maps for the MP and PP0 debuggers in the display area of the parallel debug manager.

## Create a batch file for repetitive tasks

Recall from the first two lessons in this chapter that when you first set up your multiprocessing environment (for example, invoking and naming multiple debuggers, grouping processors, and setting the default group), you had to enter numerous commands. If you often debug code using the same group or groups of processors, you may want to put the SPAWN, SEND, and SET commands in a batch file.

This lesson shows you how to create a batch file by using the parallel debug manager's DLOG command. The DLOG command records the commands that you enter from the parallel debug manager command line into a batch (or log) file. In this lesson, the batch file that you create can serve as the parallel debug manager's initialization batch file (init.pdm) that the parallel debug manager automatically looks for during invocation.

1) Since you're going to actually execute the commands that you're logging, you need to quit out of the current parallel debug manager session. From the parallel debug manager command line, enter:

   ```
   quit ↵
   ```

   This command quits the parallel debug manager and each of the debuggers that you invoked from the parallel debug manager.

2) From your command shell window, re-invoke the parallel debug manager:

   ```
   pdm ↵
   ```

3) Tell the parallel debug manager to log the next few commands that you enter:

   ```
   dlog init.pdm  ↵
   ```

   *init.pdm* is the name of the file that the commands are recorded in.

lesson continues on the next page →

4) Invoke the MP debugger and wake up the PPs:

```
spawn mpsim  ↵
send –g MP load sample ↵
send –g MP ba DummyClient ↵
send –g MP run ↵
```

5) Enter the SPAWN commands to invoke the PP debuggers:

```
spawn ppsim –n PP0 ↵
spawn ppsim –n PP1 ↵
spawn ppsim –n PP2 ↵
spawn ppsim –n PP3 ↵
```

6) Enter the SET commands to group your processors:

```
set ONE = MP PP0 ↵
set PPS = PP0 PP1 PP2 PP3 ↵
set ALL = MP $PPS ↵
```

7) Load the symbol table into the PP debuggers:

```
send –g PPS sload sample ↵
```

8) Tell the parallel debug manager to stop logging your commands:

```
dlog close ↵
```

Now that you've created the init.pdm file in the current directory, upon invocation, the parallel debug manager will automatically execute the commands inside the file and set up your multiprocessing environment.

## *Exit the parallel debug manager and debuggers*

You can close the parallel debug manager and/or debuggers by using the QUIT command. You can enter this command from one of the individual debuggers or from the parallel debug manager.

You can also close the parallel debug manager or debuggers by closing the operating-system window that contains one of these processes.

Close the MP debugger:

↖ 1) Point your mouse cursor on the border of the window that contains the MP debugger.

🖱 2) Execute Quit or Close from the window-manager menu.

Close the parallel debug manager and the PP debuggers by entering the following from the command line of the parallel debug manager:

```
quit ↵
```

Chapter 4

# Using the Parallel Debug Manager

The MVP debugging system is a true multiprocessing debugging system. It allows you to debug your entire application, including all MP and PP portions, by using the parallel debug manager. The parallel debug manager is a command shell that controls and coordinates multiple debuggers. This chapter describes the functions that you can perform with the parallel debug manager.

For information about invoking the parallel debug manager and debuggers, see Section 1.5, *Invoking the Debuggers and the Parallel Debug Manager*.

**Topics**

## 4.1 Identifying Processors and Groups

You can send commands to an individual processor or to a group of processors. To do this, you must assign names to the individual processors or to groups of processors.

- ❑ Individual processor names are assigned when you invoke the individual debuggers.
- ❑ You can assign group names with the SET command after the individual processor names have been assigned.

**Note:**

Each debugger that runs under the parallel debug manager must have a unique processor name. The parallel debug manager does not keep track of existing processor names. When you send a command to a debugger, the parallel debug manager will validate the existence of a debugger invoked with that processor name.

### *Assigning names to individual processors*

You must associate each debugger within the multiprocessing system with a unique name, referred to as a **processor name**. The processor name is used for:

- Identifying a processor to send commands to
- Assigning a processor to a group
- Setting the default prompts for the associated debuggers. For example, if you invoke a debugger with a processor name of MVP1_PP0, that debugger's prompt will be MVP1_PP0>.
- Identifying the individual debuggers on the screen. The processor name that you assign will appear at the top of the operating-system window that contains the debugger. Additionally, if you turn one of the windows into an icon, the icon name is the same as the processor name that you assigned.

To assign a processor name, you can use the –n option when you invoke a debugger. For example, to name the simulator version of PP1, use the following command to invoke the debugger:

```
spawn ppemu –n MVP1_PP1 ↵
```

From this point on, whenever you needed to identify the debugger for the PP1, you could identify it by its processor name, **PP1***.*

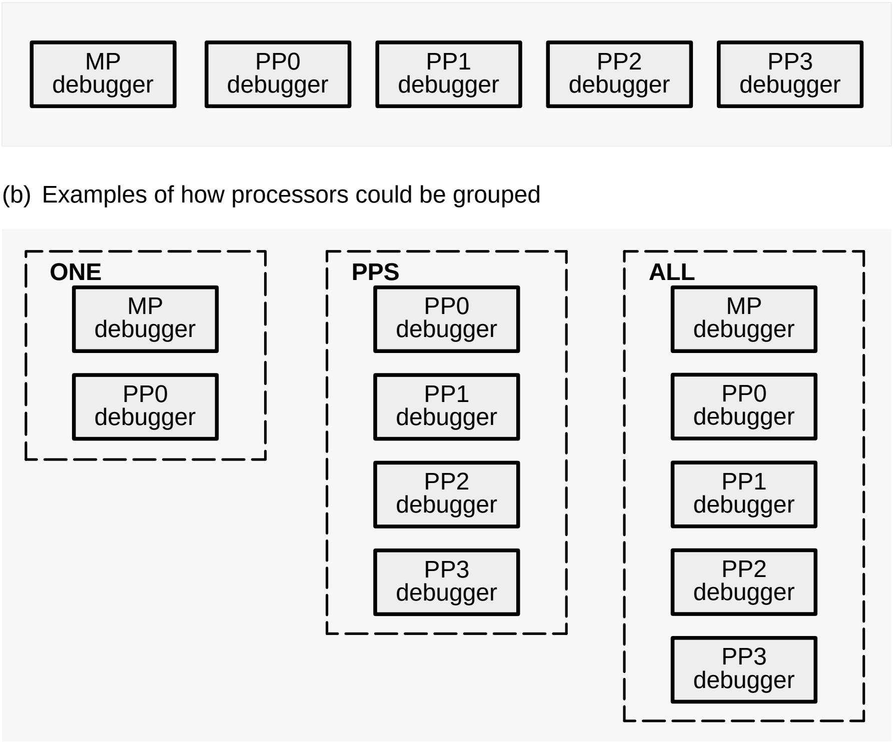
The processor name that you supply can consist of up to eight alphanumeric or underscore characters and must begin with an alphabetic character. Note that the name is not case sensitive.

For more information about processor names, refer to the invocation instructions in Section 1.5, *Invoking the Debuggers and the Parallel Debug Manager.*

## *Organizing processors into groups*

Processors can be organized into groups; these groups are identified by names defined with the SET command. Each processor can belong to any group, all groups, or a group of its own. Figure 4–1 (a) shows an example of all the processors that could exist in a system, and Figure 4–1 (b) illustrates three examples of named groups. Group ONE contains the MP and PP0, group PPS contains all the PPs, and group ALL contains the MP and all PPs.

Figure 4–1. Grouping Processors



(a) All possible processors in a system

(b) Examples of how processors could be grouped

To define and manipulate software groupings of named processors, use the SET and UNSET commands.

❑ **Defining a group of processors**

To define a group, use the SET command. The format for this command is:

**set** [*group name* [**=** *list of processor names*] ]

This command allows you to specify a group name and the list of processors you want in the group. The *group name* can consist of up to 128 alphanumeric or underscore characters.

For example, to create the ONE group illustrated in Figure 4–1 (b), you could enter the following on the command line of the parallel debug manager:

```
set ONE = MVP1_MP MVP1_PP0 ↵
```

The result is a group called ONE that contains MVP1_MP and MVP1_PP0. Note that the order in which you add processors to a group is the same order in which commands will be sent to the members of that group.

❑ **Setting the default group**

Many of the parallel debug manager (PDM) commands can be sent to groups; if you often send commands to the same group and you want to avoid typing the group name each time, you can assign a default group.

To set the default group, use the SET command with a special group name called **dgroup**. For example, if you want the default group to contain the MP and three PPs, enter:

```
set dgroup = MVP1_MP MVP1_PP0 MVP1_PP1 MVP1_PP2 ↵
```

The parallel debug manager will automatically send commands to the default group when you don’t specify a group name.

### □ Modifying an existing group or creating a group based on another group

Once you've created a group, you can add processors to it by using the SET command and preceding the existing *group name* with a dollar sign ($) in the list of processors. You can also use a group as part of another group by preceding the existing group's name with a dollar sign. The dollar sign tells the parallel debug manager to use the processors listed previously in the group as part of the new list of processors.

Suppose GROUPA contains the MVP1_MP and MVP1_PP0. If you wanted to add two more PPs to the group, you'd enter:

```
set GROUPA = $GROUPA MVP1_PP1 MVP1_PP2 ⏎
```

After entering this command, GROUPA would contain MVP1_MP, MVP1_PP0, MVP1_PP1, and MVP1_PP2.

If you decided to send numerous commands to GROUPA, you could make it the default group:

```
set dgroup = $GROUPA ⏎
```

### □ Listing all groups of processors

To list all groups of processors in the system, use the SET command without any parameters:

```
set ⏎
```

The parallel debug manager lists all of the groups and the processors associated with them:

```
GROUPA  ″MVP1_MP MVP1_PP0 MVP1_PP1 MVP1_PP2″
ONE     ″MVP1_MP MVP1_PP0″
dgroup  ″MVP1_MP MVP1_PP0 MVP1_PP1 MVP1_PP2″
```

You can also list all of the processors associated with a particular group by supplying a group name:

```
set dgroup ⏎
dgroup  ″MVP1_MP MVP1_PP0 MVP1_PP1 MVP1_PP2″
```

- **Deleting a group**

  To delete a group, use the UNSET command. The format for this command is:

  **unset** *group name*

  You can use this command in conjunction with the SET command to remove a particular processor from a group. For example, suppose GROUPB contained MVP1_MP, MVP1_PP0, MVP1_PP1, and MVP1_PP2. If you wanted to remove MVP1_PP2, you could enter:

  ```
  unset GROUPB ↵
  set GROUPB = MVP1_MP MVP1_PP0 MVP1_PP1 ↵
  ```

  If you want to delete all of the groups you have created, use the UNSET command with an asterisk instead of a group name:

  ```
  unset * ↵
  ```

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

**Note:**

When you use UNSET * to delete all of your groups, the default group (dgroup) is also deleted. As a result, if you issue a command such as PRUN and don’t specify a group or processor, the command will fail because the parallel debug manager can’t find the default group name (dgroup).

## 4.2 Sending Debugger Commands to One or More Debuggers

The SEND command sends a debugger command to an individual processor or to a group of processors. The command is sent directly to the command interpreter of each individual debugger. You can send any valid debugger command string.

The syntax for the SEND command is:

**send** [**–r**] [**–g** {*group* **|** *processor name*}] *debugger command*

- The **–g** option specifies the group or processor that the debugger command should be sent to. If you don't use this option, the command is sent to the default group (dgroup).
- The **–r** (return) option determines when control returns to the command line of the parallel debug manager:
  - **Without –r**, control is not returned to the command line until each debugger in the group finishes running code. Any results that are printed in the COMMAND window of the individual debuggers are also echoed in the display area of the parallel debug manager. These results are displayed by processor. For example:

    ```
    send ?pc ↵
    [MVP1_MP]  0x40000000
    [MVP1_PP2]  0x40000008
    ```

    If you want to break out of a synchronous command and regain control of the command line, press CONTROL C while your cursor is over the parallel debug manager window. This returns control to the command line. However, no debugger executing the command is interrupted.
  - **With –r**, control is returned to the command line immediately, even if a debugger is still executing a command. When you use –r, you **do not** see the results of the commands that the debuggers are executing.

The –r option is useful when you want to exit from a debugger but not from the parallel debug manager. When you send the QUIT command to a debugger or group of debuggers without using the –r command, you will not be able to enter another PDM command until all debuggers that QUIT was sent to finish quitting; the parallel debug manager waits for a response from all of the debuggers that are quitting. By using –r, you can gain immediate control of the parallel debug manager and continue sending commands to the remaining debuggers.

The SEND command is useful for loading a common object file into a group of debuggers. For example, to load a file called test.out into the debuggers contained in GROUPA, you could use the following command:

```
send –g GROUPA load test.out ↵
```

## 4.3 Running and Halting Code

The PRUNF, PRUN, and PSTEP commands synchronize the debuggers to cause the processors to begin execution at the same real time.

- ❑ PRUNF starts the processors running freely, which means they are disconnected from the emulator.
- ❑ PRUN starts the processors running. If you're using the emulator, this command starts the processors running under the control of the emulator.
- ❑ PSTEP causes the processors to single-step synchronously through assembly language code with interrupts disabled.

The formats for these commands are:

**prunf** [**–g** {*group* **|** *processor name*}]

**prun** [**–r**] [**–g** {*group* **|** *processor name*}]

**pstep** [**–g** {*group* **|** *processor name*}] [*count*]

- ❑ The **–g** option identifies the group or processor that the command should be sent to. If you don't use this option, the command is sent to the default group (dgroup).
- ❑ The **–r** (return) option for the PRUN command determines when control returns to the command line of the parallel debug manager:
  - ■ **Without –r**, control is not returned to the command line until each debugger in the group finishes running code. If you want to break out of a synchronous command and regain control of the command line, press CONTROL C while your cursor is over the parallel debug manager window. This returns control to the command line. However, no debugger executing the command is interrupted.
  - ■ **With –r**, control is returned to the command line immediately, even if a debugger is still executing a command. You can type new commands, but the processors can't execute the commands until they finish with the current command; however, you can perform PHALT, PESC, and STAT commands when the processors are still executing.
- ❑ You can specify a *count* for the PSTEP command so that each processor in the group steps for *count* number of times.

**Note:**

1) If the current statement that a processor is pointing to has a breakpoint, that processor does not step synchronously with the other processors (it takes the breakpoint) when you use the PSTEP command. However, that processor can still single-step.

2) See Section A.1, *Execution Considerations*, for discussions on controlling pipeline execution, managing interrupts while single-stepping, and understanding differences between RUN and STEP commands.

### *Halting processors at the same time*

You can use the PHALT command after you enter a PRUNF command to stop an individual processor or a group of processors (global halt). Each processor in the group is halted at the same real time. The syntax for the PHALT command is:

**phalt** [**–g** {*group* **|** *processor name*}]

See Section A.2, *Halting Considerations*, for more information on halting processors.

### *Sending ESCAPE to all processors*

Use the PESC command to send the escape key to an individual processor or to a group of processors after you execute a PRUN command. Entering PESC is like typing an escape key in all of the individual debuggers. However, the PESC command is **asynchronous**; the processors don’t halt at the same real time. When you halt a group of processors, the individual processors are halted in the order in which they were added to the group.

The syntax for this command is:

**pesc** [**–g** {*group* **|** *processor name*}]

## *Finding the execution status of a processor or a group of processors*

The STAT command tells you whether a processor is running or halted. If a processor is halted when you execute this command, then the parallel debug manager also lists a PC value. This value is the address of the next instruction to be executed (current IP value for the MP or current IPE value for the PP). The syntax for the command is:

**stat** [**–g** {*group* **|** *processor name*}]

For example, to find the execution status of all of the processors in group ONE after you've executed a global PRUN, enter:

```
stat –g ONE ↵
```

After entering this command, you'll see something similar to this in the parallel debug manager window:

```
[MVP1_MP]  Running
[MVP1_PP0] Halted   PC=200001A
```

**Note:**

Do not confuse the PC value shown in the parallel debug manager display with the PC registers in either the MP or PP CPU window. The PC value is a parallel debug manager variable that holds the current IP value (for the MP) or the current IPE value (for a PP).

## 4.4 Entering PDM Commands

The parallel debug manager provides a flexible command-entry interface that allows you to:

- ❑ Execute parallel debug manager (PDM) commands from a batch file
- ❑ Record the information shown in the display area of the parallel debug manager
- ❑ Conditionally execute or loop through PDM commands
- ❑ Echo strings to the display area of the parallel debug manager
- ❑ Pause command execution
- ❑ Repeat previously entered commands (use the command history)

This section describes the PDM commands that you can use to perform these tasks.

## *Executing PDM commands from a batch file*

The TAKE command tells the parallel debug manager to execute commands from a batch file. The syntax for the parallel debug manager version of this command is:

**take** *batch filename*

The *batch filename* **must** have a .pdm extension for the parallel debug manager to be able to read the file. If you don't supply a pathname as part of the filename, the parallel debug manager first looks for *batch filename*.pdm in the current directory and then searches directories named with the D_DIR environment variable.

The TAKE command is similar to the debugger version of this command (described on page DB:6-18). However, there are some differences when you enter TAKE as a PDM command instead of a debugger command.

- **Similarities.** As with the debugger version of the TAKE command, you can nest batch files up to 10 deep.
- **Differences.** Unlike the debugger version of the TAKE command:
  - There is no suppress-echo-flag parameter. Therefore, all command output is echoed to the parallel debug manager window, and this behavior cannot be changed.
  - To halt batch-file execution, you must press CONTROL C instead of ESC.
  - The batch file must contain only PDM commands (no debugger commands).

The TAKE command is advantageous for executing a batch file in which you have defined often-used aliases. Additionally, you can use the SET command in a batch file to set up group configurations that you use frequently, and then execute that file with the TAKE command. You can also put your flow-control commands (described on page DB:4-16) in a batch file and execute the file with the TAKE command.

### *Recording information from the parallel debug manager display area*

By using the DLOG command, you can record the information shown in the display area of the parallel debug manager into a log file. This command is identical to the debugger DLOG command described on page DB:6-6.

- To begin recording the information shown in the display area of the parallel debug manager, use:

  **dlog** *filename*

  This command opens a log file called *filename* that the information is recorded into. If you plan to execute the log file with the TAKE command, the filename **must** have a .pdm extension.

- To end the recording session, enter:

  **dlog close** ↵

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

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

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

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

## Controlling PDM command execution

You can control the flow of parallel debug manager (PDM) commands in a batch file or interactively. With the IF/ELIF/ELSE/ENDIF or LOOP/BREAK/CONTINUE/ENDLOOP flow-control commands, you can execute PDM commands conditionally or set up a looping situation, respectively.

❑ To make PDM commands execute conditionally, use the IF/ELIF/ELSE/ENDIF commands. The syntax is:

  **if** *expression*<br>
  *PDM commands*<br>
  [**elif** *expression*<br>
  *PDM commands*]<br>
  [**else**<br>
  *PDM commands*]<br>
  **endif**

  - If the expression for the IF results in a nonzero value, the parallel debug manager executes all commands between the IF and ELIF, ELSE, or ENDIF.
  - The ELIF is optional. If the expression for the ELIF results in a nonzero value, the parallel debug manager executes all commands between the ELIF and ELSE or ENDIF.
  - The ELSE is optional. If the expressions for the IF and ELIF (if present) are false (zero), the parallel debug manager executes the commands between the ELSE and ENDIF.

❑ To set up a looping situation to execute PDM commands, use the LOOP/BREAK/CONTINUE/ENDLOOP commands. The syntax is:

  **loop** *Boolean expression*<br>
  *PDM commands*<br>
  [**break**]<br>
  [**continue**]<br>
  **endloop**

The parallel debug manager version of the LOOP command is different from the debugger version of this command (described on page DB:6-20). Instead of accepting any expression, the parallel debug manager version of the LOOP command evaluates only Boolean expressions. If the Boolean expression evaluates to true (1), the parallel debug manager executes all commands between the LOOP and BREAK, CONTINUE, or ENDLOOP. If the Boolean expression evaluates to false (0), the loop is not entered.

- The optional BREAK command allows you to exit the loop without having to reach the ENDLOOP. This is helpful when you are testing a group of processors and want to exit if an error is detected.
- The CONTINUE command, which is also optional, acts as a goto and returns command flow to the enclosing LOOP command. CONTINUE is useful when the part of the loop that follows is complicated, and returning to the top of the loop avoids further nesting.

You can enter the flow-control commands interactively or include the commands in a batch file that is executed by the TAKE command. When you enter LOOP or IF from the command line of the parallel debug manager, a question mark (?) prompts you for the next entry:

```
PDM:11>>if $i > 10
?echo ERROR IN TEST CASE
?endif
ERROR IN TEST CASE

PDM:12>>
```

The parallel debug manager continues to prompt you for input using the ? until you enter ENDIF (for an IF command) or ENDLOOP (for a LOOP command). After you enter ENDIF or ENDLOOP, the parallel debug manager immediately executes the IF or LOOP command.

If you are in the middle of entering a LOOP or IF statement interactively and want to abort it, type CONTROL C.

You can use the IF/ENDIF and LOOP/ENDLOOP commands together to perform a series of tests. For example, within a batch file, you can create a loop like the following (the SET and @ commands are described in Section 4.8):

```
set i = 10                                    Set the counter (i) to 10
loop $i > 0                              Loop while i is greater than 0
  .
  test commands
  .
  if $k > 500                                Test for error condition
    echo ERROR ON TEST CASE 8               Display an error message
  endif
  .
  @ i = $i – 1                                 Decrement the counter
endloop
```

You can record the results of this loop in a log file (see page DB:4-15) to examine which test cases failed during the testing session.

## *Echoing strings to the parallel debug manager display area*

You can display a string in the display area of the parallel debug manager by using the ECHO command. This command is especially useful when you are executing a batch file or running a flow-control command such as IF or LOOP. The syntax for the command is:

**echo** *string*

This displays the *string* in the display area of the parallel debug manager.

You can also use ECHO to show the contents of a system variable (system variables are described in Section 4.8):

```
echo $var_proc1
34
```

The parallel debug manager version of the ECHO command works like the debugger version described on page DB:6-19 does, except that you can use the parallel debug manager version outside of a batch file.

## *Pausing command execution*

Sometimes you may want the parallel debug manager to pause while it’s running a batch file or when it’s executing a flow control command such as LOOP/ENDLOOP. Pausing is especially helpful in debugging the commands in a batch file.

The syntax for the PAUSE command is:

**pause**

When the parallel debug manager reads this command in a batch file or during a flow control command segment, the parallel debug manager stops execution and displays the following message:

```
<< pause – type return >>
```

To continue processing, press ↵.

## *Using the command history*

The parallel debug manager supports a command history that is similar to the UNIX command history. The parallel debug manager prompt identifies the number of the current command. This number is incremented with every command. For example, PDM:12>> indicates that 11 commands have previously been entered, and the parallel debug manager is now ready to accept the twelfth command.

The parallel debug manager command history allows you to re-enter any of the last 20 commands:

□ To repeat the last command that you entered, type:

```
!! ↵
```

□ To repeat any of the last 20 commands, use the following command:

**!***number*

The *number* is the number of the parallel debug manager prompt that contains the command that you want to re-enter. For example,

```
PDM:100>>echo hello ↵
hello
PDM:101>>echo bye ↵
bye
PDM:102>>!100 ↵
echo hello
hello
```

Notice that the parallel debug manager displays the command that you are re-entering.

- An alternate way to repeat any of the last 20 commands is to use:

  **!***string*

  This command tells the parallel debug manager to execute the last command that began with *string.* For example,

  ```
  PDM:103>>pstep –g GROUPA ↵
  PDM:104>>send –g GROUPA ?pc ↵
  [MVP2_MP]   0x40000000
  [MVP2_PP0]  0x40000004
  PDM:103>>pstep –g GROUPB ↵
  PDM:104>>send –g GROUPB ?pc ↵
  [MVP1_PP1]  0x4000001A
  [MVP1_PP2]  0x40000014
  PDM:105>>!p ↵
  pstep –g GROUPB
  ```

- To see a list of the last 20 commands that you entered, type:

  ```
  history ↵
  ```

The command history for the parallel debug manager works differently from that of the debugger (described on page DB:6-5); the TAB and F2 keys have no command-history meaning for the parallel debug manager.

## 4.5 Defining Your Own Command Strings

The ALIAS command provides a shorthand method of entering often-used commands or command sequences. The UNALIAS command deletes one or more ALIAS definitions. The syntax for the parallel debug manager version of each of these commands is:

**alias** [*alias name* [**,** ”*command string*”]]
**unalias** {*alias name* **|** *}

The parallel debug manager versions of the ALIAS and UNALIAS commands are similar to the debugger versions of these commands. You can:

- ❑ Include several commands in the command string by separating the individual commands with semicolons
- ❑ Define parameters in the command string by using a percent sign and a number (%1, %2, etc.) to represent a parameter whose value will be supplied when you execute the alias command
- ❑ List all currently defined parallel debug manager aliases by entering ALIAS with no parameters
- ❑ Find the definition of a parallel debug manager alias by entering ALIAS with only an alias-name parameter
- ❑ Nest alias definitions
- ❑ Redefine an alias
- ❑ Delete a single parallel debug manager alias by supplying the UNALIAS command with an alias name, or delete all parallel debug manager aliases by entering UNALIAS *

Like debugger aliases, parallel debug manager alias definitions are lost when you exit the parallel debug manager. However, individual commands within a PDM command string don’t have an expanded-length limit.

For more information about these features, see Section 6.5, *Defining Your Own Command Strings*.

The parallel debug manager version of this command is especially useful for aliasing often-used command strings involving the SEND and SET commands.

❑ You can use the ALIAS command to create parallel debug manager versions of debugger commands. For example, the ML debugger command lists the memory ranges that are currently defined. To make a parallel debug manager version of the ML command to list the memory ranges of all the debuggers in a particular group, enter:

```
alias ml, ”send –g %1 ml” ⏎
```

You could then list the memory maps of a group of processors such as those in group GROUPA:

```
ml GROUPA ⏎
```

❑ The ALIAS command can be helpful if you frequently change the default group. For example, suppose you plan to switch between two groups. You can set up the following alias:

```
alias switch, ”set dgroup $%1; set prompt %1” ⏎
```

The %1 parameter will be filled in with the group information that you enter when you execute SWITCH. Notice that the %1 parameter is preceded by a dollar sign ($) to set up the default group. The dollar sign tells the parallel debug manager to evaluate (take the list of processor names defined in the group instead of the actual group name). However, to change the prompt, you don’t want the parallel debug manager to evaluate (use the processors associated with the group name as the prompt)—you just want the group name. As a result, you don’t need to use the dollar sign when you want to use only the group name.

Assume that GROUPB contains MVP1_MP, MVP1_PP0, and MVP1_PP1. To make GROUPB the current default group and make the parallel debug manager prompt the same name as your default group, enter:

```
switch GROUPB ⏎
```

This causes the default group (dgroup) to contain the MVP1_MP, MVP1_PP0, and MVP1_PP1, and changes the parallel debug manager prompt to GROUPB:x>>.

## 4.6 Entering Operating-System Commands

The SYSTEM command provides you with a method of entering operating-system commands. The format for this command is:

**system** *operating-system command*

By using the SYSTEM command, you can enter operating-system commands without having to leave the primary environment (in this case, the parallel debug manager) and without having to open another operating-system window.

## 4.7 Understanding the Parallel Debug Manager's Expression Analysis

The parallel debug manager analyzes expressions differently than individual debuggers do (expression analysis for the debugger is described in Chapter 15, *Basic Information About C Expressions*). The parallel debug manager uses a simple integral expression analyzer. You can use expressions to cause the parallel debug manager to make decisions as part of the @ command and the flow control commands (described on pages DB:4-27 and DB:4-16, respectively).

Note that you cannot evaluate string variables with the parallel debug manager expression analyzer. You can evaluate only constant expressions.

Table 4–1 summarizes the parallel debug manager operators. The parallel debug manager interprets the operators in the order that they're listed in Table 4–1 (left to right, top to bottom).

Table 4–1. Parallel Debug Manager Operators

| Operator | Definition | Operator | Definition |
|---|---|---|---|
| ( ) | take highest precedence | * | multiplication |
| / | division | % | modulo |
| + | addition (binary) | – | subtraction (binary) |
| < < | left shift | ~ | complement |
| < | less than | > > | right shift |
| > | greater than | < = | less than or equal to |
| = = | is equal to | > = | greater than or equal to |
| & | bitwise AND | ! = | is not equal to |
| \| | bitwise OR | ^ | bitwise exclusive-OR |
| \| \| | logical OR | && | logical AND |

## 4.8 Using System Variables

You can use the SET, @, and UNSET commands to:

- Create your own system variables
- Assign a variable to the result of an expression
- Change the parallel debug manager prompt
- Check the execution status of the processors
- List system variables
- Delete system variables

### *Creating your own system variables*

The SET command lets you create system variables that you can use with the parallel debug manager (PDM) commands. The syntax for the SET command is:

**set** [*variable name* [**=** *string*] ]

The *variable name* can consist of up to 128 alphanumeric or underscore characters.

For example, suppose you have an array that you want to examine frequently. You can use the SET command to define a system variable that represents that array value:

```
set result = ar1[0] + 100 ↵
```

In this case, **result** is the variable name, and **ar1[0] + 100** is the expression that will be evaluated when you use the variable result.

Once you have defined result, you can use it with other PDM commands such as the SEND command:

```
send MVP1_PP1 ? $result ↵
```

The dollar sign (**$**) tells the parallel debug manager to replace result with ar1[0] + 100 (the string defined in result) as the expression parameter for the ? command. You **must** precede the name of a system variable with a $ when you want to use the string value you defined with the variable as a parameter.

You can also use the SET command to concatenate and substitute strings.

### ❑ Concatenating strings

The dollar sign followed by a system variable name enclosed in braces ( **{** and **}** ) tells the parallel debug manager to append the contents of the variable name to a string that precedes or follows the braces. For example:

```
set k = Hel ↵                                    Set k to the string Hel
set i = ${k}lo ${k}en ↵     Concatenate the contents of k before
                                     lo and en and set the result to i
echo $i ↵                                     Show the contents of i
Hello Helen
```

### ❑ Substituting strings

You can substitute defined system variables for parts of variable names or strings. This series of commands illustrates the substitution feature:

```
set err0 = 25 ↵                                      Set err0 to 25
set j = 0 ↵                                              Set j to 0
echo $err$j ↵                Show the value of $err$j → $err0 → 25
25
```

Note that substitution stops when the parallel debug manager detects recursion (for example, $k = k).

## *Assigning a variable to the result of an expression*

The @ (substitute) command is similar to the SET command. You can use the @ command to assign the result of an expression to a variable. The syntax for the @ command is:

@ *variable name* **=** *expression*

The following series of commands illustrates the difference between the @ command and the SET command. Assume that mask1 equals 36 and mask2 equals 47.

```
set mask3 = $mask1+$mask2 ⏎          Set mask3 to the contents of
                                 mask1 plus the contents of mask2
echo $mask3 ⏎                          Show the contents of mask3
36+47
@ mask3 = $mask1+$mask2 ⏎           Set mask3 to the result of the
                                     expression $mask1+$mask2
echo $mask3 ⏎                          Show the contents of mask3
83
```

Notice the difference between the two commands. The @ command evaluates the expression and assigns the result to the variable name.

The @ command is useful in setting loop counters. For example, you can initialize a counter with the following command:

```
@ j = 0 ⏎
```

Inside the loop, you can increment the counter with the following statement:

```
@ j = $j + 1 ⏎
```

## *Changing the parallel debug manager prompt*

The parallel debug manager recognizes a system variable called prompt. You can change the parallel debug manager prompt by setting the prompt variable to a string. For example, to change the parallel debug manager prompt to 3PROCs, enter:

```
set prompt = 3PROCs ↵
```

After entering this command, the parallel debug manager prompt will look like this: 3PROCs:x>>.

## *Checking the execution status of the processors*

In addition to displaying the execution status of a processor or group of processors, the STAT command (described on page DB:4-12) sets a system variable called status.

- If *all* of the processors in the specified group are running, the status variable is set to 1.
- If one or more of the processors in the group is halted, the status variable is set to 0.

You can use this variable when you want an instruction loop to execute until a processor halts:

```
loop stat == 1
send ?pc
.
.
```

## *Listing system variables*

To list all system variables, use the SET command without parameters:

```
set ↵
```

You can also list contents of a single variable. For example,

```
set j ↵
j  ″100″
```

## *Deleting system variables*

To delete a system variable, use the UNSET command. The format for this command is:

**unset** *variable name*

If you want to delete all of the variables you have created and any groups you have defined (as described on page DB:4-5), use the UNSET command with an asterisk instead of a variable name:

```
unset * ↵
```

**Note:**

When you use UNSET * to delete all of your system variables and processor groups, variables such as prompt, status, and dgroup are also deleted.

## 4.9 Evaluating Expressions

The debugger includes an EVAL command that evaluates an expression (see Section 9.2, *Basic Commands For Managing Data*, for more information about the debugger version of the EVAL command). The parallel debug manager has a similar command called EVAL that you can send to a processor or a group of processors. The EVAL command evaluates an expression in a debugger and sets a variable to the result of the expression. The syntax for the parallel debug manager version of the EVAL command is:

**eval** [**–g** {*group* **|** *processor name*}] *variable name***=***expression*[**,** *format*]

- ❑ The **–g** option specifies the group or processor that EVAL should be sent to. If you don't use this option, the command is sent to the default group (dgroup).
- ❑ When you send the EVAL command to more than one processor, the parallel debug manager takes the *variable name* that you supply and appends a suffix for each processor. The suffix consists of the underscore character (_) followed by the name that you assigned the processor. That way, you can differentiate between the resulting variables.
- ❑ The *expression* can be any expression that uses the symbols described in Section 4.7.
- ❑ When you use the optional *format* parameter, the value that the variable is set to will be in one of the following formats:

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

Suppose the program that is running on the MP has two variables defined: j is equal to 5, and k is equal to 17. Also assume that the program that is running on PP0 contains variables j and k: j is equal to 12, and k is equal to 22.

```
set dgroup = MVP1_MP MVP1_PP0 ↵
eval val = j + k ↵
set ↵
dgroup        ”MVP1_MP MVP1_PP0”
val_MVP1_MP   ”23”
val_MVP1_PP0  ”34”
```

Notice that the parallel debug manager created a system variable for each processor: val_MVP1_MP for the MVP1_MP and val_MVP1_PP0 for the MVP1_PP0.

Chapter 5

# The Debugger Display

The C source debugger uses a common window-oriented display. This chapter shows what windows can look like and describes the basic types of windows that you'll use.

**Topics**

## 5.1 Debugging Modes and Default Displays

The debugger has three debugging modes:

❑ Auto mode
❑ Assembly mode
❑ Mixed mode

Each mode changes the debugger display by adding or hiding specific windows. Some windows, such as the COMMAND window, are present in all modes. The following figures show the default displays for these modes and show the windows that the debugger automatically displays for these modes.

### *Auto mode*

In **auto mode**, the debugger automatically displays whatever type of code is currently running—assembly language or C. This is the default mode; when you first invoke the debugger, you'll see a a display similar to Figure 5–1. Auto mode has two types of displays:

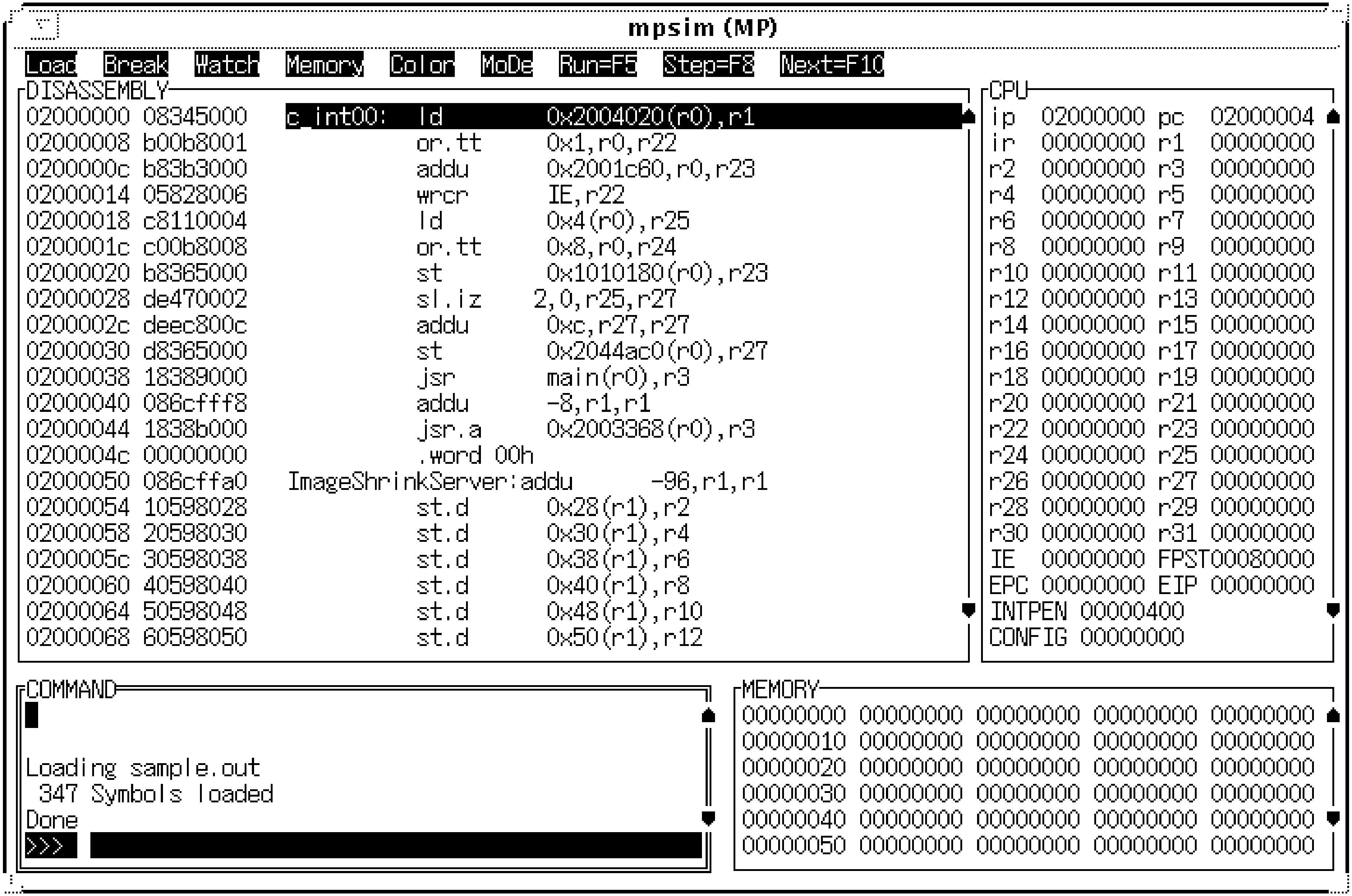
❑ When the debugger is running assembly language code, you'll see an assembly display similar to the one in Figure 5–1. The DISASSEMBLY window displays the reverse assembly of memory contents.

❑ When the debugger is running C code, you'll see a C display similar to the one in Figure 5–2. (This assumes that the debugger can find your C source file to display in the FILE window. If the debugger can't find your source, then it switches to mixed mode.)

When you're running assembly language code, the debugger automatically displays windows as described in the *Assembly mode* subsection on page DB:5-5.

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

Figure 5–1. Typical Assembly Display (for Auto Mode and Assembly Mode)



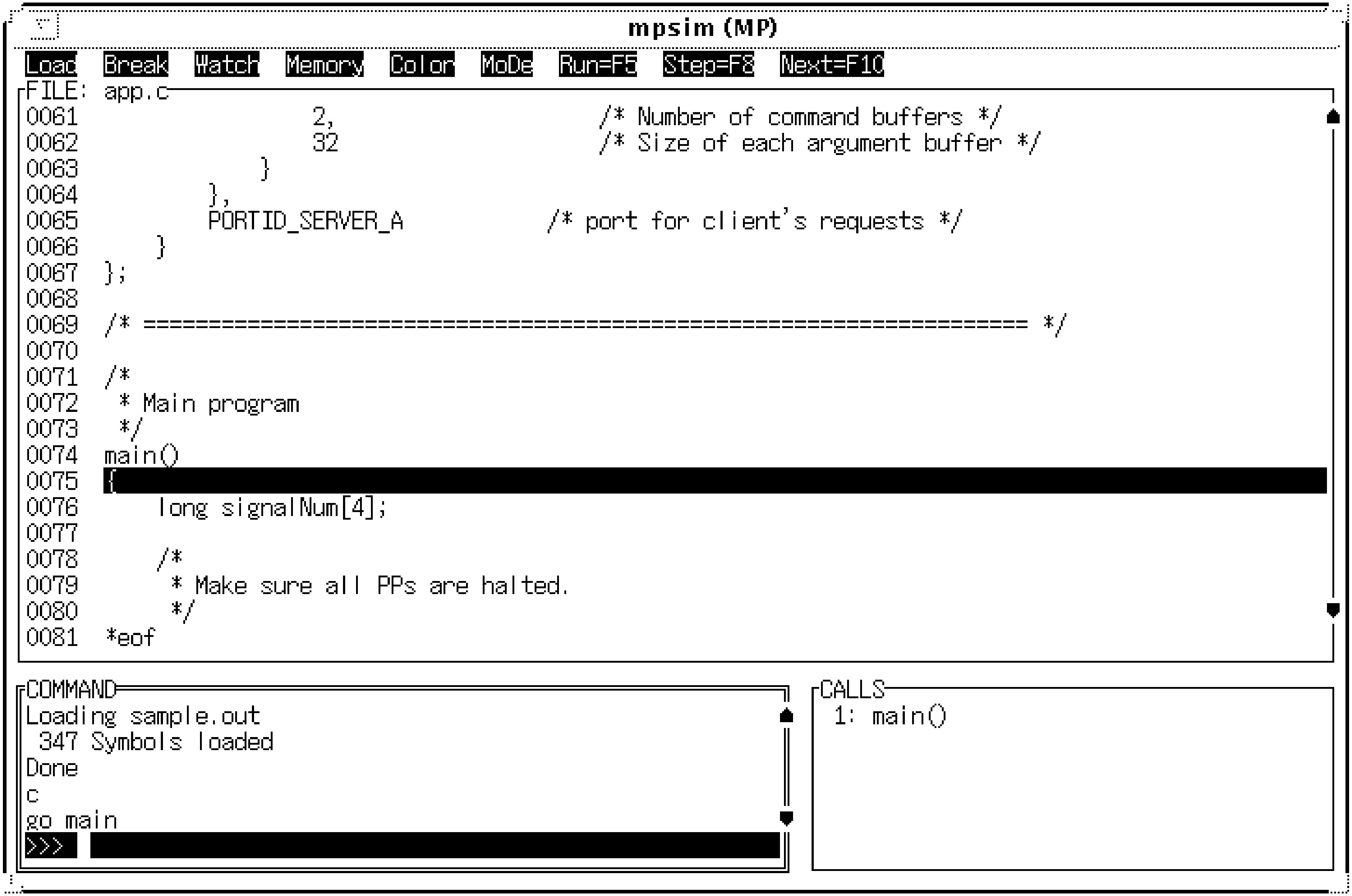
```
DISASSEMBLY
02000000 08345000   c_int00:  ld       0x2004020(r0),r1
02000008 b00b8001             or.tt    0x1,r0,r22
0200000c b83b3000             addu     0x2001c60,r0,r23
02000014 05828006             wrcr     IE,r22
02000018 c8110004             ld       0x4(r0),r25
0200001c c00b8008             or.tt    0x8,r0,r24
02000020 b8365000             st       0x1010180(r0),r23
02000028 de470002             sl.iz   2,0,r25,r27
0200002c deec800c             addu     0xc,r27,r27
02000030 d8365000             st       0x2044ac0(r0),r27
02000038 18389000             jsr      main(r0),r3
02000040 086cfff8             addu     -8,r1,r1
02000044 1838b000             jsr.a    0x2003368(r0),r3
0200004c 00000000             .word 00h
02000050 086cffa0   ImageShrinkServer:addu      -96,r1,r1
02000054 10598028             st.d     0x28(r1),r2
02000058 20598030             st.d     0x30(r1),r4
0200005c 30598038             st.d     0x38(r1),r6
02000060 40598040             st.d     0x40(r1),r8
02000064 50598048             st.d     0x48(r1),r10
02000068 60598050             st.d     0x50(r1),r12
```

```
CPU
ip  02000000 pc  02000004
ir  00000000 r1  00000000
r2  00000000 r3  00000000
r4  00000000 r5  00000000
r6  00000000 r7  00000000
r8  00000000 r9  00000000
r10 00000000 r11 00000000
r12 00000000 r13 00000000
r14 00000000 r15 00000000
r16 00000000 r17 00000000
r18 00000000 r19 00000000
r20 00000000 r21 00000000
r22 00000000 r23 00000000
r24 00000000 r25 00000000
r26 00000000 r27 00000000
r28 00000000 r29 00000000
r30 00000000 r31 00000000
IE  00000000 FPST00080000
EPC 00000000 EIP 00000000
INTPEN 00000400
CONFIG 00000000
```

```
COMMAND


Loading sample.out
 347 Symbols loaded
Done
>>>
```

```
MEMORY
00000000 00000000 00000000 00000000 00000000
00000010 00000000 00000000 00000000 00000000
00000020 00000000 00000000 00000000 00000000
00000030 00000000 00000000 00000000 00000000
00000040 00000000 00000000 00000000 00000000
00000050 00000000 00000000 00000000 00000000
```

Figure 5–2. Typical C Display (for Auto Mode Only)



```
                2,                    /* Number of command buffers */
                32                    /* Size of each argument buffer */
            }
        },
        PORTID_SERVER_A           /* port for client's requests */
    }
};

/* ==================================================================== */

/*
 * Main program
 */
main()
{
    long signalNum[4];

    /*
     * Make sure all PPs are halted.
     */
*eof
```

```
Loading sample.out
 347 Symbols loaded
Done
c
go main
>>>
```

```
 1: main()
```

## *Assembly mode*

**Assembly mode** is for viewing assembly language programs only. In this mode, you'll see a display similar to the one shown in Figure 5–1. When you're in assembly mode, you'll always see the assembly display, regardless of whether C or assembly language is currently running.

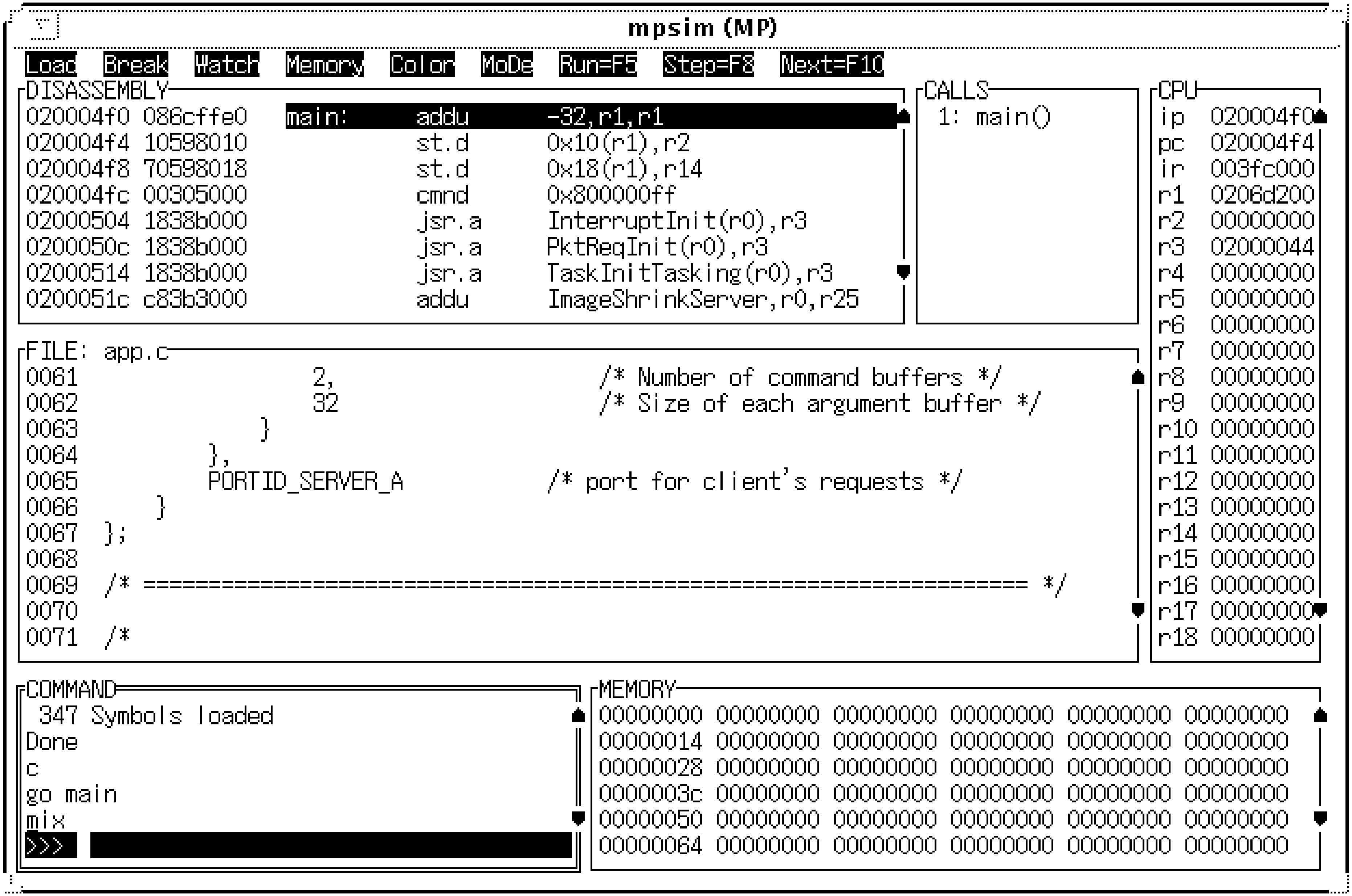
Windows that are automatically displayed in assembly mode include the MEMORY window, the DISASSEMBLY window, the CPU window, and the COMMAND window. You can also open a WATCH window in assembly mode.

## *Mixed mode*

**Mixed mode** is for viewing assembly language and C code at the same time. Figure 5–3 shows the default display for mixed mode.

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

Figure 5–3. Typical Mixed Display (for Mixed Mode Only)



DISASSEMBLY

```
020004f0 086cffe0    main:      addu       -32,r1,r1
020004f4 10598010               st.d       0x10(r1),r2
020004f8 70598018               st.d       0x18(r1),r14
020004fc 00305000               cmnd       0x800000ff
02000504 1838b000               jsr.a      InterruptInit(r0),r3
0200050c 1838b000               jsr.a      PktReqInit(r0),r3
02000514 1838b000               jsr.a      TaskInitTasking(r0),r3
0200051c c83b3000               addu       ImageShrinkServer,r0,r25
```

CALLS

```
 1: main()
```

CPU

```
ip   020004f0
pc   020004f4
ir   003fc000
r1   0206d200
r2   00000000
r3   02000044
r4   00000000
r5   00000000
r6   00000000
r7   00000000
r8   00000000
r9   00000000
r10  00000000
r11  00000000
r12  00000000
r13  00000000
r14  00000000
r15  00000000
r16  00000000
r17  00000000
r18  00000000
```

FILE: app.c

```
0061                  2,                        /* Number of command buffers */
0062                  32                        /* Size of each argument buffer */
0063              }
0064          },
0065          PORTID_SERVER_A           /* port for client's requests */
0066      }
0067  };
0068
0069  /* ================================================================== */
0070
0071  /*
```

COMMAND

```
 347 Symbols loaded
Done
c
go main
mix
>>>
```

MEMORY

```
00000000 00000000 00000000 00000000 00000000 00000000
00000014 00000000 00000000 00000000 00000000 00000000
00000028 00000000 00000000 00000000 00000000 00000000
0000003c 00000000 00000000 00000000 00000000 00000000
00000050 00000000 00000000 00000000 00000000 00000000
00000064 00000000 00000000 00000000 00000000 00000000
```

### *Restrictions associated with debugging modes*

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

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

dasm
calls
func
file
mem
disp

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

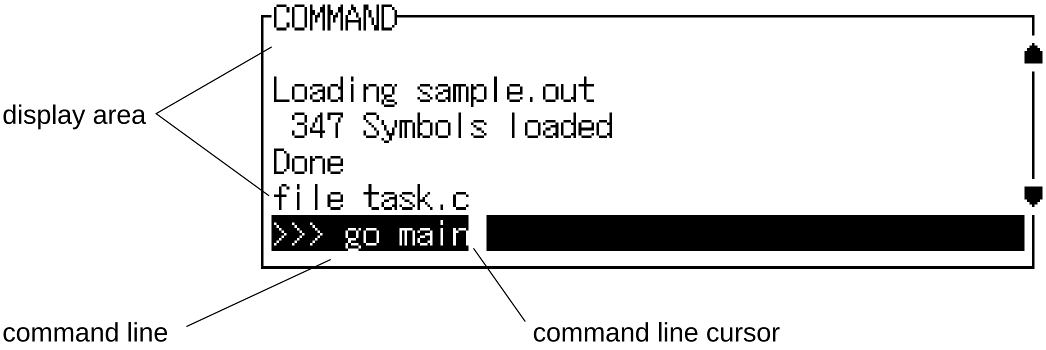
The debugger can show several types of windows. Every window is identified by a name in its upper left corner. Each type of window serves a specific purpose and has unique characteristics. There are eight different windows, divided into three general categories.

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

You can move or resize any of these windows; you can also edit any value in a data-display window. Before you can perform any of these actions, however, you must select the window you want to move, resize, or edit and make it the **active window**. For more information about making a window active, see Section 5.4, *The Active Window*.

The remainder of this section describes the individual windows.

## COMMAND window



```
Loading sample.out
 347 Symbols loaded
Done
file task.c
>>> go main
```

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

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

*Modes* All modes

*Created* Automatically

*Affected by*
- All commands entered on the command line
- All commands that display output in the display area
- Any 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. However, there are a few instances, such as when you're editing a field, in which you can't type a command (see page DB:6-4 for more information).

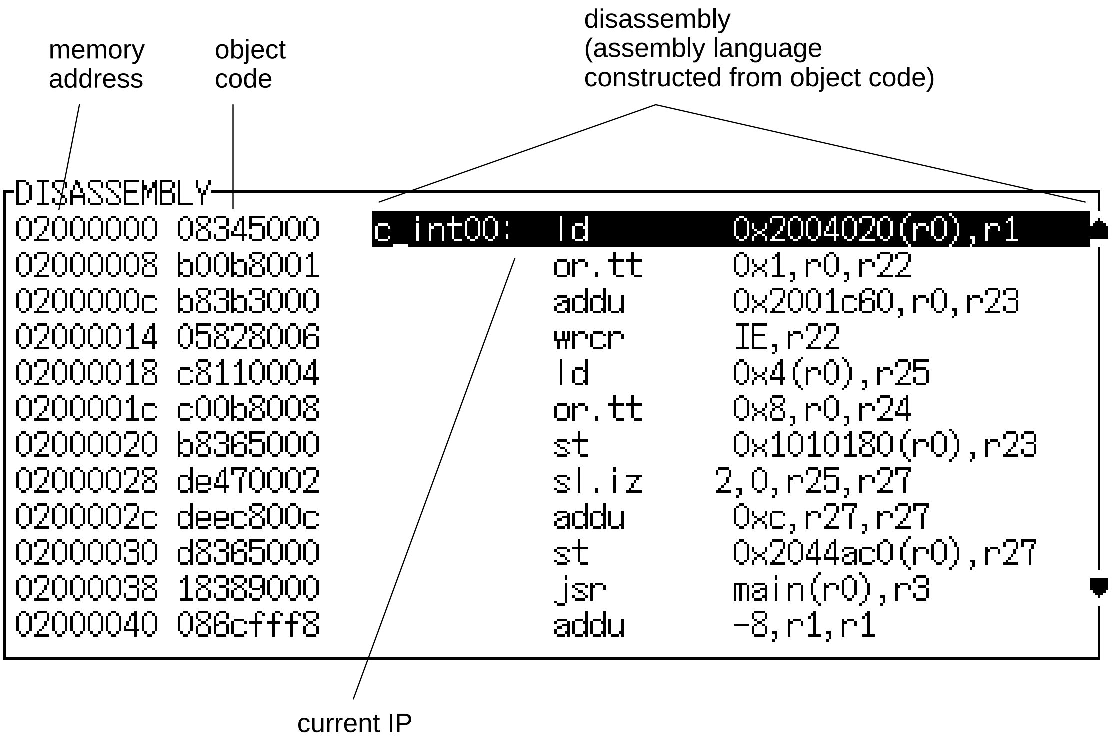
  The debugger keeps a list of the last 50 commands that you entered. You can select and re-enter commands from the list without retyping them. (For more information on using the command history, see *Using the command history,* page DB:6-5.)

- **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 6, *Entering and Using Commands*.

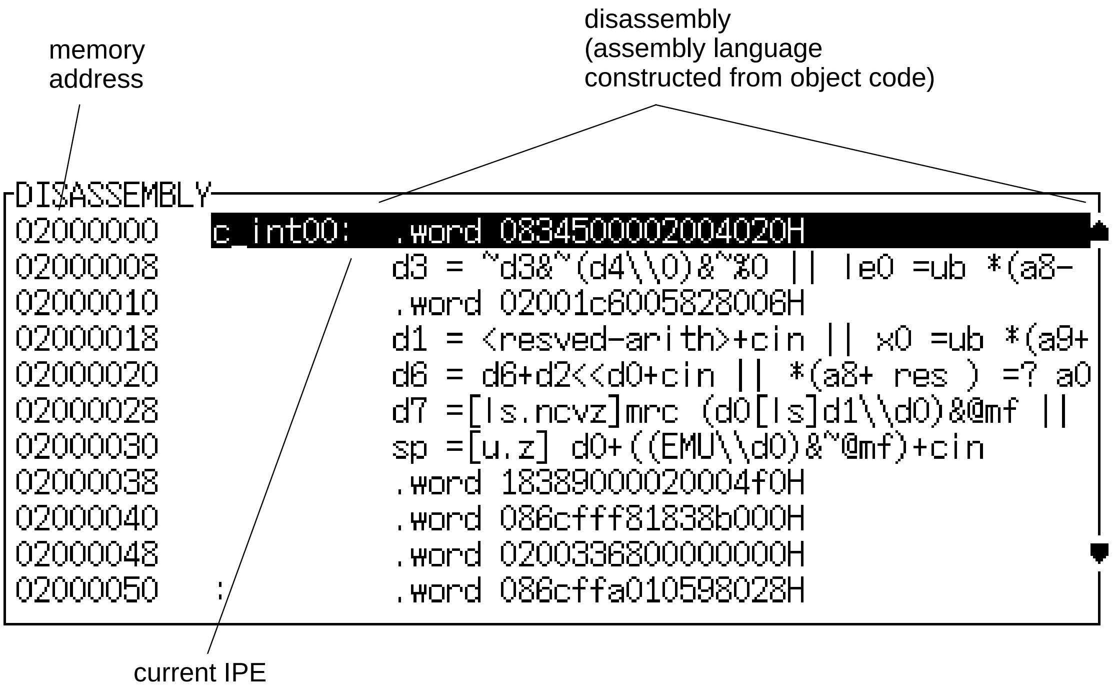
## *DISASSEMBLY window*

Figure 5–4. DISASSEMBLY Window—MP Version



```
DISASSEMBLY
02000000 08345000  c_int00:  ld         0x2004020(r0),r1
02000008 b00b8001            or.tt      0x1,r0,r22
0200000c b83b3000            addu       0x2001c60,r0,r23
02000014 05828006            wrcr       IE,r22
02000018 c8110004            ld         0x4(r0),r25
0200001c c00b8008            or.tt      0x8,r0,r24
02000020 b8365000            st         0x1010180(r0),r23
02000028 de470002            sl.iz     2,0,r25,r27
0200002c deec800c            addu       0xc,r27,r27
02000030 d8365000            st         0x2044ac0(r0),r27
02000038 18389000            jsr        main(r0),r3
02000040 086cfff8            addu       -8,r1,r1
```

Figure 5–5. DISASSEMBLY Window—PP Version

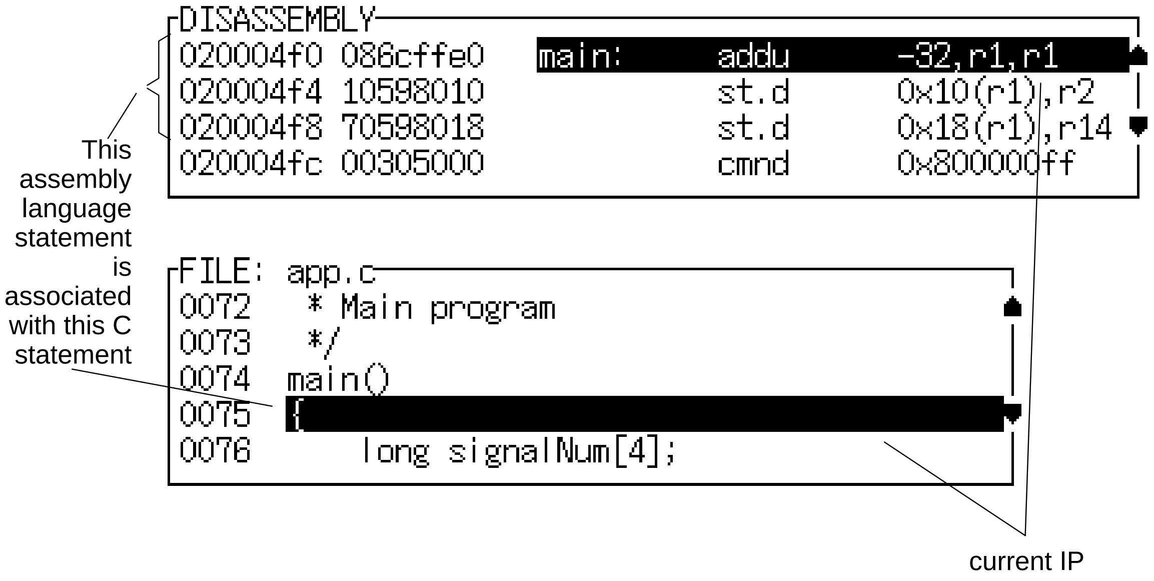


```
DISASSEMBLY
02000000   c_int00:   .word 0834500002004020H
02000008              d3 = ~d3&~(d4\\0)&~%0 || le0 =ub *(a8-
02000010              .word 02001c6005828006H
02000018              d1 = <resved-arith>+cin || x0 =ub *(a9+
02000020              d6 = d6+d2<<d0+cin || *(a8+ res ) =? a0
02000028              d7 =[ls.ncvz]mrc (d0[ls]d1\\d0)&@mf ||
02000030              sp =[u.z] d0+((EMU\\d0)&~@mf)+cin
02000038              .word 1838900020004f0H
02000040              .word 086cfff81838b000H
02000048              .word 0200336800000000H
02000050   :          .word 086cffa010598028H
```

| | |
|---|---|
| *Purpose* | Displays the disassembly (or reverse assembly) of memory contents |
| *Editable?* | No; pressing the edit key (F9) or the left mouse button sets a software breakpoint on an assembly language statement. |
| *Modes* | Auto (assembly display only), assembly, and mixed |
| *Created* | Automatically |
| *Affected by* | ❑ DASM and ADDR commands<br>❑ Breakpoint and run commands |

Within the DISASSEMBLY window, the debugger highlights:

- The statement that the MP's IP (instruction pointer) or the PP's IPE (instruction pointer execution stage) is pointing to after the processor is halted (if that line is in the current display)
- Any statements with software breakpoints
- The address and object code (MP only) fields for all statements associated with the current C statement, as shown:



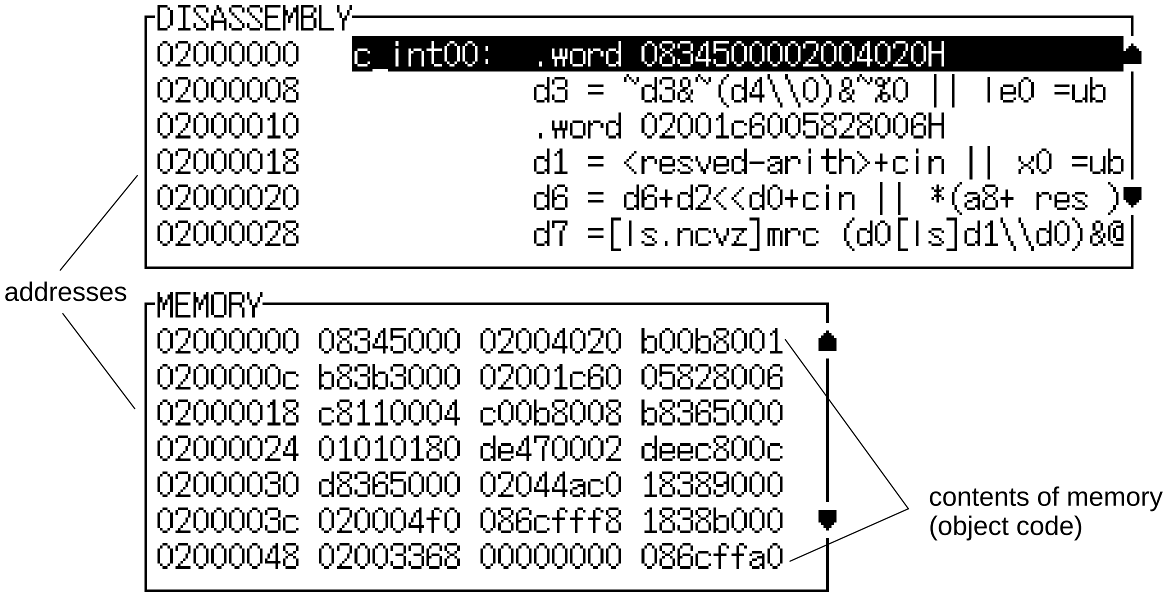
```
DISASSEMBLY
020004f0 086cffe0   main:     addu       -32,r1,r1
020004f4 10598010             st.d       0x10(r1),r2
020004f8 70598018             st.d       0x18(r1),r14
020004fc 00305000             cmnd       0x800000ff
```

```
FILE: app.c
0072   * Main program
0073   */
0074  main()
0075  {
0076      long signalNum[4];
```

This assembly language statement is associated with this C statement

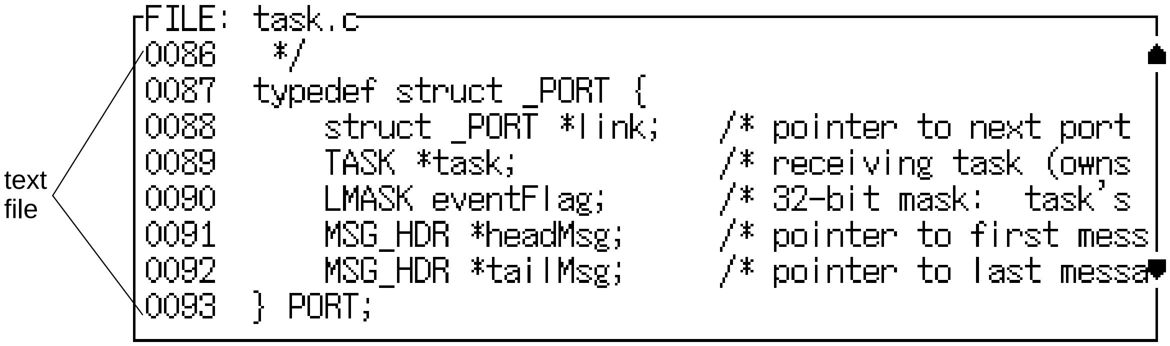
current IP

The PP version of the debugger does not display the object code in the DISASSEMBLY window. However, you can view the object code by displaying the program memory in the MEMORY window. To do this, use the MEM command (see the *MEMORY windows* discussion on page DB:5-16).



```
DISASSEMBLY
02000000   c_int00:   .word 0834500002004020H
02000008              d3 = ~d3&~(d4\\0)&~%0 || le0 =ub
02000010              .word 02001c6005828006H
02000018              d1 = <resved-arith>+cin || x0 =ub
02000020              d6 = d6+d2<<d0+cin || *(a8+ res )
02000028              d7 =[ls.ncvz]mrc (d0[ls]d1\\d0)&@

MEMORY
02000000 08345000 02004020 b00b8001
0200000c b83b3000 02001c60 05828006
02000018 c8110004 c00b8008 b8365000
02000024 01010180 de470002 deec800c
02000030 d8365000 02044ac0 18389000
0200003c 020004f0 086cfff8 1838b000
02000048 02003368 00000000 086cffa0
```

## *FILE window*



```
FILE: task.c
0086   */
0087  typedef struct _PORT {
0088      struct _PORT *link;   /* pointer to next port
0089      TASK *task;           /* receiving task (owns
0090      LMASK eventFlag;      /* 32-bit mask:  task's
0091      MSG_HDR *headMsg;     /* pointer to first mess
0092      MSG_HDR *tailMsg;     /* pointer to last messa
0093  } PORT;
```

*Purpose* Shows any text file you want to display

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

*Modes* Auto (C display only) and mixed

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

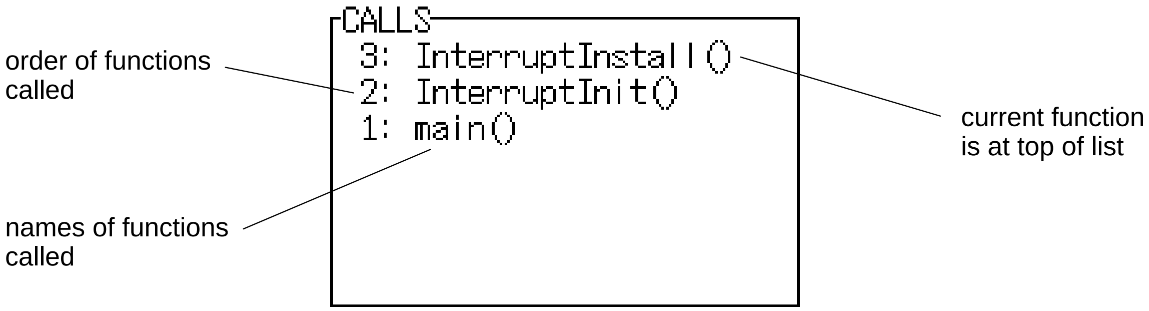
*Affected by*
- FILE, FUNC, and ADDR commands
- Breakpoint and run commands

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

Within the FILE window, the debugger highlights:

- The statement that the IP (for the MP) or IPE (for the PP) is pointing to (if that line is in the current display)
- Any statements where you've set a software breakpoint

## CALLS window



```
CALLS
 3: InterruptInstall()
 2: InterruptInit()
 1: main()
```

order of functions called

names of functions called

current function is at top of list

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

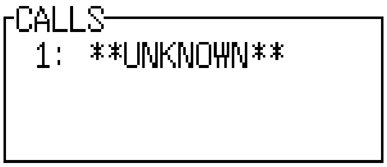
*Editable?* No; pressing the edit key (F9) or the left mouse button changes the FILE display to show the source associated with the called function.

*Modes* Auto (C display only) and mixed

*Created*
- ❏ Automatically when you’re displaying C code
- ❏ With the CALLS command if you closed the window

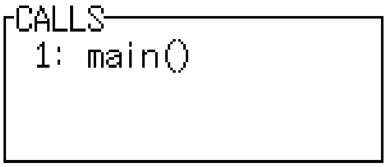
*Affected by* Run and single-step commands

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



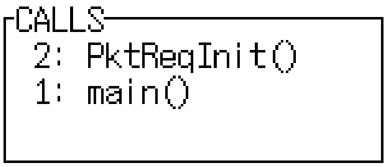
```
CALLS
 1: **UNKNOWN**
```

If you haven’t run any code, then no functions have been called yet. You’ll also see this if you’re running code but are not currently running a C function.

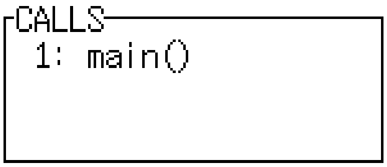


```
CALLS
 1: main()
```

In C programs, the first C function is main.



```
CALLS
 2: PktReqInit()
 1: main()
```



```
CALLS
 1: main()
```

As your program runs, the contents of the CALLS window change to reflect the current routine that you’re in and where the routine was called from. When you exit a routine, its name is popped from the CALLS list.

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

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

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

1) Make the CALLS window the active window (see Section 5.4, *The Active Window*).

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

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

You can close and reopen the CALLS window.

❑ Closing the window is a two-step process:

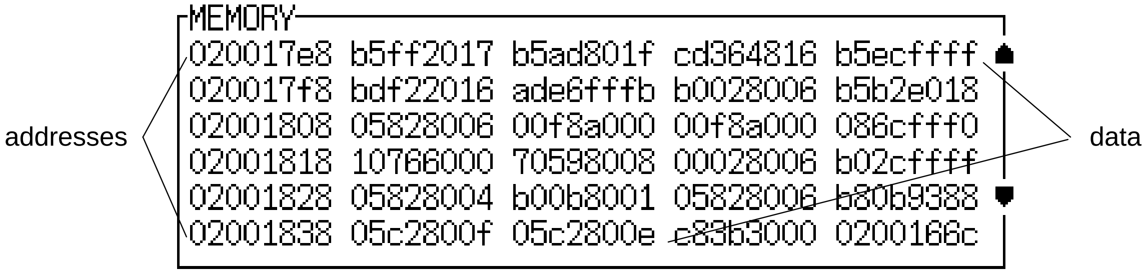
   1) Make the CALLS window the active window.

   2) Press F4.

❑ To reopen the CALLS window after you've closed it, enter the CALLS command. The format for this command is:

   **calls**

## *MEMORY windows*



```
MEMORY
020017e8 b5ff2017 b5ad801f cd364816 b5ecffff
020017f8 bdf22016 ade6fffb b0028006 b5b2e018
02001808 05828006 00f8a000 00f8a000 086cfff0
02001818 10766000 70598008 00028006 b02cffff
02001828 05828004 b00b8001 05828006 b80b9388
02001838 05c2800f 05c2800e c83b3000 0200166c
```

| | |
|---|---|
| *Purpose* | Displays the contents of memory |
| *Editable?* | Yes; you can edit the data (but not the addresses). |
| *Modes* | Auto (assembly display only), assembly, and mixed |
| *Created* | □ Automatically (the default MEMORY window only)<br>□ With the MEM# commands (up to three additional MEMORY windows ) |
| *Affected by* | MEM commands: MEM, MEM1, MEM2, and MEM3 |

A MEMORY window has two parts:

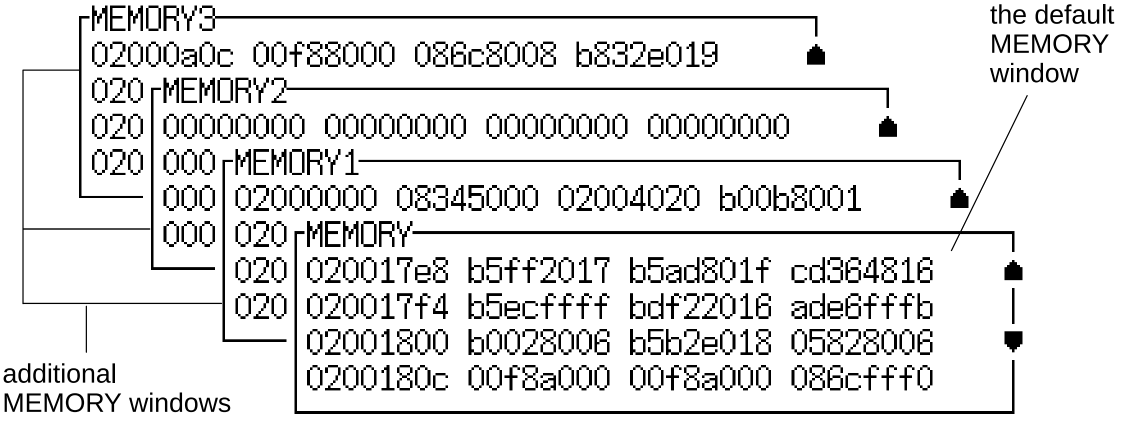
- □ **Addresses.** The first column of numbers identifies the addresses of the first column of displayed data. No matter how many columns of data you display, only one address column is displayed. Each address in this column identifies the address of the data immediately to its right.
- □ **Data.** The remaining columns display the values at the listed addresses. You can display more data by making the window wider and/or longer.

  The MEMORY window above has four columns of data, and each new address is incremented by four. Although the window shows four columns of data, there is still only one column of addresses; the first data value is at address 0x20017e8, the second at address 0x20017ec, etc.; the fifth value (first value in the second row) is at address 0x20017f8, the sixth at address 0x20017fc, etc.

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

Three additional MEMORY windows called MEMORY1, MEMORY2, and MEMORY3 are available. The default MEMORY window does not have an extension number in its name; this is because MEMORY1, MEMORY2, and MEMORY3 are optional windows and can be opened and closed throughout your debugging session. Having four windows allows you to view four different memory ranges. Refer to Figure 5–6.

Figure 5–6. The Default and Additional MEMORY Windows



```
MEMORY3
02000a0c 00f88000 086c8008 b832e019
020 MEMORY2
020 00000000 00000000 00000000 00000000
020 000 MEMORY1
    000 02000000 08345000 02004020 b00b8001
    000 020 MEMORY
        020 020017e8 b5ff2017 b5ad801f cd364816
        020 020017f4 b5ecffff bdf22016 ade6fffb
            02001800 b0028006 b5b2e018 05828006
            0200180c 00f8a000 00f8a000 086cfff0
```

the default MEMORY window

additional MEMORY windows

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

❑ **Creating a new MEMORY window.**

If the default MEMORY window is the only MEMORY window open and you want to open another MEMORY window, enter the MEM command with the appropriate extension number:

**mem[#]** *address*

For example, if you want to create a new memory window starting at address 0x2085000, you would enter:

```
mem1 0x2085000 ⏎
```

This displays a new window, MEMORY1, showing the contents of memory starting at the address 0x2085000.

❑ **Displaying a new memory range in the current MEMORY window.**

Displaying another block of memory identifies a new starting address for the memory range shown in the current MEMORY window. 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.

To view a different memory location in the default MEMORY window, use the MEM command:

**mem** *address*

To view different memory locations in the optional MEMORY windows, use the MEM command with the appropriate extension number on the end. For example:

| **To do this. . .** | **Enter this. . .** |
|---|---|
| View the block of memory starting at address 0x2004000 in the MEMORY1 window | **`mem1 0x2004000`** |
| View another block of memory starting at address 0x2085000 in the MEMORY2 window | **`mem2 0x2085000`** |

You can close and reopen additional MEMORY windows as often as you like.

- ❑ **Closing an additional MEMORY window.**

  Closing a window is a two-step process:

  1) Make the appropriate MEMORY window the active window (see Section 5.4).
  2) Press F4 .

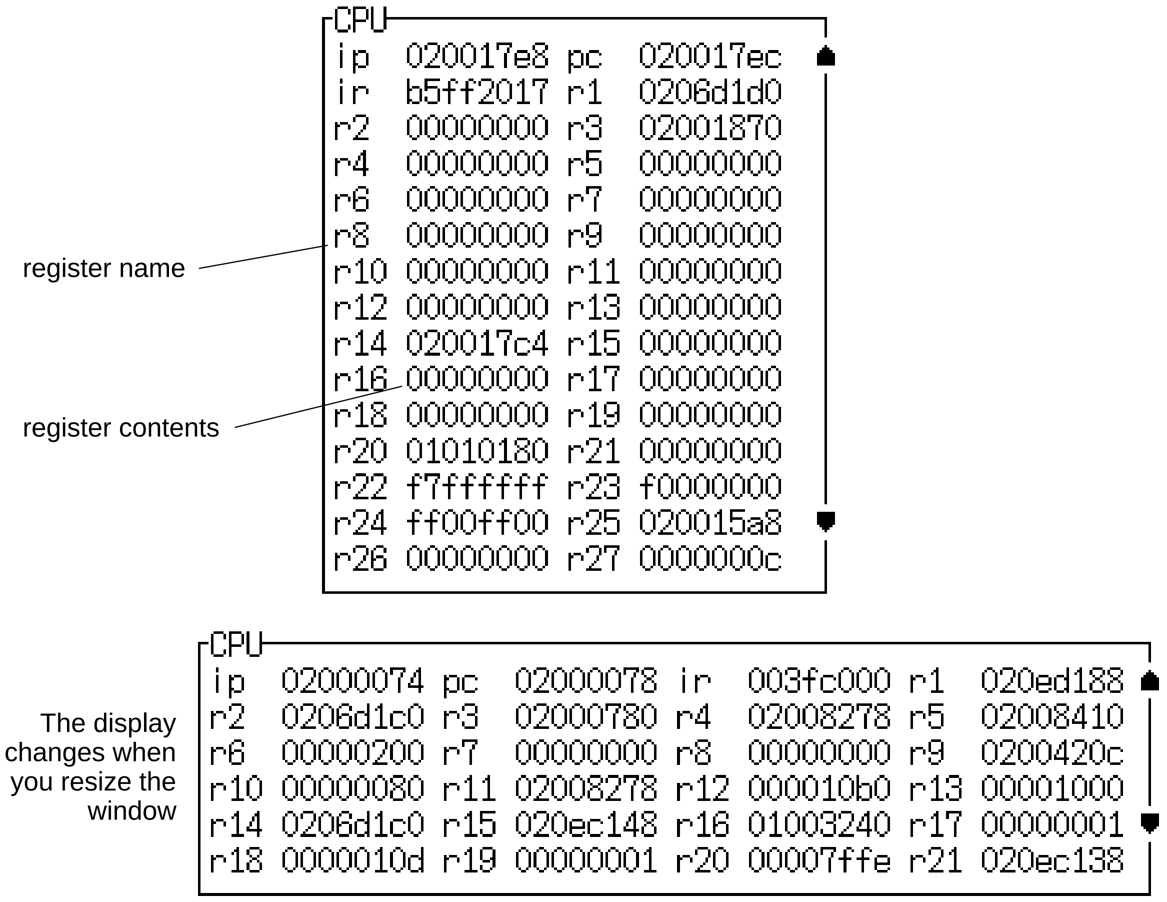
  Remember, you cannot close the default MEMORY window.

- ❑ **Reopening an additional MEMORY window.**

  To reopen an additional MEMORY window after you've closed it, enter the MEM command with its appropriate extension number.

## CPU window

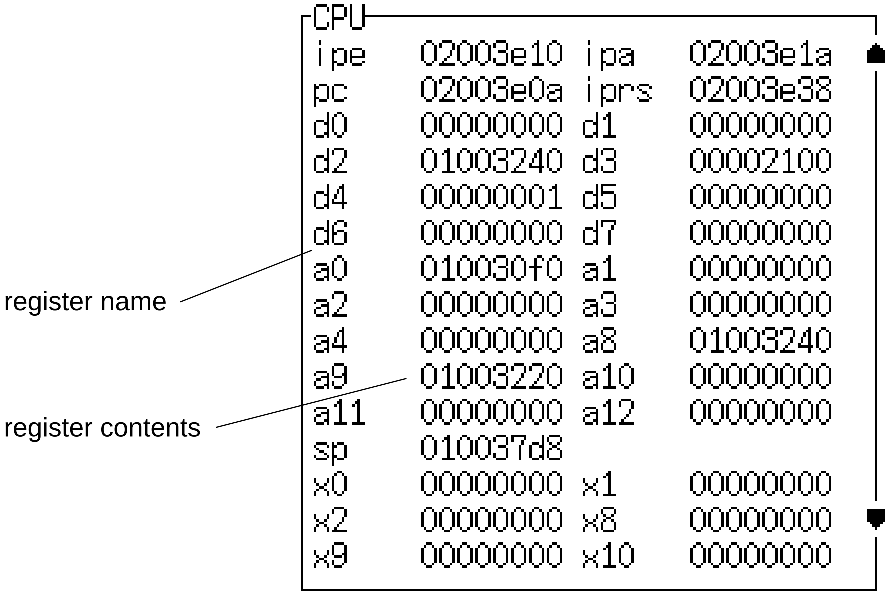
Figure 5–7. CPU Window—MP Version



```
CPU
ip   020017e8 pc   020017ec
ir   b5ff2017 r1   0206d1d0
r2   00000000 r3   02001870
r4   00000000 r5   00000000
r6   00000000 r7   00000000
r8   00000000 r9   00000000
r10  00000000 r11  00000000
r12  00000000 r13  00000000
r14  020017c4 r15  00000000
r16  00000000 r17  00000000
r18  00000000 r19  00000000
r20  01010180 r21  00000000
r22  f7ffffff r23  f0000000
r24  ff00ff00 r25  020015a8
r26  00000000 r27  0000000c
```

```
CPU
ip   02000074 pc   02000078 ir   003fc000 r1   020ed188
r2   0206d1c0 r3   02000780 r4   02008278 r5   02008410
r6   00000200 r7   00000000 r8   00000000 r9   0200420c
r10  00000080 r11  02008278 r12  000010b0 r13  00001000
r14  0206d1c0 r15  020ec148 r16  01003240 r17  00000001
r18  0000010d r19  00000001 r20  00007ffe r21  020ec138
```

Figure 5–8. CPU Window—PP Version

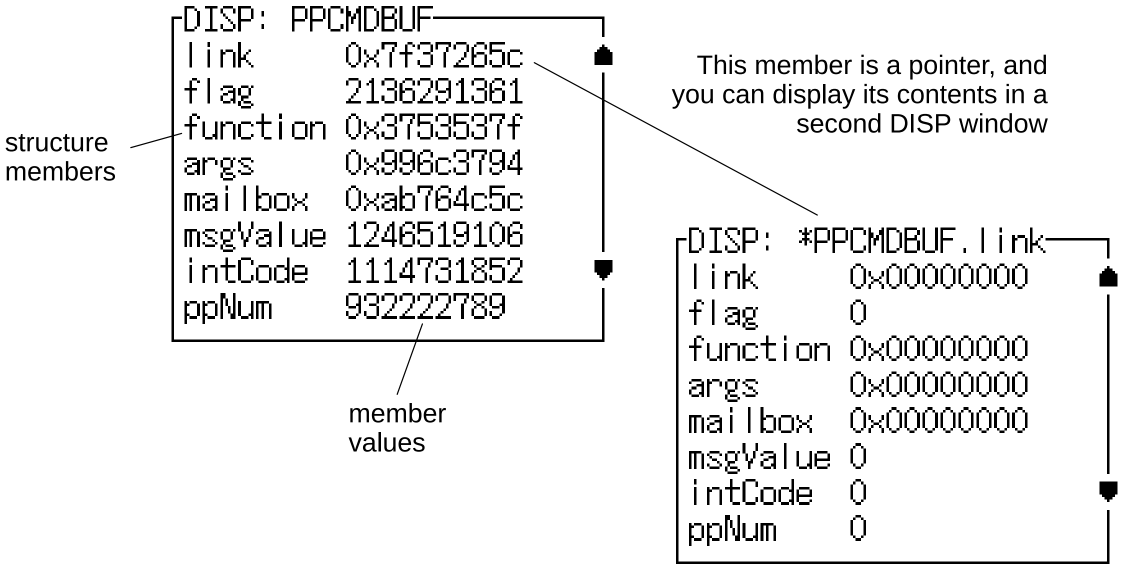


```
CPU
ipe    02003e10 ipa    02003e1a
pc     02003e0a iprs   02003e38
d0     00000000 d1     00000000
d2     01003240 d3     00002100
d4     00000001 d5     00000000
d6     00000000 d7     00000000
a0     010030f0 a1     00000000
a2     00000000 a3     00000000
a4     00000000 a8     01003240
a9     01003220 a10    00000000
a11    00000000 a12    00000000
sp     010037d8
x0     00000000 x1     00000000
x2     00000000 x8     00000000
x9     00000000 x10    00000000
```

| | |
|---|---|
| *Purpose* | Shows the contents of the MP or PP registers |
| *Editable?* | Yes; you can edit the value of any displayed register. |
| *Modes* | Auto (assembly display only), assembly, and mixed |
| *Created* | Automatically |
| *Affected by* | Data-management commands |

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

## DISP windows



```
DISP: PPCMDBUF
link      0x7f37265c
flag      2136291361
function  0x3753537f
args      0x996c3794
mailbox   0xab764c5c
msgValue  1246519106
intCode   1114731852
ppNum     932222789
```

```
DISP: *PPCMDBUF.link
link      0x00000000
flag      0
function  0x00000000
args      0x00000000
mailbox   0x00000000
msgValue  0
intCode   0
ppNum     0
```

structure members

member values

This member is a pointer, and you can display its contents in a second DISP window

*Purpose* Displays the members of a selected structure, array, or pointer, and the value of each member

*Editable?* Yes; you can edit individual values as long as they are completely displayed.

*Modes* Auto (C display only) and mixed

*Created* With the DISP command

*Affected by* DISP command

A DISP window is similar to a WATCH window (described on page DB:5-22), but it shows the values of an entire array or structure instead of a single value. Use the DISP command to open a DISP window; the basic syntax is:

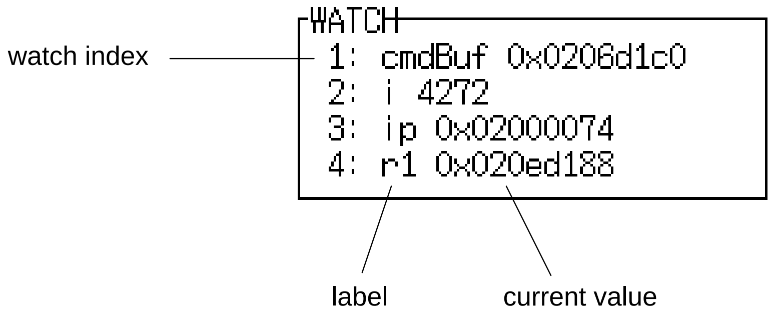
**disp** *expression*

Data is displayed in its natural format:

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

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

## WATCH window



```
WATCH
 1: cmdBuf 0x0206d1c0
 2: i 4272
 3: ip 0x02000074
 4: r1 0x020ed188
```

*Purpose* Displays the values of selected expressions

*Editable?* Yes; you can edit the value of any expression whose value corresponds to a single storage location (in registers or memory). In the window above, for example, you could edit the value of IP but couldn't edit the value of X+X.

*Modes* Auto, assembly, and mixed

*Created* With the WA command

*Affected by* WA, WD, and WR commands

The WATCH window helps you to track the values of arbitrary expressions, variables, and registers. Use the WA command for this; the syntax is:

**wa** *expression* [, *label*]

WA adds *expression* to the WATCH window. (If there's no WATCH window, then WA also opens a WATCH window).

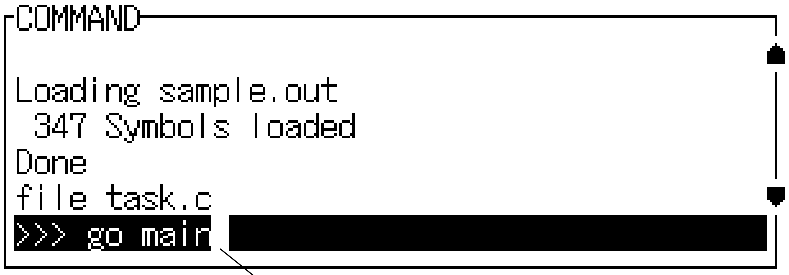
To delete individual entries from the WATCH window, use the WD command. To delete all entries at once and close the WATCH window, use the WR command.

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

## 5.3 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. Arrow keys **do not affect** the position of this cursor.

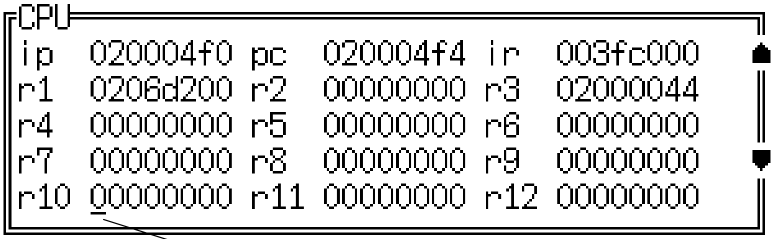


```
Loading sample.out
 347 Symbols loaded
Done
file task.c
>>> go main
```

command-line cursor

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

- The **current-field cursor** is an underscore cursor that identifies the current field in the active window. Arrow keys **do** affect this cursor's movement.



```
ip  020004f0 pc  020004f4 ir  003fc000
r1  0206d200 r2  00000000 r3  02000044
r4  00000000 r5  00000000 r6  00000000
r7  00000000 r8  00000000 r9  00000000
r10 00000000 r11 00000000 r12 00000000
```

current field cursor

# 5.4 The Active Window

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

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

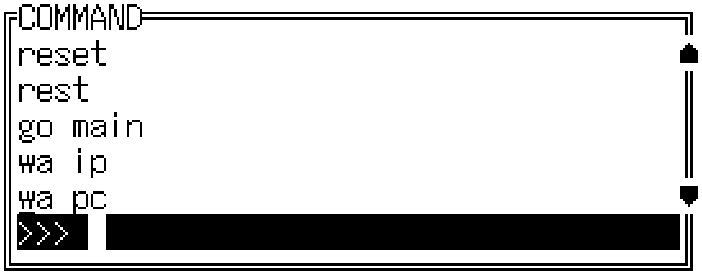
## *Identifying the active window*

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

You can alter the active window’s border style and colors if you wish; Figure 5–9 illustrates the default appearance of an active window and an inactive window.

Figure 5–9. Default Appearance of an Active and an Inactive Window

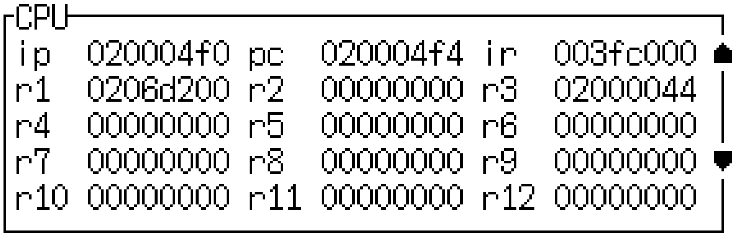
(a) An active window (default appearance)



```
COMMAND
reset
rest
go main
wa ip
wa pc
>>>
```

This window is highlighted to show that it is active

(b) An inactive window (default appearance)



```
CPU
ip  020004f0 pc  020004f4 ir  003fc000
r1  0206d200 r2  00000000 r3  02000044
r4  00000000 r5  00000000 r6  00000000
r7  00000000 r8  00000000 r9  00000000
r10 00000000 r11 00000000 r12 00000000
```

This window is not highlighted and is not active

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

## *Selecting the active window*

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

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

2) Click the left mouse button.

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

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

Press ESC to get out of this.

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

Press the left mouse button again to clear the breakpoint.

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

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

**win** *WINDOW NAME*

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

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

```
      win  DISASSEMBLY ⏎
or    win  DISA⏎
```

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

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

# 5.5 Manipulating Windows

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

**Note:**

You can resize, zoom, or move any window, but first the window must be **active**. For information about selecting the active window, see Section 5.4.

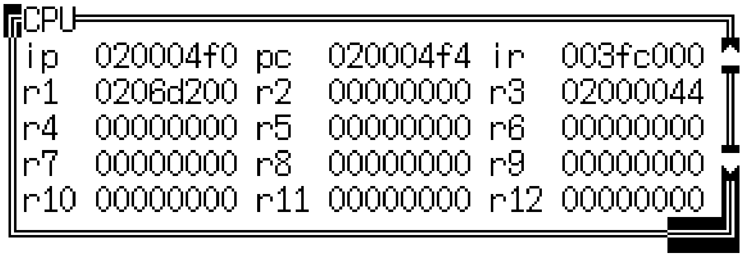
## *Resizing a window*

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

There are two basic ways to resize a window:

❑ By using the mouse
❑ By using the SIZE command

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



```
CPU
ip  020004f0 pc  020004f4 ir  003fc000
r1  0206d200 r2  00000000 r3  02000044
r4  00000000 r5  00000000 r6  00000000
r7  00000000 r8  00000000 r9  00000000
r10 00000000 r11 00000000 r12 00000000
```

lower right corner
(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 can't size it to the maximum height and width—you can size it only to the right and bottom screen borders. The easiest way to make a window as large as possible is to zoom it, as described on page DB:5-29.

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

```
win  CALLS ⏎
size 8, 20 ⏎
```

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

↓ Makes the active window one line longer

↑ Makes the active window one line shorter

← Makes the active window one character narrower

→ Makes the active window one character wider

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

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

```
win  CPU ⏎
size ⏎
↓ ↓ ↓     ← ←      ESC
```

## *Zooming a window*

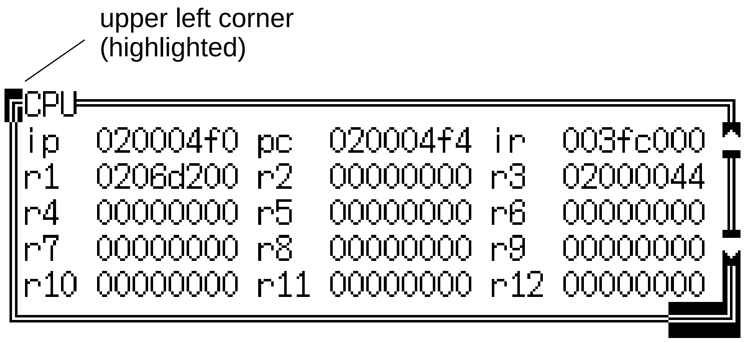
Another way to resize the active window is to zoom it. Zooming a window makes it as large as possible, so that it takes up the entire display (except for the menu bar) and hides all the other windows. Unlike the SIZE command, zooming is not affected by the window's position in the display.

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

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

❑ By using the mouse
❑ By using the ZOOM command

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



```
CPU
ip  020004f0 pc  020004f4 ir  003fc000
r1  0206d200 r2  00000000 r3  02000044
r4  00000000 r5  00000000 r6  00000000
r7  00000000 r8  00000000 r9  00000000
r10 00000000 r11 00000000 r12 00000000
```

2) Click the left mouse button.

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

**zoom**

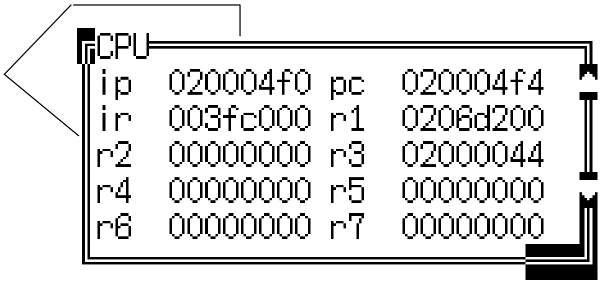
## *Moving a window*

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

There are two ways to move a window:

❑ By using the mouse
❑ By using the MOVE command

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



Point to the top edge or the left edge

```
CPU
ip  020004f0 pc  020004f4
ir  003fc000 r1  0206d200
r2  00000000 r3  02000044
r4  00000000 r5  00000000
r6  00000000 r7  00000000
```

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

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

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

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

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

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

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

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

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

**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 with the MOVE command in the same way that they are used for the SIZE command.

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

↓ Moves the active window down one line

↑ Moves the active window up one line

← Moves the active window left one character position

→ Moves the active window right one character position

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

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

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

# 5.6 Manipulating a Window’s Contents

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

**Note:**

You can scroll and edit only the **active window**. For information about selecting the active window, see Section 5.4.

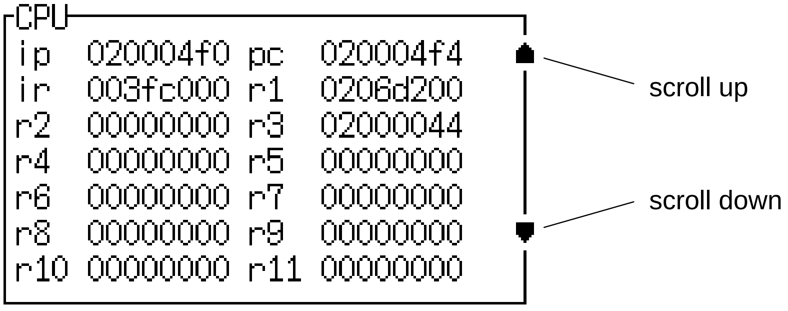
## *Scrolling through a window’s contents*

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

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

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

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



```
CPU
ip  020004f0 pc  020004f4
ir  003fc000 r1  0206d200
r2  00000000 r3  02000044
r4  00000000 r5  00000000
r6  00000000 r7  00000000
r8  00000000 r9  00000000
r10 00000000 r11 00000000
```

To scroll window contents up or down:

↖ 1) Point to the appropriate scroll arrow.

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

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

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

---

key

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 window contents, one window length at a time. You can use CONTROL PAGE UP to scroll up through an array of structures displayed in a DISP window.

PAGE DOWN

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

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

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

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

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

← In the FILE window, pressing this key scrolls the display left eight characters at a time. In other windows, it moves the field cursor left one field; at the first field on a line, it wraps back to the last fully displayed field on the previous line.

→ In the FILE window, pressing this key scrolls the display right eight characters at a time. In other windows, it moves the field cursor right one field; at the last field on a line, it wraps around to the first field on the next line.

### Editing the data displayed in windows

You can edit the data displayed in the MEMORY, CPU, DISP, and WATCH windows by using an overwrite "click-and-type" method or by using commands that change the values. (This is described in detail in Section 9.3, *Basic Methods for Changing Data Values*.)

**Note:**

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

❑ In the FILE and DISASSEMBLY windows, pressing F9 or the mouse button sets or clears a breakpoint on any line of code that you select. You can't modify text in a FILE or DISASSEMBLY window.

❑ In the CALLS window, pressing F9 or the mouse button shows the source for the function named on the selected line.

## 5.7 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 may choose to open DISP and WATCH windows and additional MEMORY windows.

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

1) Make the appropriate window active.

2) Press F4 .

**Note:**

You cannot close the default MEMORY window.

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

```
wr ⏎
```

When you close a window, the debugger remembers the window’s size and position. The next time you open the window, it will have the same size and position. That is, if you close the CALLS window, then reopen it, it will have the same size and position as it did before you closed it. Since you can open numerous DISP and MEMORY windows, when you open one, it will occupy the same position as the last one of that type that you closed.

Chapter 6

# 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. Also included is information about entering operating-system commands and defining your own command strings.

**Topics**

## 6.1 Entering Commands From the Debugger 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 current topic. Chapter 14, *Summary of Commands and Special Keys*, summarizes all of the debugger commands with an alphabetic reference.

Although there are various methods for entering most of the commands, **all** debugger commands can be entered by typing them on the command line in the COMMAND window. Figure 6–1 shows the COMMAND window.

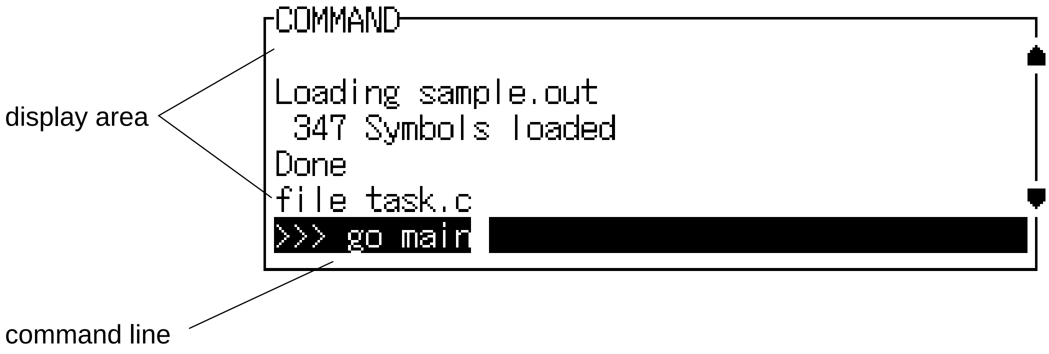
The COMMAND window serves two purposes:

- The **command line** portion of the window provides you with an area for entering commands. For example, the command line in Figure 6–1 shows that a GO command was typed in (but not yet entered).

- The **display area** provides the debugger with a space for echoing commands, displaying command output, or displaying errors and messages for you to read. For example, the command output in Figure 6–1 shows the messages that are displayed when you first bring up the debugger and also shows that a FILE TASK.C command was entered.

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

Figure 6–1. The COMMAND Window



```
Loading sample.out
 347 Symbols loaded
Done
file task.c
>>> go main
```

## How to type in and enter commands

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

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

1) Echoes the command to the display area,

2) Executes the command and displays any resulting output, and

3) Clears the command line when command execution completes.

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

| To... | Press... |
|---|---|
| Move back over text without erasing characters | CONTROL H or<br>BACK SPACE |
| Move forward through text without erasing characters | CONTROL L |
| Move back over text while erasing characters | DEL |
| Move forward through text while erasing characters | SPACE |
| Insert text into the characters that are already on the command line | INSERT |

**Notes:**

1) You cannot use the arrow keys to move through or edit text on the command line.

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

3) If you press ↵ when the cursor is in the middle of text, the debugger truncates the input text at the point where you press ↵.

## *Sometimes you can’t type a command*

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

- ❑ When you’re pressing the ALT key, typing certain letters causes the debugger to display a pulldown menu.
- ❑ When a pulldown menu is displayed, typing a letter causes the debugger to execute a selection from the menu.
- ❑ When you’re pressing the CONTROL key, pressing H or L moves the command-line cursor backward or forward through the text on the command line.
- ❑ When you’re editing a field, typing enters a new value in the field.
- ❑ When you’re using the MOVE or SIZE command interactively, pressing keys affects the size or position of the active window. Before you can enter any more commands, you must press ESC to terminate the interactive moving or sizing.
- ❑ When you’ve brought up a dialog box, typing enters a parameter value for the current field in the box. For more information on dialog boxes, see Section 6.3.

## *Using the command history*

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

Use these keystrokes to move through the command history.

| To... | Press... |
|---|---|
| Repeat the last command that you entered | F2 |
| 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 |

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.

For information about using the parallel debug manager's command history, see page DB:4-19.

## *Clearing the display area*

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

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

**cls**

## *Recording information from the display area*

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

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

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

  **dlog** *filename*

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

- To end the recording session, enter:

  **dlog close** ↵

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

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

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

- **Creating a new log file.** If you use the DLOG command without one of the optional parameters, the debugger creates a new file that it records the information into. If you are recording to a log file already, entering a new DLOG command and filename closes the previous log file and opens a new one.
- **Appending to an existing file.** Use the **a** parameter to open an existing file and append the information in the display area to it.
- **Writing over an existing file.** Use the **w** parameter to open an existing file and write over the current contents of the file. Note that this is the default action if you specify an existing filename without using either the **a** or **w** options; you will lose the contents of an existing file if you don't use the **a** option.

For more information about the parallel debug manager version of the DLOG command, see page DB:4-15.

## 6.2 Using the Menu Bar and the Pulldown Menus

In all three of the debugger modes, you'll see a menu bar at the top of the screen. The menu selections offer you an alternative method for entering many of the debugger commands. Figure 6–3 points out the menu bar in a mixed-mode display. There are several ways to use the selections on the menu bar, depending on whether the selection has a pulldown menu or not.

Several of the selections on the menu bar have pulldown menus; if they could all be pulled down at once, they'd look like Figure 6–2.

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

Figure 6–2. All of the Pulldown Menus (Basic Debugger Display for the Simulator)

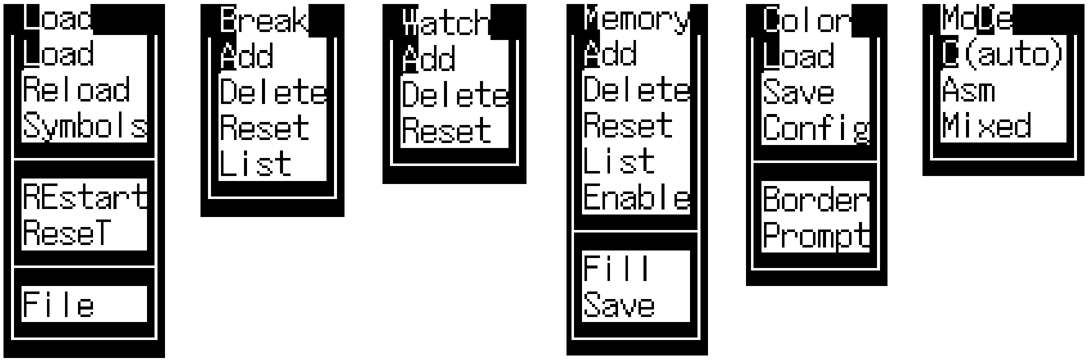
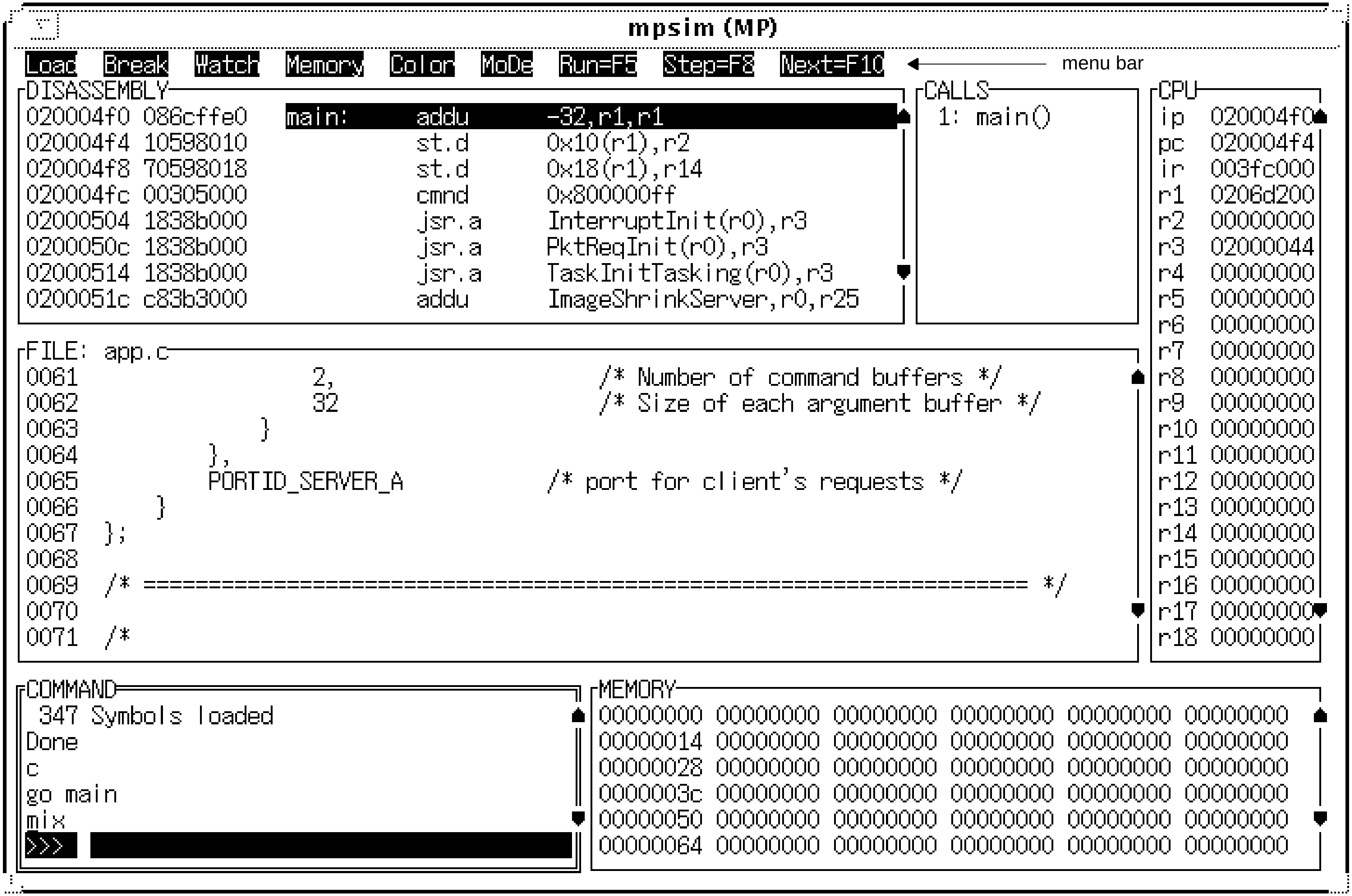


Figure 6–3. The Menu Bar in the Basic Debugger Display



```
DISASSEMBLY
020004f0 086cffe0   main:     addu      -32,r1,r1
020004f4 10598010             st.d      0x10(r1),r2
020004f8 70598018             st.d      0x18(r1),r14
020004fc 00305000             cmnd      0x800000ff
02000504 1838b000             jsr.a     InterruptInit(r0),r3
0200050c 1838b000             jsr.a     PktReqInit(r0),r3
02000514 1838b000             jsr.a     TaskInitTasking(r0),r3
0200051c c83b3000             addu      ImageShrinkServer,r0,r25
```

```
CALLS
 1: main()
```

```
CPU
ip   020004f0
pc   020004f4
ir   003fc000
r1   0206d200
r2   00000000
r3   02000044
r4   00000000
r5   00000000
r6   00000000
r7   00000000
r8   00000000
r9   00000000
r10  00000000
r11  00000000
r12  00000000
r13  00000000
r14  00000000
r15  00000000
r16  00000000
r17  00000000
r18  00000000
```

```
FILE: app.c
0061                  2,                      /* Number of command buffers */
0062                  32                      /* Size of each argument buffer */
0063              }
0064          },
0065          PORTID_SERVER_A             /* port for client's requests */
0066      }
0067  };
0068
0069  /* ==================================================================== */
0070
0071  /*
```

```
COMMAND
 347 Symbols loaded
Done
c
go main
mix
>>>
```

```
MEMORY
00000000 00000000 00000000 00000000 00000000 00000000
00000014 00000000 00000000 00000000 00000000 00000000
00000028 00000000 00000000 00000000 00000000 00000000
0000003c 00000000 00000000 00000000 00000000 00000000
00000050 00000000 00000000 00000000 00000000 00000000
00000064 00000000 00000000 00000000 00000000 00000000
```

## *Using the pulldown menus*

There are several ways to display the pulldown menus and then execute your selections from them. Executing a command from a menu is similar to executing a command by typing it in with these exceptions:

❑ If you select a command that has no parameters or only optional parameters, then 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 one of the appropriate selections in the menu bar.

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

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

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

**Mouse method 2**

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

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

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

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

key

**Keyboard method 1**

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

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

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

**Keyboard method 2**

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

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

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

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

## *Escaping from the pulldown menus*

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

■ Press ESC

**or**

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

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

## *Using menu bar selections that don't have pulldown menus*

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

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

There are two ways to execute these choices.

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

2) Click the left mouse button.

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

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

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

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

## 6.3 Using Dialog Boxes

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

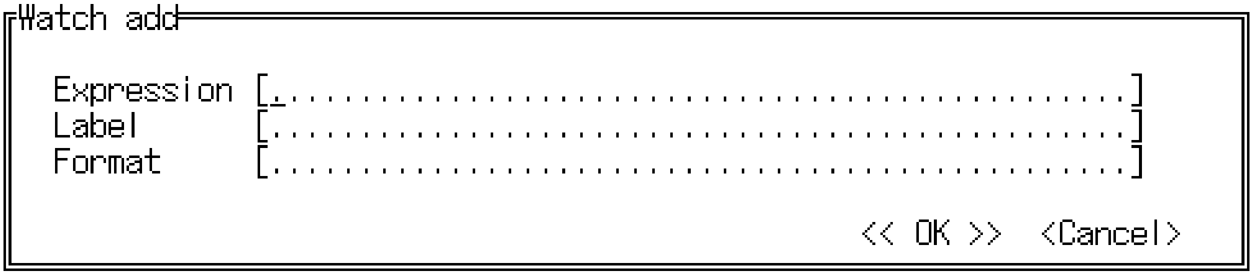
Some debugger commands have very simple dialog boxes that provide you with an alternative method for typing in values. Other commands, such as analysis commands, have more complex dialog boxes; in addition to typing in values, you may be asked to make selections from a list of predefined parameters.

### *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 three parameters:

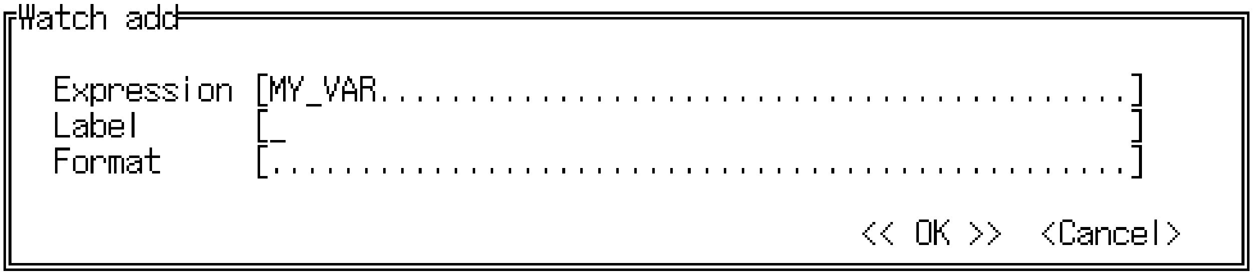
**wa** *expression* [,[ *label*] [, *display format*]]

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:



```
Watch add

  Expression [_.............................................]
  Label      [..............................................]
  Format     [..............................................]

                                          << OK >>  <Cancel>
```

You can enter an *expression* just as you would if you were to type the WA command; then press TAB or ↓. The cursor moves down to the next parameter:



```
Watch add

  Expression [MY_VAR........................................]
  Label      [_                                             ]
  Format     [..............................................]

                                          << OK >>  <Cancel>
```

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

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

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

- When you display a dialog box for the first time during a debugging session, the parameter fields are empty. When you bring up the same dialog box again, though, the box displays the last values that you entered. (This is similar to having a command history.) If you want to use the same value, just press TAB or ↓ to move to the next parameter.
- You can edit what you type (or values that remain from a previous entry) in the same way that you can edit text on the command line. For more information on editing text on the command line, see Section 6.1.

When you've entered a value for the final parameter, point and click on <OK> to save your changes, or <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.

## *Selecting parameters in a dialog box*

More complex dialog boxes, such as those associated with analysis commands, allow you to:

❑ **Enter text.** Entering text in a more complex dialog box is the same as entering text on the command line. For more information, refer to the *Entering text in a dialog box* discussion on DB:6-12.

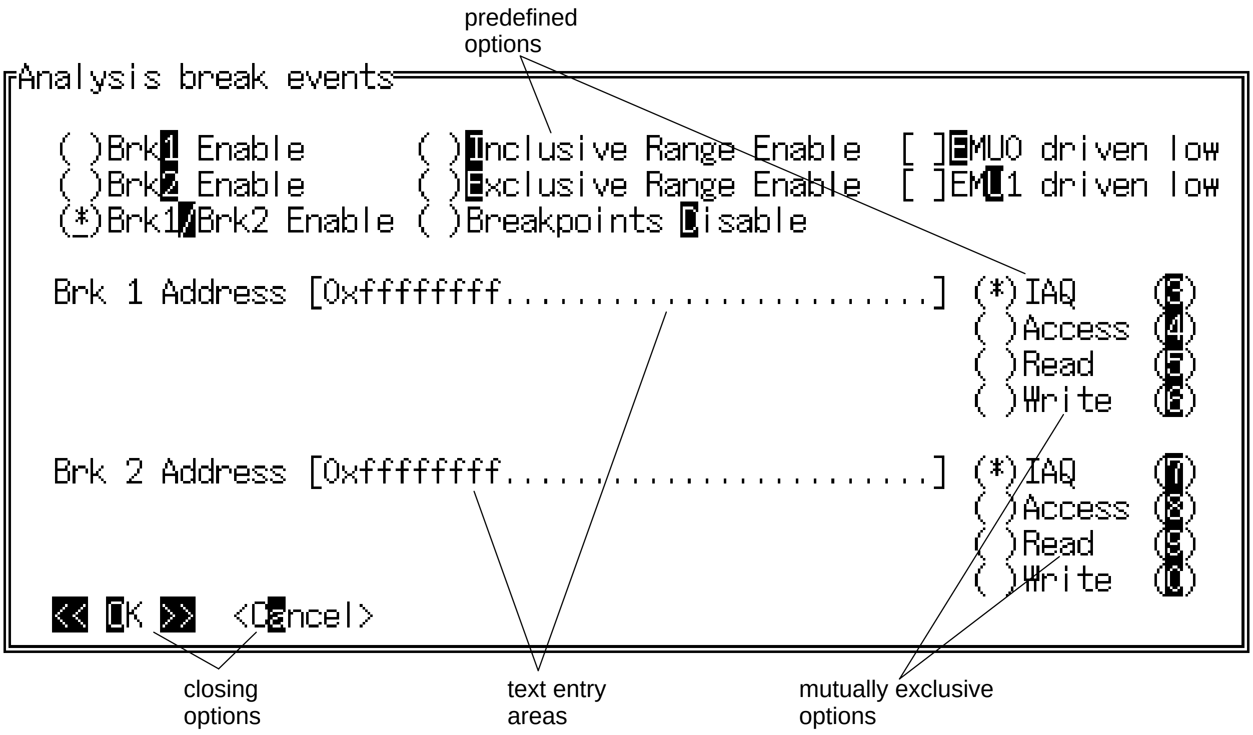
❑ **Choose from a list of predefined options.** There are two types of predefined options in a dialog box. The first type of option allows you to enable one or more predefined options. Options of the second type are **mutually exclusive**; therefore, you can enable only one at a time.

Valid options (of the opened dialog box) are listed for you so that all you have to do is point and click to make your selections.

❑ **Close the dialog box.** The more complex dialog boxes do not close automatically. They allow you the option of saving or discarding any changes you made to your parameter choices. All you have to do to close the dialog box is point and click on the appropriate option, either <OK> or <Cancel>.

Figure 6–4 shows you the components of a complex dialog box used with the analysis module.

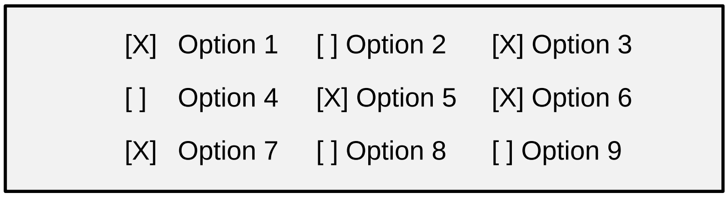
Figure 6–4. The Components of a Dialog Box



```
Analysis break events

  ( )Brk1 Enable      ( )Inclusive Range Enable  [ ]EMU0 driven low
  ( )Brk2 Enable      ( )Exclusive Range Enable  [ ]EMU1 driven low
  (*)Brk1/Brk2 Enable ( )Breakpoints Disable

  Brk 1 Address [0xffffffff.........................] (*)IAQ    (3)
                                                      ( )Access (4)
                                                      ( )Read   (5)
                                                      ( )Write  (6)

  Brk 2 Address [0xffffffff.........................] (*)IAQ    (7)
                                                      ( )Access (8)
                                                      ( )Read   (9)
                                                      ( )Write  (0)
  << OK >>  <Cancel>
```

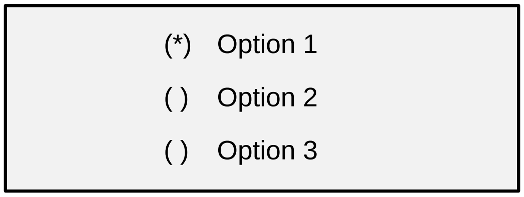
When you display a dialog box for the first time during a debugging session, nothing is enabled. When you bring up the same dialog box again, though, your previous selections are remembered. (This is similar to having a command history.)

As Figure 6–4 shows, options are preceded by either square brackets or parentheses; mutually exclusive options are preceded by parentheses. Enabling options preceded by square brackets is like turning a switch on and off. When the option is enabled, the debugger displays an X inside the brackets preceding the option. You can enable as many of these options as you want:



```
[X]  Option 1   [ ] Option 2    [X] Option 3
[ ]  Option 4   [X] Option 5    [X] Option 6
[X]  Option 7   [ ] Option 8    [ ] Option 9
```

Mutually exclusive options, however, are enabled when the debugger displays an asterisk inside the parentheses preceding your selection. The following example illustrates this:



```
(*)  Option 1
( )  Option 2
( )  Option 3
```

Notice that only one mutually exclusive option is enabled at a time and that there is always an enabled option. There are several ways to enable both types of options:

1) Point the cursor at the option you want to enable.

2) Click the left mouse button. This enables the event and displays an X next to the option (or an asterisk next to a mutually exclusive option).

Repeat these two steps to disable an option. When the X is no longer displayed, that option has been disabled. For mutually exclusive options, you disable an option by enabling another option—one option is selected at all times. The additional selections that you enter in the dialog box determine whether or not the set of mutually exclusive options is active.

key

### Keyboard Method 1

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

X 2) Press and release the key that corresponds to the highlighted letter or number of the option you want to enable. The debugger displays an X (or asterisk) next to the option, indicating that selection is enabled.

Repeat these two steps to disable an option. When the X (or asterisk) is no longer displayed, that option has been disabled.

### Keyboard Method 2

TAB 1) Press the TAB key to move throughout the dialog box until your cursor points to the option you want to enable.

↓ ↑ 2) Use the arrow keys to move up and down or left and right.

When you enable a mutually exclusive option, moving the arrow keys alone will place an asterisk inside the parentheses, indicating that the option is enabled. However, to enable an option preceded by square brackets, you must:

SPACE Press the SPACE bar. The debugger displays an X next to your selection, thus enabling that particular option.

*or*

F9 Press the F9 key. The debugger displays an X next to your selection, thus enabling that particular option.

Repeat these steps to disable an option.

## *Closing a dialog box*

The more complex dialog boxes do not close automatically; the debugger expects input from you. When you close a dialog box, you can:

- ❑ Save the changes you made

*or*

- ❑ Discard any of the changes you made

**Note:**

The default option, <OK>, is highlighted; clicking on this option saves your changes and closes the dialog box.

There are several ways to close a dialog box:

↖ 1) Point the cursor at <OK> to close the dialog box and save your changes. Or you can opt to discard your changes by pointing the cursor at <Cancel>. The <Cancel> option restores previous values.

2) Click the left mouse button. This executes your choice and closes the dialog box.

**Keyboard Method 1**

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

X 2) Press and release the O key to save your changes. Press and release the A key to discard your changes. Both of these actions execute your choice and close the dialog box.

**Keyboard Method 2**

TAB 1) Press the TAB key to move through the dialog box until your cursor is in the <OK> or <Cancel> field.

← → 2) Use the arrow keys to switch between <OK> and <Cancel>.

↵ 3) Press the ↵ key to accept your selection. This executes your choice and closes the dialog box.

## 6.4 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** The TAKE command tells the debugger to read and execute commands from a batch file. A batch file can call another batch file; they can be nested in this manner up to 10 deep. To halt execution of a batch file, press ESC .

The format for the TAKE command is:

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

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

    **setenv D_DIR ”***pathname;pathname***”**

    This allows you to name several directories that can be searched. If you often use the same directories, it may be convenient to set D_DIR in your .cshrc file.
- ❑ By default, the debugger echoes the commands in the display area of the COMMAND window and updates the display as it reads commands from the batch file.
  - ■ If you don’t use the *suppress echo flag* parameter, or if you use it but supply a nonzero value, 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.

For information about the parallel debug manager version of the TAKE command, see page DB:4-14.

## *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 COMMAND window display area.

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 should not be in quotes.)

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

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

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

For more information about the parallel debug manager version of the ECHO command, see page DB:4-18.

## *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 commands
      .
[else
      .
      debugger commands
      .]
endif
```

  The debugger includes some predefined constants for use with IF. These constants evaluate to 0 (false) or 1 (true). Table 6–1 shows the constants and their corresponding tools.

Table 6–1. Predefined Constant for Use With Conditional Commands

| Constant | Debugger Tool |
|---|---|
| $$EMU$$ | emulator |
| $$SIM$$ | simulator |
| $$MVP_MP$$ | MP debugger |
| $$MVP_PP$$ | PP debugger |

  If the Boolean expression evaluates to true (1), the debugger executes all commands between the IF and ELSE or ENDIF. Note that the ELSE portion of the command is optional. (For more information about expressions and expression analysis, see Chapter 15, *Basic Information About C Expressions*.)

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

```
loop expression
      .
      debugger commands
      .
endloop
```

These looping commands evaluate in the same way as in the run conditional command expression. (For more information about expressions and expression analysis, see Chapter 15, *Basic Information About C Expressions*.)

- 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
  runb
  .
  .
  .
  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 has one of the following operators as the highest precedence operator in the expression (these operators are described in Chapter 15, *Basic Information About C Expressions*):

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

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

  ```
  loop !0
  step
  ? PC
  ? R1
  endloop
  ```

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

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

See *Controlling PDM command execution*, page DB:4-16, for more information about the parallel debug manager versions of the IF and LOOP commands.

## 6.5 Defining Your Own Command Strings

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

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

**alias** [*alias name* [, “*command string*”] ]

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

❑ **Aliasing several commands.** The *command string* can contain more than one command—just separate the commands with semicolons.

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

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

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

❑ **Supplying parameters to the command string.** The *command string* can define parameters that you’ll supply later. To do this, use a percent sign and a number (%1) to represent the parameter that will be filled in later. 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”
```

Then you could enter:

```
mfil 0x2000080,0x18,0x1234abcd
```

The first value (0x2000080) would be substituted for the first FILL parameter and the MEM parameter (%1). The second and third values would be 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 will list the aliases and their definitions in the COMMAND window.

For example, assume that the init and mfil aliases had been defined as shown in the previous two examples. If you entered:

**`alias`** ↵

you’d see:

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

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

For example, if you had defined the init alias as shown in the first example above, you could enter:

**`alias init`** ↵

Then you’d see:

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

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

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

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

**unalias** *alias name*

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

**`unalias *`**

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

**Notes:**

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

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

For information about the parallel debug manager versions of the ALIAS and UNALIAS commands, see page DB:4-21.

Chapter 7

# Defining a Memory Map

Before you begin a debugging session, you must supply the debugger with a memory map. The memory map tells the debugger which areas of memory it can and can't access. Note that the commands described in this chapter can also be executed by using the Memory pulldown menu.

**Topics**

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

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

**Note:**

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

A special default initialization batch file called init.cmd is included with the debugger package. It defines a memory map for the MP and PP versions of the debugger. This memory map may be sufficient when you first begin using the debugger. However, the debugger provides a complete set of memory-mapping commands that let you modify the default memory map or define a new memory map.

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

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

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

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

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

1) It checks to see whether you've used the –t debugger option. The –t option allows you to specify a batch file other than the initialization batch file shipped with the debugger. If it finds the –t option, the debugger reads and executes the specified file.
2) If you don't use the –t option, the debugger looks for the default initialization batch file called *init.cmd*. If the debugger finds the file, it reads and executes the file.

## *Potential memory map problems*

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

- ❑ **Accessing invalid memory addresses.** If you don't supply a batch file containing memory-map commands, then the debugger is initially unable to access any target memory locations. Invalid memory addresses and their contents are highlighted in the data-display windows. (On color monitors, invalid memory locations, by default, are displayed in red.)
- ❑ **Accessing an undefined or protected area.** When memory mapping is enabled, the debugger checks each of its memory accesses against the memory map. If you attempt to access an undefined or protected area, the debugger displays an error message.
- ❑ **Loading a COFF file with sections that cross a memory range.** Be sure that the map ranges you specify in a COFF file match those that you define with the MA command (described on page DB:7-7). Alternatively, you can turn memory mapping off during a load by using the MAP OFF command (see page DB:7-10).

## 7.2 Sample Memory Maps

The init.cmd file contains default memory maps for the MP and PPs. You can use the file as is, edit it, or create your own memory map batch file.

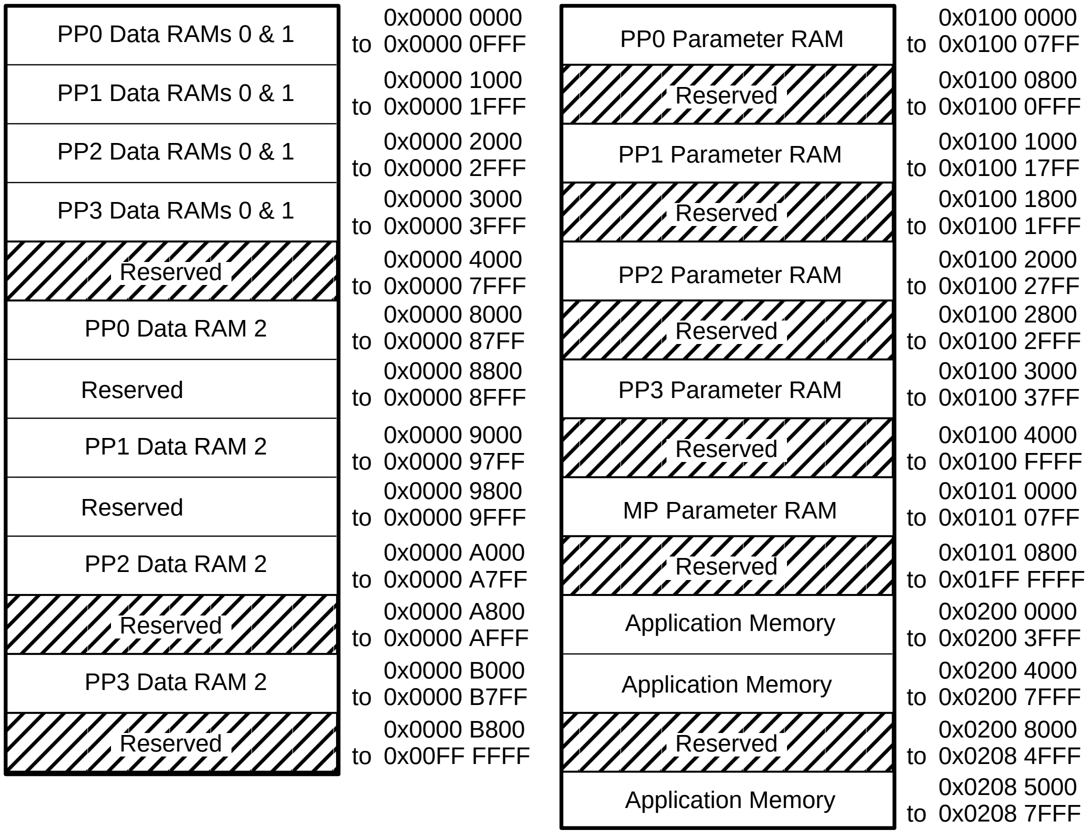
❑ Figure 7–1 shows the memory map commands that are defined in the init.cmd file for the MP debugger. The MA (map add) commands define valid memory ranges and identify the read/write characteristics of the memory ranges. Figure 7–2 illustrates the memory map defined by the default initialization batch file.

❑ Figure 7–3 shows the memory map commands that are defined in the init.cmd file for PP debuggers. Figure 7–4 illustrates the memory map defined by the default initialization batch file.

Figure 7–1. Memory Map Commands in the Sample Initialization Batch File (MP Version)

```
MA 0x01010000,0x800,RAM
MA 0x01000000,0x800,RAM
MA 0x01001000,0x800,RAM
MA 0x01002000,0x800,RAM
MA 0x01003000,0x800,RAM
MA 0x00000000,0x1000,RAM
MA 0x00001000,0x1000,RAM
MA 0x00002000,0x1000,RAM
MA 0x00003000,0x1000,RAM
MA 0x00008000,0x800,RAM
MA 0x00009000,0x800,RAM
MA 0x0000A000,0x800,RAM
MA 0x0000B000,0x800,RAM
MA 0x02000000,0x4000,SRAM0,4,64
MA 0x02004000,0x4000,DRAM2,4,32
MA 0x02085000,0x3000,DRAM3,2,64
```

Figure 7–2. Memory Map for MP Local Memory



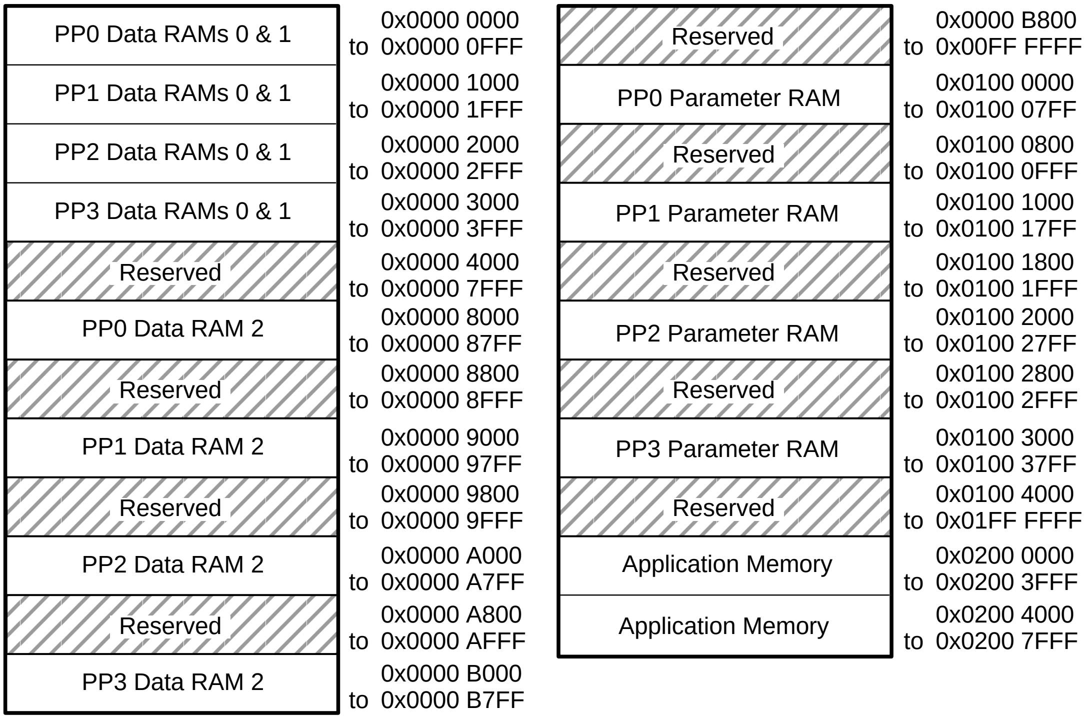
| | |
|---|---|
| PP0 Data RAMs 0 & 1 | 0x0000 0000<br>to 0x0000 0FFF |
| PP1 Data RAMs 0 & 1 | 0x0000 1000<br>to 0x0000 1FFF |
| PP2 Data RAMs 0 & 1 | 0x0000 2000<br>to 0x0000 2FFF |
| PP3 Data RAMs 0 & 1 | 0x0000 3000<br>to 0x0000 3FFF |
| Reserved | 0x0000 4000<br>to 0x0000 7FFF |
| PP0 Data RAM 2 | 0x0000 8000<br>to 0x0000 87FF |
| Reserved | 0x0000 8800<br>to 0x0000 8FFF |
| PP1 Data RAM 2 | 0x0000 9000<br>to 0x0000 97FF |
| Reserved | 0x0000 9800<br>to 0x0000 9FFF |
| PP2 Data RAM 2 | 0x0000 A000<br>to 0x0000 A7FF |
| Reserved | 0x0000 A800<br>to 0x0000 AFFF |
| PP3 Data RAM 2 | 0x0000 B000<br>to 0x0000 B7FF |
| Reserved | 0x0000 B800<br>to 0x00FF FFFF |

| | |
|---|---|
| PP0 Parameter RAM | 0x0100 0000<br>to 0x0100 07FF |
| Reserved | 0x0100 0800<br>to 0x0100 0FFF |
| PP1 Parameter RAM | 0x0100 1000<br>to 0x0100 17FF |
| Reserved | 0x0100 1800<br>to 0x0100 1FFF |
| PP2 Parameter RAM | 0x0100 2000<br>to 0x0100 27FF |
| Reserved | 0x0100 2800<br>to 0x0100 2FFF |
| PP3 Parameter RAM | 0x0100 3000<br>to 0x0100 37FF |
| Reserved | 0x0100 4000<br>to 0x0100 FFFF |
| MP Parameter RAM | 0x0101 0000<br>to 0x0101 07FF |
| Reserved | 0x0101 0800<br>to 0x01FF FFFF |
| Application Memory | 0x0200 0000<br>to 0x0200 3FFF |
| Application Memory | 0x0200 4000<br>to 0x0200 7FFF |
| Reserved | 0x0200 8000<br>to 0x0208 4FFF |
| Application Memory | 0x0208 5000<br>to 0x0208 7FFF |

Figure 7–3. Memory Map Commands in the Sample Initialization Batch File (PP Version)

```
MA 0x01000000,0x800,RAM
MA 0x01001000,0x800,RAM
MA 0x01002000,0x800,RAM
MA 0x01003000,0x800,RAM
MA 0x00000000,0x1000,RAM
MA 0x00001000,0x1000,RAM
MA 0x00002000,0x1000,RAM
MA 0x00003000,0x1000,RAM
MA 0x00008000,0x800,RAM
MA 0x00009000,0x800,RAM
MA 0x0000A000,0x800,RAM
MA 0x0000B000,0x800,RAM
MA 0x02000000,0x4000,SRAM0,4,64
MA 0x02004000,0x4000,DRAM2,4,32
```

Figure 7–4. Memory Map for PP Local Memory



| | |
|---|---|
| PP0 Data RAMs 0 & 1 | 0x0000 0000<br>to 0x0000 0FFF |
| PP1 Data RAMs 0 & 1 | 0x0000 1000<br>to 0x0000 1FFF |
| PP2 Data RAMs 0 & 1 | 0x0000 2000<br>to 0x0000 2FFF |
| PP3 Data RAMs 0 & 1 | 0x0000 3000<br>to 0x0000 3FFF |
| Reserved | 0x0000 4000<br>to 0x0000 7FFF |
| PP0 Data RAM 2 | 0x0000 8000<br>to 0x0000 87FF |
| Reserved | 0x0000 8800<br>to 0x0000 8FFF |
| PP1 Data RAM 2 | 0x0000 9000<br>to 0x0000 97FF |
| Reserved | 0x0000 9800<br>to 0x0000 9FFF |
| PP2 Data RAM 2 | 0x0000 A000<br>to 0x0000 A7FF |
| Reserved | 0x0000 A800<br>to 0x0000 AFFF |
| PP3 Data RAM 2 | 0x0000 B000<br>to 0x0000 B7FF |

| | |
|---|---|
| Reserved | 0x0000 B800<br>to 0x00FF FFFF |
| PP0 Parameter RAM | 0x0100 0000<br>to 0x0100 07FF |
| Reserved | 0x0100 0800<br>to 0x0100 0FFF |
| PP1 Parameter RAM | 0x0100 1000<br>to 0x0100 17FF |
| Reserved | 0x0100 1800<br>to 0x0100 1FFF |
| PP2 Parameter RAM | 0x0100 2000<br>to 0x0100 27FF |
| Reserved | 0x0100 2800<br>to 0x0100 2FFF |
| PP3 Parameter RAM | 0x0100 3000<br>to 0x0100 37FF |
| Reserved | 0x0100 4000<br>to 0x01FF FFFF |
| Application Memory | 0x0200 0000<br>to 0x0200 3FFF |
| Application Memory | 0x0200 4000<br>to 0x0200 7FFF |

## 7.3 Identifying Usable Memory Ranges

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

**ma** *address, length, type* [*, pagesize*] [*, bussize*]

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

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

- The *length* parameter defines the length of the range. This parameter can be any C expression. If you're using the simulator to define an external memory range, the *length* that you supply must be a multiple of the *pagesize* that you define.

**Note:**

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

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

doesn't equal

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

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

❑ The *type* parameter identifies one the following MVP memory types and its associated read/write characteristics of the memory range. The *type* must be one of these keywords shown in Table 7–1 if you're using the simulator, or in Table 7–2 if you're using the emulator.

Table 7–1. Memory-Type Keywords for Use With the Simulator

| To identify this kind of simulator memory | That has these read/write characteristics, | Use this keyword as the *type* parameter |
|---|---|---|
| Pipelined one cycle/column | Read/write memory | **SRAM0** or **DRAM0** |
| Unpipelined one cycle/column (default memory type) | Read/write memory | **SRAM, DRAM, SRAM1, DRAM1, WR,** or **RAM** |
| | Read-only memory | **R**, **ROM**, or **READONLY** |
| | Write-only memory | **W***,* **WOM***,* or **WRITEONLY** |
| | No-access memory | **PROTECT** |
| Unpipelined two cycle/column | Read/write memory | **SRAM2** or **DRAM2** |
| Unpipelined three cycle/column | Read/write memory | **SRAM3** or **DRAM3** |

Table 7–2. Memory-Type Keywords for Use With the Emulator

| To identify this kind of emulator memory, | Use this keyword as the *type* parameter |
|---|---|
| Read-only memory | **R**, **ROM**, or **READONLY** |
| Write-only memory | **W***,* **WOM***,* or **WRITEONLY** |
| Read/write memory | **WR** or **RAM** |
| No-access memory | **PROTECT** |
| Input port | **IPORT** |
| Output port | **OPORT** |
| Input/output port | **IOPORT** |

❑ The *pagesize* parameter specifies a page boundary for the external memory. You can use this parameter with the simulator only.

| To select this page boundary (in bytes), | Use this as the *pagesize* parameter |
|---|---|
| 1 to 8[†] | 0 |
| 2K | 1 |
| 4K | 2 |
| 8K | 3 |
| 16K | 4 |
| 32K | 5 |
| 64K | 6 |
| 128K | 7 |

[†] To make the page boundary the same as the bus size of the memory access (1, 2, 4, or 8 bytes), use pagesize 0.

By default, the *pagesize* is set to 128K bytes.

❑ The *bussize* parameter specifies the bus size of the memory access. You can use this parameter with the simulator only. Setting the bus size determines the maximum number of bytes that the TC can transfer during each cycle. You can enter a bus size of 8, 16, 32, or 64 bits. The bus size defaults to 64 bits.

**Note:**

The *pagesize* and *bussize* parameter values are valid only for defining external memory ranges with the simulator. These values are ignored if you're defining an on-chip memory range or if you're using the emulator.

## 7.4 Enabling Memory Mapping

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

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

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

**Note:**

When memory mapping is enabled, you cannot:

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

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

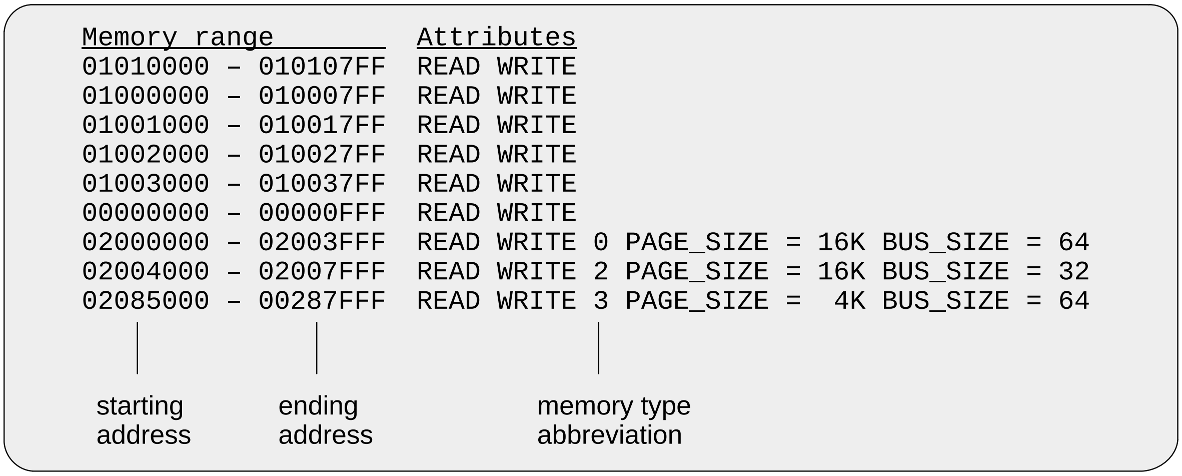
```
Error in expression
```

## 7.5 Checking the Memory Map

**ml** If you want to see which memory ranges are defined, use the ML 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. If a memory range defines external memory, this command also lists the memory type, page size (in bytes), and bus size (in bits) of the range. For example, if you're using the default memory map for the simulator version of the MP debugger and you enter the ML command, the debugger displays this:



```
Memory range         Attributes
01010000 – 010107FF  READ WRITE
01000000 – 010007FF  READ WRITE
01001000 – 010017FF  READ WRITE
01002000 – 010027FF  READ WRITE
01003000 – 010037FF  READ WRITE
00000000 – 00000FFF  READ WRITE
02000000 – 02003FFF  READ WRITE 0 PAGE_SIZE = 16K BUS_SIZE = 64
02004000 – 02007FFF  READ WRITE 2 PAGE_SIZE = 16K BUS_SIZE = 32
02085000 – 00287FFF  READ WRITE 3 PAGE_SIZE =  4K BUS_SIZE = 64
```

The memory type abbreviation identifies the kind of memory defined for the external memory range:

| If you see this memory type abbreviation, | It identifies this kind of memory | And represents this keyword as the *type* parameter |
|---|---|---|
| 0 | Pipelined one cycle/column | **SRAM0** or **DRAM0** |
| 1 | Unpipelined one cycle/column | **SRAM**, **DRAM**, **SRAM1**, **DRAM1**, **R**, **ROM**, **READONLY**, **W**, **WOM**, **WRITEONLY**, **WR**, **RAM**, or **PROTECT** |
| 2 | Unpipelined two cycle/column | **SRAM2** or **DRAM2** |
| 3 | Unpipelined three cycle/column | **SRAM3** or **DRAM3** |

## 7.6 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. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. 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* [*, pagesize*] [*, bussize*]

The MA command is described in detail on page DB:7-7.

### *Returning to the original memory map*

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

Suppose, for example, that you had set up your memory map in a batch file named *mem.map*. You could 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.

## 7.7 Using Multiple Memory Maps for Multiple Target Systems (Emulator Only)

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

**Step 1:** Let the initialization batch file define the memory map for one of your applications.

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

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

(Of course, you can include any other appropriate commands in this batch file.)

**Step 3:** Invoke the debugger as usual.

**Step 4:** The debugger reads the initialization batch file during invocation. Before you begin debugging, read in the commands from the new batch file:

```
take filename.x ↵
```

This redefines the memory map for the current debugging session.

You can also use the –t option instead of using the TAKE command when you invoke the debugger. The –t option allows you to specify a new batch file to be used instead of the default initialization batch file.

## 7.8 Managing Memory and Cache

If you use the MP debugger to read an external memory address, the data cache is always checked for the address. If the address is present, the data is read from the data cache. Otherwise, the data is read from external memory. Data is not read from the instruction cache, because external memory and the instruction cache are assumed to be identical. Note that if the address that is being read is within the data cache, you can’t read the corresponding data in external memory without flushing the cache.

If you modify MVP external memory using the MP debugger, data is written to the following locations:

❑ External memory
❑ The data cache, if the address is present
❑ The instruction cache, if the address is present

When using the PP debugger, data is always read from external memory. Since the instruction cache and external memory are assumed to be identical, data is never read from the instruction cache. During a write to external memory, data is written to external memory and to the instruction cache if the address is present.

Chapter 8

# Loading, Displaying, and Running Code

The main purpose of a debugging system is to allow you to load and run your programs in a test environment. This chapter tells you how to load your programs into the debugging environment, run them on the target system, and view the associated source code. Many of the commands described in this chapter can also be executed from the Load pulldown menu.

**Topics**

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

The debugger has three code-display windows:

❑ The DISASSEMBLY window displays the reverse assembly of program memory contents.

❑ The FILE window displays any text file; its main purpose is to display C source files.

❑ The CALLS window identifies the current function (when C code is running).

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

❑ The mode you select, and

❑ Whether your program is currently executing assembly language code or C code.

Here's a summary of the modes and displays; for a complete description of the three debugging modes, see Section 5.1, *Debugging Modes and Default Displays*.

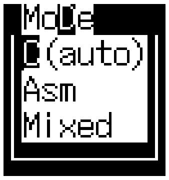
| Use this mode | To view | The debugger uses these code-display windows |
|---|---|---|
| assembly mode | *assembly language code only* (even if your program is executing C code) | ❑ DISASSEMBLY |
| auto mode | *assembly language code* (when that's what your program is running) | ❑ DISASSEMBLY |
| auto mode | *C code only* (when that's what your program is running) | ❑ FILE<br>❑ CALLS |
| mixed mode | *both assembly language and C code* | ❑ DISASSEMBLY<br>❑ FILE<br>❑ CALLS |

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

### Selecting a debugging mode

When you first invoke the debugger, it automatically comes up in auto mode. You can then choose assembly or mixed mode. There are several ways to do this.

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



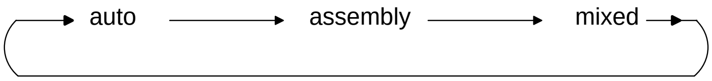
1) Point to the menu name.

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

3) Release the mouse button.

For more information about the pulldown menus, see Section 6.2, *Using the Menu Bar and the Pulldown Menus*.

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



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

**c** Changes from the current mode to auto mode.

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

**mix** Changes from the current mode to mixed mode.

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

## 8.2 Displaying Your Source Programs (or Other Text Files)

The debugger displays two types of code:

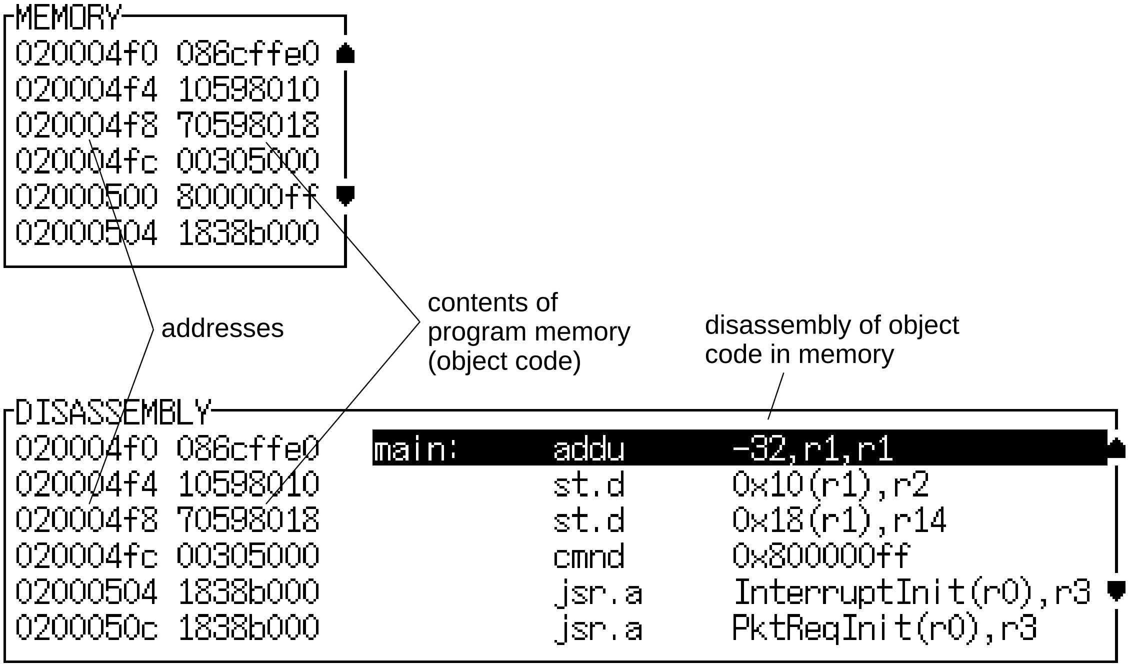
- It displays **assembly language code** in the DISASSEMBLY window in auto, assembly, or mixed mode.
- It displays **C code** in the FILE window in auto and mixed modes.

The DISASSEMBLY and FILE windows are intended primarily for displaying code that the IP (for the MP) or IPE (for the PP) points to. By default, the FILE window displays the C source for the current function (if any), and the DISASSEMBLY window shows the current disassembly.

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

### *Displaying assembly language code*

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



```
MEMORY
020004f0 086cffe0
020004f4 10598010
020004f8 70598018
020004fc 00305000
02000500 800000ff
02000504 1838b000
```

```
DISASSEMBLY
020004f0 086cffe0   main:      addu     -32,r1,r1
020004f4 10598010              st.d     0x10(r1),r2
020004f8 70598018              st.d     0x18(r1),r14
020004fc 00305000              cmnd     0x800000ff
02000504 1838b000              jsr.a    InterruptInit(r0),r3
0200050c 1838b000              jsr.a    PktReqInit(r0),r3
```

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

---

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

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

**dasm** *address*
or **dasm** *function name*

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

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

**addr** *address*
or **addr** *function name*

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

## *Displaying C code*

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

❑ You can force the debugger to show C source by entering a FILE, FUNC, or ADDR command.

❑ In auto and mixed modes, the debugger automatically opens a FILE window if you’re currently running C code.

---

These commands are valid in C and mixed modes:

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

**file** *filename*

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

(Displaying a file **doesn’t** load that file’s object code. If you want to be able to run the program, you must load the file’s associated object code as described in Section 8.3.)

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

**func** *function name*
or **func** *address*

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

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

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

**addr** *address*
or **addr** *function name*

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

---

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

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

2) Click the left mouse button.

(If the CALLS window is active, you can also use the arrow keys and F9 to display the function; see the *CALLS window* discussion on page DB:5-14 for details.)

## *Displaying other text files*

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

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

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

## 8.3 Loading Object Code

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

### *Loading code while invoking the debugger*

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

If you want to load a file's symbol table only, use the –s option (this has the same effect as using the debugger's SLOAD command). To do this, enter SPAWN, enter the name of the object file, and specify –s.

## *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. When you execute a LOAD, the debugger sets the current IP (for the MP) or IPE (for the PP) to the entry point of the loaded code. The format for this command is:

**load** *object filename*

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

**reload** Use the RELOAD command to load only an object file without loading its associated symbol table. This is useful for reloading a program when memory has been corrupted. When you execute a RELOAD, the debugger sets the current IP or IPE to the entry point of the loaded code. The syntax 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.

**Note:**

When you use the LOAD or RELOAD command in the MP debugger, the MP data cache is first "cleaned." In other words, all of the modified (dirty) bits are written back to external memory. Then, the instruction and data caches are cleared (all of the DTAG and ITAG registers are cleared).

**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). In addition, you can use SLOAD if code has already been loaded into the simulator core by another debugger. SLOAD clears the existing symbol table before loading the new one but does not modify memory or set the program entry point.

## 8.4 Where the Debugger Looks for Source Files

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

❑ If you’re using LOAD, RELOAD, or SLOAD, you can specify the path as part of the filename.

❑ If you’re using the FILE command, you have several options:

- You can name additional directories by setting the D_SRC environment variable in the operating-system environment. The format for doing this is:

  **setenv D_SRC ”***pathname;pathname***”**

  This allows you to name several directories that the debugger can search. If you use the same directories often, it may be convenient to set the D_SRC environment variable in your .cshrc file. If you do this, then the list of directories is always available when you’re using the debugger.

- When you invoke the debugger, you can use the –i option to name additional source directories for the debugger to search. The format for this option is:

  **–i** *pathname*

  You can specify multiple pathnames by using several –i options (one pathname per option). The list of source directories that you create with –i options is valid until you quit the debugger.

**use**

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

  **use** *directory name*

  You can specify only one directory at a time.

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

## 8.5 Running Your Programs

To debug your programs, you must execute them on one of the MVP debugging tools (simulator or emulator). 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 or C code one statement at a time and update the display after each execution.

## *Defining the starting point for program execution*

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

❑ Finding its entry in the CPU window

**or**

❑ Finding the appropriately highlighted line in the FILE or DISASSEMBLY window. To do this, execute one of these commands:

**dasm** **ip** or **ipe**
or **addr** **ip** or **ipe**

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

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

**restart**
or **rest**

You can also access this command from the Load pulldown menu.

**Note:**

If you invoked the current PP debugging session with start-up code from the MP to unhalt the PP, after you enter the RESTART command, you must set the IPE register to provide the correct entry point. Use the ? or EVAL command described below.

However, if you used the PP compiler to create the object code that you're currently debugging, you don't need to reset the entry point after entering the RESTART command; the entry point is set automatically.

**?/eval** ❑ You can directly modify the contents of the IP or IPE with one of these commands:

**?ip =** *new value*
or **eval ipe =** *new value*

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

## *Running code*

The debugger supports several run commands.

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

**run** [*expression*]

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

- ❑ If you don’t supply an *expression*, the program executes until it encounters a breakpoint or until you press ESC or the left mouse button.
- ❑ If you supply a logical or relational *expression*, this becomes a conditional run (see page DB:8-18).
- ❑ If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger executes *count* instructions, halts, then updates the display.

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

**go** [*address*]

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

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

**return**
or **ret**

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

**runb** Use the RUNB (run benchmark) command to execute a specific section of code and count the number of clock cycles consumed by the execution. The format for this command is:

**runb**

Using the RUNB command to benchmark code is a multistep process described in Section 8.7, *Benchmarking*.

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

## *Single-stepping through code*

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

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

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

- If you don't supply an *expression*, the program executes a single statement, then halts.
- If you supply a logical or relational *expression*, this becomes a conditional single-step execution (see page DB:8-18).
- If you supply any other type of *expression*, the debugger treats the expression as a *count* parameter. The debugger single-steps *count* C or assembly language statements, depending on the type of code you're in.

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

**step** [*expression*]

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

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

See the *Managing interrupts while single-stepping* discussion, on page DB:A-3, for information on interrupts and the STEP command. See the *Understanding execution differences between RUN and STEP* discussion, on page DB:A-5, for more information on the RUN and STEP commands.

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

**cstep** [*expression*]

**next**
**cnext**

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

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

---

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

| | |
|---|---|
| F8 | Acts as a STEP command. |
| F10 | Acts as a NEXT command. |

---

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.

## *Running code while disconnected from the target system*

emulator only

**runf** Use the RUNF (run free) command to disconnect the emulator from the target system while code is executing. The format for this command is:

**runf**

When you enter RUNF, the debugger clears all breakpoints, disconnects the emulator from the target system, and causes the processor to begin execution at the current IP (for the MP) or current IPE (for the PP). You can quit the debugger, or you can continue to enter commands. However, any command that causes the debugger to access the target at this time will produce an error.

RUNF is useful in a multiprocessor system. It's also useful in a system in which several target systems share an emulator: RUNF enables you to disconnect the emulator from one system and connect it to another.

**halt** Use the HALT command to halt the target system after you've entered a RUNF command. The format for this command is:

**halt**

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

**reset** The RESET command resets the target processor. This is a *software* reset. The format for this command is:

**reset**

When you perform a RESET on the MP, an MVP reset is performed. In other words, the MP resets itself and the PPs.

When you execute RESET on a PP, the debugger performs a software reset by forcing all registers to their reset values. Note that RESET does not set the PP halt bit in the MP configuration register, so the processor is not halted. As a result, you can resume execution without running MP code to unhalt the PP.

## *Running code conditionally*

The RUN, STEP, CSTEP, NEXT, and CNEXT commands all have an optional *expression* parameter that can be a relational or logical expression. This type of expression has 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. 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 left mouse button
        halts execution)
   if (halted by breakpoint, not by ESC  or mouse button) go to top
end:
```

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

## 8.6 Halting Program Execution

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

▮▯ Click the left mouse button.

ESC Press the escape key.

After halting execution, you can continue program execution from the current IP (for the MP) or IPE (for the PP) by reissuing any of the run or single-step commands.

See Section A.2, *Halting Considerations*, for additional information about debugger halts.

## 8.7 Benchmarking

The debugger allows you to keep track of the number of CPU clock cycles consumed by a particular section of code. The debugger maintains the count in a pseudoregister named *CLK*.

Benchmarking code is a multiple-step process:

**Step 1:** Set a software or hardware breakpoint at the statement that marks the beginning of the section of code you'd like to benchmark.

**Step 2:** Set a software or hardware breakpoint at the statement that marks the end of the section of code you'd like to benchmark.

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

**Step 4:** Now enter the RUNB command:

```
runb ↵
```

When the processor halts at the second breakpoint, the value of CLK is valid. To display it, use the ? command or enter it into the WATCH window with the WA command. This value is valid until you enter another RUN command. You may also use the value of CLK in an expression.

**Notes:**

1) The value in CLK is valid only after you use a RUNB command that is terminated by a software or hardware breakpoint.

2) The simulator is not completely timing accurate.

3) The RUNB command also generates an EMU1 trigger out when a software or hardware breakpoint occurs.

If a RUNB command does not have access to the emulator resources necessary to perform an operation, the debugger generates an error, execution does not start, and the debugger does not respond to commands until you execute a HALT command.

Chapter 9

# Managing Data

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

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

**Topics**

## 9.1 Where Data Is Displayed

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

| Type of data | Window name and purpose |
|---|---|
| memory locations | **MEMORY windows**<br>Display the contents of a range of memory |
| register values | **CPU window**<br>Displays the contents of MVP registers |
| pointer data or selected variables of an aggregate type | **DISP windows**<br>Display the contents of aggregate types and show the values of individual members |
| selected variables (scalar types or individual members of aggregate types) and specific memory locations or registers | **WATCH window**<br>Displays selected data |

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

## 9.2 Basic Commands for Managing Data

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

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

**whatis** *symbol*

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

| Command | Result displayed in the COMMAND window |
|---|---|
| **`whatis giant`** | `struct zzz giant[100];` |
| **`whatis xxx`** | `struct xxx   {`<br>`   int a;`<br>`   int b;`<br>`   int c;`<br>`   int f1 : 2;`<br>`   int f2 : 4;`<br>`   struct xxx *f3;`<br>`   int f4[10];`<br>`}` |

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

**?** *expression*

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

If the result of *expression* is scalar, then 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 |
|---|---|
| **`? giant`** | `giant[0].b1 436547877`<br>`giant[0].b2 –791051538`<br>`giant[0].b3 1952557575`<br>`giant[0].b4 –1555212096`<br>etc. |
| **`? j`** | `4194425` |
| **`? j=0x5a`** | `90` |

Note that the DISP command (described in detail on page DB:9-20) behaves like the ? command when its *expression* parameter does not identify an aggregate type.

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

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

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

For information about the parallel debug manager version of the EVAL command, see Section 4.9, *Evaluating Expressions*.

## 9.3 Basic Methods for Changing Data Values

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

## *Editing data displayed in a window*

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

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

There are two similar methods for overwriting displayed data:

---

This method is sometimes referred to as the “click-and-type” method.

1) Point to the data item that you want to modify.
2) Click the left button. The debugger highlights the selected field. (Note that the window containing this field becomes active when you press the mouse button.)
3) Type the new information. If you make a mistake or change your mind, press ESC or move the mouse outside the field and press/release the left button; this resets the field to its original value.
4) When you finish typing the new information, press ↵ or any arrow key. This replaces the original value with the new value.

---

1) Select the window that contains the field you’d like to modify; make this the active window. (Use the mouse, the WIN command, or F6. For more detail, see Section 5.4, *The Active Window*.)
2) Use arrow keys to move the cursor to the field you’d like 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’d like to edit is highlighted, press F9. The debugger highlights the field that the cursor is pointing to.
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.

## *Advanced “editing”—using expressions that have side effects*

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

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

For example, if you are using the MP debugger and want to see what’s in register r1, you can enter:

```
? r1 ↵
```

You can also use this type of command to modify r1’s contents. Here are some examples of how you might do this:

| | |
|---|---|
| `? r1++ ↵` | Side effect: increments the contents of r1 by 1 |
| `eval ––r1 ↵` | Side effect: decrements the contents of r1 by 1 |
| `? r1 = 8 ↵` | Side effect: sets r1 to 8 |
| `eval r1/=2 ↵` | Side effect: divides contents of r1 by 2 |

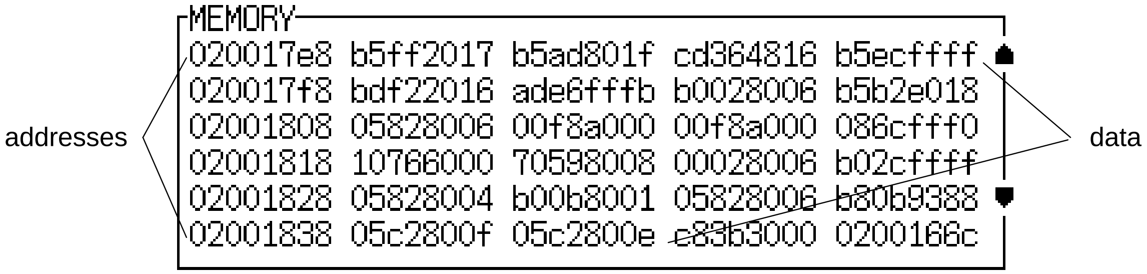
Note that not all expressions have side effects. For example, if you enter ? r1+4, the debugger displays the result of adding 4 to the contents of r1 but does not modify r1’s contents. Expressions that have side effects must contain an assignment operator or an operator that implies an assignment. Operators that can cause a side effect are:

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

These operators are described in Chapter 15, *Basic Information About C Expressions*.

## 9.4 Managing Data in Memory

In mixed and assembly modes, the debugger maintains a MEMORY window that displays the contents of memory. For details concerning the MEMORY window, see the *MEMORY windows* discussion (page DB:5-16).



```
MEMORY
020017e8 b5ff2017 b5ad801f cd364816 b5ecffff
020017f8 bdf22016 ade6fffb b0028006 b5b2e018
02001808 05828006 00f8a000 00f8a000 086cfff0
02001818 10766000 70598008 00028006 b02cffff
02001828 05828004 b00b8001 05828006 b80b9388
02001838 05c2800f 05c2800e c83b3000 0200166c
```

The debugger has commands that show the data values at a specific location or that display a different range of memory in the MEMORY window. The debugger allows you to change the values at individual locations; For more information, see Section 9.3.

## *Displaying memory contents*

The main way to observe memory contents is to view the display in a MEMORY window. Four MEMORY windows are available: the default window is labeled MEMORY, and the three additional windows are called MEMORY1, MEMORY2, and MEMORY3. Notice that the default window does not have an extension number in its name; this is because MEMORY1, MEMORY2, and MEMORY3 are pop-up windows that can be opened and closed throughout your debugging session. Having four windows allows you to view four different memory ranges.

The amount of memory that you can display is limited by the size of the individual MEMORY windows (which is limited only by the screen size). During a debugging session, you may need to display different areas of memory within a window. You can do this by typing a command or using the mouse.

**mem** If you want to display a different memory range in the MEMORY window, use the MEM command. The basic syntax for this command is:

**mem** *expression*

To view different memory locations in an additional MEMORY window, use the MEM command with the appropriate extension number on the end. For example:

| **To do this. . .** | **Enter this. . .** |
|---|---|
| View the block of memory starting at address 0x2004000 in the MEMORY1 window | **`mem1 0x2004000`** |
| View another block of memory starting at address 0x2085000 in the MEMORY2 window | **`mem2 0x2085000`** |

**Note:**

If you want to view a different block of memory explicitly in the default MEMORY window, you can use the aliased command, MEM0. This works *exactly* the same as the MEM command. To use this command, enter:

**mem0** *address*

The *expression* you type in 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. (See *Resizing a window*, page DB:5-27, for more information.)

*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 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:

  ```
  mem1 &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 ip – r1 + label ⏎
  ```

---

You can also change the display of any data-display window—including the MEMORY window—by scrolling through the window's contents. See the *Scrolling through a window's contents* discussion (page DB:5-32) for more details.

## *Displaying memory contents while you're debugging C*

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

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

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

  ```
  ? *0x26 ↵
  ```

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

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

  ```
  wa *0x26 ↵
  ```

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

  ```
  disp *(float *)0x26 ↵
  ```

## *Saving memory values to a file*

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

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

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

For example, to save the values in memory locations 0x0000–0x0040 to a file named memsave, you could enter:

```
ms 0x0,0x40,memsave ↵
```

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

```
load memsave.obj ↵
```

## Filling a block of memory

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

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

❑ The *address* parameter identifies the first address in the block.

❑ The *length* parameter defines the number of words to fill.

❑ The *data* parameter is the value that is placed in each word in the block.

For example, to fill locations 0x0208 5000–0x0208 5300 with the value 0x1234 abcd, you would enter:

```
fill 0x2085000,0x301,0x1234abcd ↵
```

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

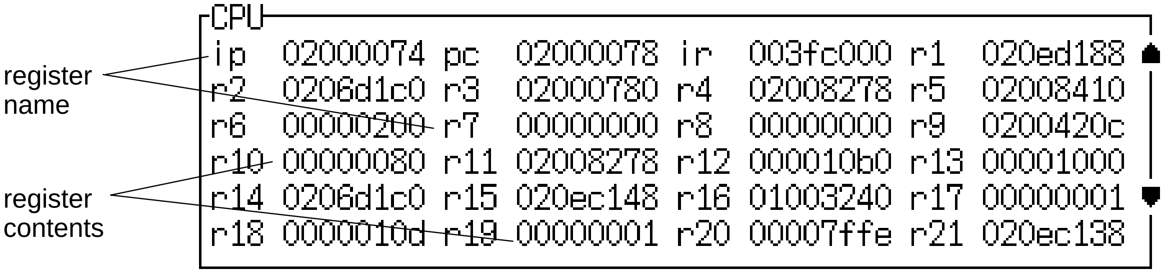
```
mem 0x2085000 ↵
```

This changes the MEMORY window display to show the block of memory beginning at address 0x0208 5000.

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

## 9.5 Managing Register Data

In mixed and assembly modes, the debugger maintains a CPU window that displays the contents of individual registers. For details concerning the CPU window, see the *CPU window* discussion on page DB:5-19.



```
CPU
ip   02000074 pc   02000078 ir   003fc000 r1   020ed188
r2   0206d1c0 r3   02000780 r4   02008278 r5   02008410
r6   00000200 r7   00000000 r8   00000000 r9   0200420c
r10  00000080 r11  02008278 r12  000010b0 r13  00001000
r14  0206d1c0 r15  020ec148 r16  01003240 r17  00000001
r18  0000010d r19  00000001 r20  00007ffe r21  020ec138
```

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 9.3, *Basic Methods for Changing Data Values*.

**Note:**

When you update the MP's IP register, the PC is also modified to the next contiguous instruction address. However, if you modify the PC, the IP is not affected.

Likewise, when you update the PP's IPE register, the IPA and PC registers are also modified to the next contiguous instruction addresses. However, if you modify the IPA or PC, the IPE is not affected.

## *Displaying register contents*

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

- If you have only a temporary interest in the contents of a register, you can use the ? command to display the register's contents. For example, if you are using a PP debugger and want to know the contents of a12, you could enter:

  ```
  ? a12 ↵
  ```

  The debugger displays a12's current contents in the display area of the COMMAND window.

- If you want to observe a register over a longer period of time, you can use the WA command to display the register in a WATCH window. For example, if you want to observe the status register using the MP debugger, you could enter:

  ```
  wa FPST,Status Register ↵
  ```

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

When you're debugging C in auto mode, these methods are also useful because the debugger doesn't show the CPU window in the C-only display.

## *Accessing single-precision floating-point registers*

The MP version of the debugger allows you to display the 31 general-purpose integer register values in single-precision floating-point format. Each register has a pseudoregister name associated with it:

| MP Register Name | Pseudoregister Name |
|---|---|
| r2 | f2 |
| r3 | f3 |
| r4 | f4 |
| . | . |
| . | . |
| r31 | f31 |

- For example, assume r2 = 0x4140 0000. If you want to evaluate the f2 register, enter:

  ```
  ?f2 ↵
  ```

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

  ```
  ?f2  1.2000000+01
  ```

- To modify the f2 register and set it equal to 15.75, enter:

  ```
  ?f2 = 15.75 ↵
  ```

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

  ```
  ?f2 = 15.75 1.575000e+01
  ```

Similarly, the PP version of the debugger allows you to display the eight data register values in single-precision floating-point format. Each register has a pseudoregister name associated with it:

| PP Register Name | Pseudoregister Name |
|---|---|
| d1 | f1 |
| d2 | f2 |
| d3 | f3 |
| . | . |
| . | . |
| d7 | f7 |

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

The MP version of the debugger allows you to access double-precision floating-point values in even/odd register pairs. There are 15 sets of MP general-purpose register pairs. Each pair has a pseudoregister name associated with it:

| **MP Register Pair** | **Pseudoregister Name** | **MP Register Pair** | **Pseudoregister Name** |
|---|---|---|---|
| r2/r3 | df2 | r18/r19 | df18 |
| r4/r5 | df4 | r20/r21 | df20 |
| r6/r7 | df6 | r22/r23 | df22 |
| r8/r9 | df8 | r24/r25 | df24 |
| r10/r11 | df10 | r26/r27 | df26 |
| r12/r13 | df12 | r28/r29 | df28 |
| r14/r15 | df14 | r30/r31 | df30 |
| r16/r17 | df16 | | |

□ For example, if you want to evaluate the df10 register pair, enter:

```
?df10 ↵
```

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

```
?df10 1.1234500e+00
```

□ To modify the df10 register pair and set them equal to 25.75, enter:

```
?df10 = 25.75 ↵
```

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

```
?df10 = 25.75 2.575000e+01
```

## *Accessing floating-point accumulators*

The MP version of the debugger allows you to display the MP’s FPU accumulator (a0–a3) values in floating-point format. Each accumulator has two pseudoregister names associated with it:

| MP Register Name | Pseudoregister Pair |
|---|---|
| a0 | a0h/a0l |
| a1 | a1h/a1l |
| a2 | a2h/a2l |
| a3 | a1h/a3l |

To display an accumulator in 64-bit hexadecimal format, use the pseudoregister pair to show the 32-bit high and 32-bit low values.

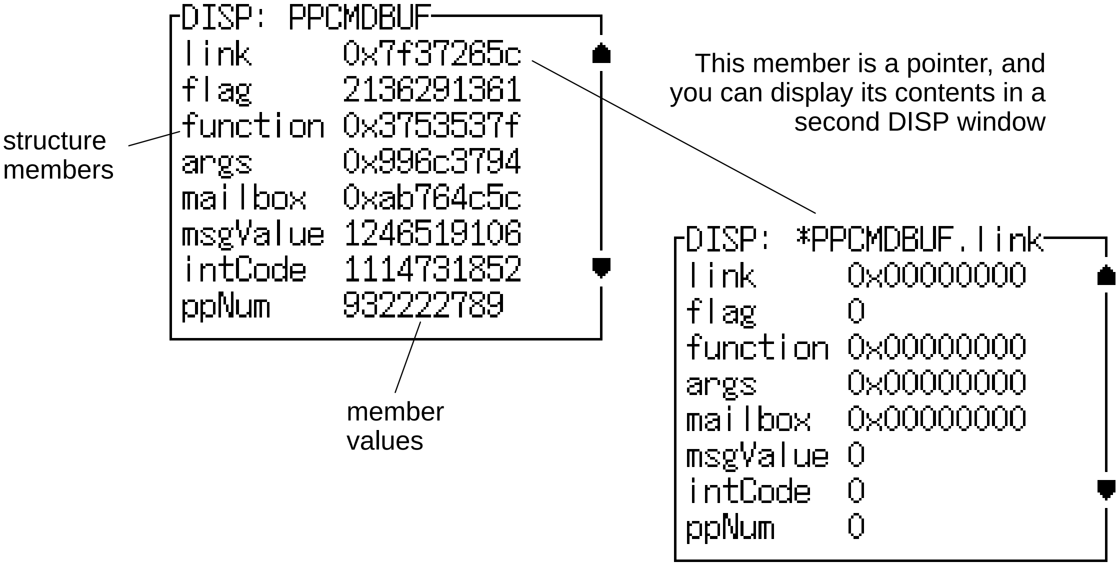
## *Additional MVP pseudoregisters*

**simulator only**

- ❑ You can use the halt override register (HALTOVR) to disable the PP’s processor halt bit. This allows you to execute PP code without running any MP code to unhalt the PP. This register exists in each of the PPs.
- ❑ When the LSB of the HALTMODE register is 1, all processors are forced to the halt state if one processor encounters a software breakpoint. This register is common to the MP and the PPs.

## 9.6 Managing Data in a DISP (Display) Window

The main purpose of the DISP window is to display members of complex, aggregate data types such as arrays and structures. The debugger shows DISP windows **only when you specifically request to see DISP windows** with the DISP command (described below). Note that you can have up to 120 DISP windows open at once. For additional details about DISP windows, see the *DISP windows* discussion (page DB:5-21).



```
DISP: PPCMDBUF
link      0x7f37265c
flag      2136291361
function  0x3753537f
args      0x996c3794
mailbox   0xab764c5c
msgValue  1246519106
intCode   1114731852
ppNum     932222789
```

```
DISP: *PPCMDBUF.link
link      0x00000000
flag      0
function  0x00000000
args      0x00000000
mailbox   0x00000000
msgValue  0
intCode   0
ppNum     0
```

structure members
member values
This member is a pointer, and you can display its contents in a second DISP window

Remember, you can use the data-management commands or the debugger’s overwrite editing capability to modify the contents of any value displayed in a DISP window. For more information, see Section 9.3, *Basic Methods for Changing Data Values*.

## *Displaying data in a DISP window*

**disp** To open a DISP window, use the DISP command. Its basic syntax is:

**disp** *expression*

If the *expression* is not an array, structure, or pointer (of the form **pointer name*), the DISP command behaves like the ? command. However, if *expression* **is** one of these types, the debugger opens a DISP window to display the values of the members.

If a DISP window contains a long list of members, you can use PAGE DOWN, PAGE UP, or arrow keys to scroll through the window. If the window contains an array of structures, you can use CONTROL PAGE DOWN and CONTROL PAGE UP to scroll through the array.

Once you open a DISP window, you may find that a displayed member is another one of these types. This is how you identify the members that are arrays, structures, or pointers:

| | |
|---|---|
| A member that is an array looks like this | [. . .] |
| A member that is a structure looks like this | {. . .} |
| A member that is a pointer looks like an address | 0x00000000 |

You can display the additional data (the data pointed to or the members of the array or structure) in additional DISP windows (these are referred to as *children*). You can do this with function keys, with the mouse, or by typing.

Use the DISP command again; this time, *expression* must identify the member that has additional data. For example, if the first expression identifies a structure named str and one of str's members is an array named f4, you can display the contents of the array by entering this command:

```
disp str.f4 ↵
```

This opens a new DISP window that shows the contents of the array. If str has a member named f3 that is a pointer, you could enter:

```
disp *str.f3 ↵
```

This opens a window to display what str.f3 points to.

Here’s another method of displaying the additional data:

↖ 1) Point to the member in the DISP window.

2) Now click the left button.

Here’s the third method:

↑ ↓ 1) Use the arrow keys to move the cursor up and down in the list of members.

F9 2) When the cursor is on the desired field, press F9.

When the debugger opens a second DISP window, the new window may at first be displayed on top of the original DISP window—if so, you can move the windows so that you can see both at once. If the new windows also have members that are pointers or aggregate types, you can continue to open new DISP windows.

## *Closing a DISP window*

Closing a DISP window is a simple, two-step process.

**Step 1:** Make the DISP window that you want to close active (see Section 5.4, *The Active Window*).

**Step 2:** Press F4.

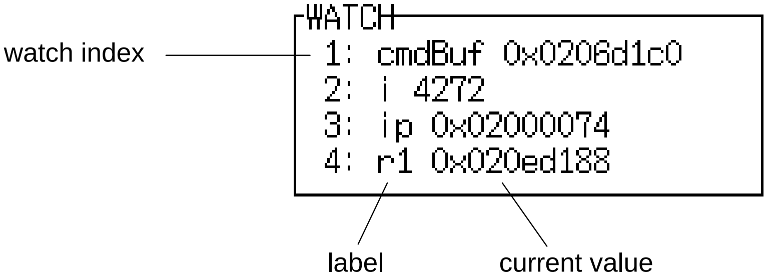
Note that you can close a window and all of its children by closing the original window.

**Note:**

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

## 9.7 Managing Data in a WATCH Window

The debugger doesn't maintain a dedicated window that tells you about the status of all the symbols defined in your program. Such a window might be so large that it wouldn't be useful. Instead, the debugger allows you to create a WATCH window that shows you how program execution affects specific expressions, variables, registers, or memory locations.



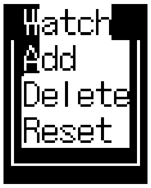
```
WATCH
 1: cmdBuf 0x0206d1c0
 2: i 4272
 3: ip 0x02000074
 4: r1 0x020ed188
```

The debugger displays a WATCH window *only when you specifically request a WATCH window* with the WA command (described below). Note that there is only one WATCH window. For additional details concerning the WATCH window, see the *WATCH window* discussion (page DB:5-22).

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 9.3, *Basic Methods for Changing Data Values*.

**Note:**

All of the watch commands described here can also be accessed from the Watch pulldown menu. For more information about using the pulldown menus, see Section 6.2, *Using the Menu Bar and the Pulldown Menus*.



## Displaying data in the WATCH window

The debugger has one command for adding items to the WATCH window.

**wa** To open the WATCH window, use the WA (watch add) command. The basic syntax is:

**wa** *expression* [*, label*]

When you first execute WA, the debugger opens the WATCH window. After that, executing WA adds additional values to the WATCH window.

- The *expression* parameter can be any C expression, including an expression that has side effects. It's most useful to watch an expression whose value will change over time; constant expressions provide no useful function in the watch window.
- If you want to use the **contents** of an address as a parameter, be sure to prefix the address with the C indirection operator (*****). Use the WA command to do this:

  ```
  wa *0x26 ⏎
  ```

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

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

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

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

**wr**

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

**wd** *index number*

Whenever you add an item to the WATCH window, the debugger assigns it an index number. (The illustration of the WATCH window on page DB:9-22 points to these watch indexes.) The WD command's *index number* parameter must correspond to one of the watch indexes in the WATCH window.

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

**Note:**

The debugger automatically closes the WATCH window when you execute a LOAD or SLOAD command.

## 9.8 Displaying Data in Alternative Formats

By default, all data is displayed in its natural format. This means that:

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

However, any data displayed in the COMMAND, MEMORY, WATCH, or DISP window can be displayed in a variety of formats.

### *Changing the default format for specific data types*

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

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

The *display format* parameter identifies the new display format for any data of type *data type*. Table 9–1 lists the available formats and the corresponding characters that can be used as the *display format* parameter.

Table 9–1. Display Formats for Debugger Data

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

Table 9–2 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 9–2 also shows valid combinations of data types and display formats.

Table 9–2. Data Types for Displaying Debugger Data

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

Here are some examples:

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

**`setf short, u`** ↵

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

**`setf short, *`** ↵

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

**`setf`** ↵

You'll see a display that looks something like this:

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

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

**`setf *`** ↵

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

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

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

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

Here are some examples:

- To watch the PC register in decimal, enter:

  ```
  wa pc,,d ↵
  ```

- To display memory contents in octal, enter:

  ```
  mem 0x0,o ↵
  ```

- To display an array of integers as characters, enter:

  ```
  disp ai,c ↵
  ```

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

Chapter 10

# Using Software Breakpoints

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

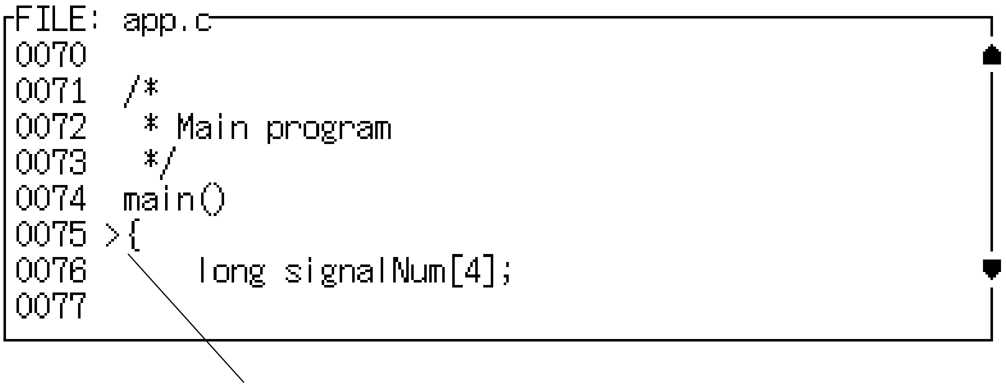
Software breakpoints are especially useful in combination with conditional execution (described on page DB:8-18) and benchmarking (described on page DB:8-20).

**Topics**

## 10.1 Setting a Software Breakpoint

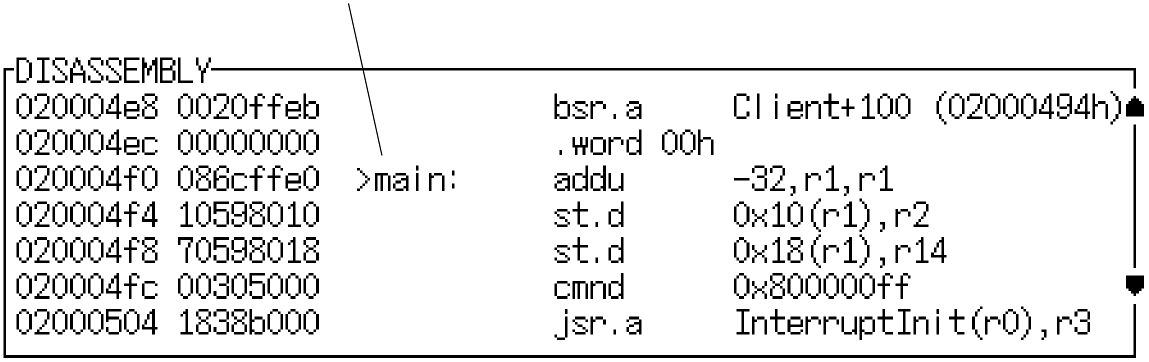
When you set a software breakpoint, the debugger highlights the breakpointed line by prefixing the statement with the character >.

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



```
FILE: app.c
0070
0071  /*
0072   * Main program
0073   */
0074  main()
0075 >{
0076      long signalNum[4];
0077
```

A breakpoint is set at this C statement; notice how the line is highlighted.
A breakpoint is also set at the associated assembly language statement (it’s highlighted, too).



```
DISASSEMBLY
020004e8 0020ffeb               bsr.a     Client+100 (02000494h)
020004ec 00000000               .word 00h
020004f0 086cffe0  >main:       addu      -32,r1,r1
020004f4 10598010               st.d      0x10(r1),r2
020004f8 70598018               st.d      0x18(r1),r14
020004fc 00305000               cmnd      0x800000ff
02000504 1838b000               jsr.a     InterruptInit(r0),r3
```

When you are debugging C code, you cannot set a breakpoint at a function name; you can set a breakpoint only at the beginning of the *code* within a function or at the “{” symbol. For example, in the FILE window above, the breakpoint is set at line 75 instead of line 74 to indicate a breakpoint for the main( ) function.

When you set a breakpoint, the opcode in memory is replaced with a special software breakpoint instruction. If the address of the instruction is within the instruction or data cache, the software breakpoint is also written to those locations.

When the debugger encounters a software breakpoint, it halts execution. When you resume execution by using a run or single-step command, the original opcode is loaded back into the instruction cache only, and the instruction is stepped (with interrupts disabled). The breakpoint opcode is then placed back in the instruction cache, and execution is resumed.

If you are running common PP code and you want all of the PPs to execute a certain breakpoint, you must set the breakpoint for each individual processor. If you don't set the breakpoint for a processor, that processor will step over the breakpoint and resume execution.

**Notes:**

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

2) Up to 200 breakpoints can be set.

3) If you set a hardware and software breakpoint at the same location, the emulator cannot advance. Every time you enter a RUN or STEP command from this condition, the hardware and software breakpoints toggle, which prevents the pipeline from advancing. To proceed, you must either delete the software breakpoint or disable the hardware breakpoint.

There are several ways to set a software breakpoint:

1) Point to the line of assembly language code or C code where you'd like to set a breakpoint.

2) Click the left button.

Repeating this action clears the breakpoint.

1) Make the FILE or DISASSEMBLY window the active window.

2) Use the arrow keys to move the cursor to the line of code where you'd like to set a breakpoint.

3) Press the F9 key.

Repeating this action clears the breakpoint.

**ba** If you know the address where you'd like to set a software breakpoint, you can use the BA (breakpoint add) command. This command is useful because it doesn't require you to search through code to find the desired line. The syntax for the BA command is:

**ba** *address*

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

## 10.2 Clearing a Software Breakpoint

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

---

↖ 1) Point to a breakpointed assembly language or C statement.

2) Click the left button.

---

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

F9 2) Press the F9 key.

---

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

**br**

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

**bd** *address*

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

## 10.3 Finding the Software Breakpoints That Are Set

**bl** Sometimes you may need to know where software breakpoints are set. For example, the BD command's *address* parameter must correspond to the address of a breakpoint that is set. The BL (breakpoint list) command provides an easy way to get a complete listing of all the 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 breakpoints that are set, in the order in which you set them. Here's an example of this type of list:

```
 Address       Symbolic Information

02000040    in main, at line 56, ”/mvphll/sample.c”
020000b0
```

The address is the memory address of the breakpoint. The symbolic information identifies the function, line number, and filename of the breakpointed C statement:

- ❑ If the breakpoint was set in assembly language code, you'll see only an address unless the statement defines a symbol.
- ❑ If the breakpoint was set in C code, you'll see the address together with symbolic information.

Chapter 11

# Using the Simulator Graphics Display Features

The MVP simulator has features for displaying two types of information. You can display the images created by your code. You can also display the memory access patterns and status information of the MVP processors on a cycle-by-cycle basis.

This chapter tells you how to use these two graphics display features:

❑ The **image display** shows a grayscale or color representation of an image stored in memory.

❑ The **data-flow plot** provides a cycle-by-cycle visual representation of processor status and memory accesses.

These displays are shown separately from the rest of the debugger interface. Each is displayed in its own X window instead of the types of windows used within the debugger. These features are available only when you are using the simulator and a color monitor.

**Topics**

## 11.1 Displaying Images

The image display feature shows an image stored in MVP memory. There are four types of images that you can display:

- **8-bit grayscale.** This image type allows for 256 shades of gray. The 8-bit grayscale pixels are packed, 8-bit bytes and are displayed as 256 shades of gray on either an 8-bit or 24-bit color display.
- **16-bit color.** This is a color image that uses 16-bit pixels that have the following bit configuration:

  MSB LSB

  | | | | |
  |---|---|---|---|
  | 1 | 5 | 5 | 5 |
  | x | Blue | Green | Red |

  Two 16-bit pixels are packed into a 32-bit word.

  - When a 16-bit color image is displayed on an 8-bit color display, the color pixels are compressed into an 8-bit approximation that uses six shades each of red, green, and blue that make a 6 × 6 × 6 color cube.
  - When a 16-bit color image is displayed on a 24-bit color display, the 16-bit pixels are expanded to 24 bits by shifting each 5-bit field to the left by three bits.
- **32-bit color.** This is a color image that uses 24-bit pixels embedded into a 32-bit word, as shown:

  MSB LSB

  | | | | |
  |---|---|---|---|
  | 8 | 8 | 8 | 8 |
  | x x x x x x x x | Blue | Green | Red |

  - When a 32-bit color image is displayed on an 8-bit color display, the color pixels are compressed into an 8-bit approximation, as described for the 16-bit color image above.
  - When a 32-bit color image is displayed on a 24-bit color display, the 32-bit pixels will be displayed in true color.
- **YCbCr 4:1:1.** This color arrangement is commonly used in JPEG, MPEG, and P × 64 standards.

  *YCbCr* refers to the CCIR601 color space to which the pixels correspond. Instead of the familiar red, green, and blue, each pixel is composed of the following:

  - Eight bits of **luminance** (the Y field) that indicates the pixel intensity or brightness
  - Eight bits each of two **chrominance** components, Cb and Cr, that indicate color

*4:1:1* describes the subsampling arrangement of the pixels. For each 2 × 2 block of luminance (Y), there is only one corresponding Cb and Cr. As a result, the single Cb and Cr values are used with all four corresponding Y values, and the YCbCr ratio is 4:1:1.

## *Displaying an image*

To display an image, use the IMAGE command. The syntax for the IMAGE command is:

**image** *addr* [**,** *width* [**,** *height* [**,** *color* [**,** *Cb addr* [**,** *Cr addr* [**,** *frequency*]]]]]]

If you enter this command without any parameters, the debugger displays a dialog box for you to enter the parameter values.

- ❑ The *addr* parameter defines the starting address byte of the image in memory. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label. The address that you supply must be 8-byte aligned.

  If you plan to display a YCbCr 4:1:1 image, the *addr* that you enter is the starting address for the Y value.

- ❑ The *width* parameter defines the width, in pixels, of the image. The default width is 256 bytes.

  If you plan to display a YCbCr 4:1:1 image, the *width* that you enter is the width of the Y value. The Cb and Cr width values are half the *width* value that you enter. Note that the value that you enter for the image width must be a multiple of 8, or the image will look distorted.

- ❑ The *height* parameter defines the height, in pixels, of the image. The default height is 256 bytes.

  If you plan to display a YCbCr 4:1:1 image, the *height*that you enter is the height of the Y value. The Cb and Cr height values are half the *height* value that you enter.

- ❑ The *color* parameter defines the type of image that you want to display.

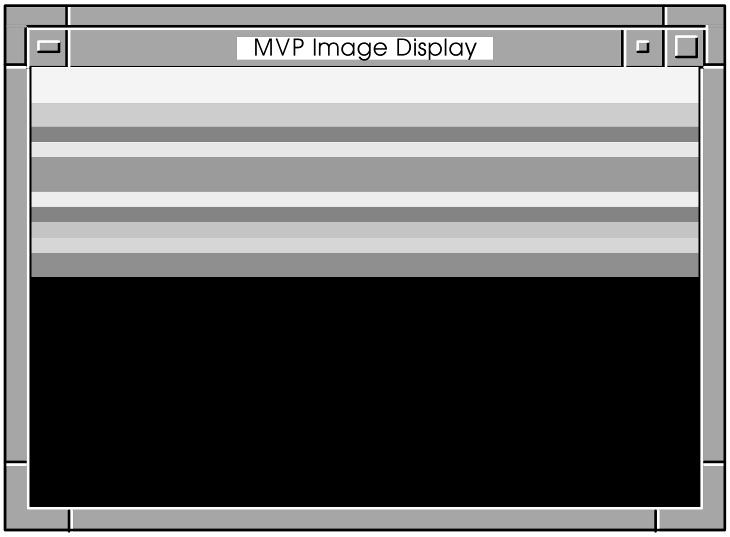
| To display this type of image. . . | Enter this color type |
|---|---|
| 8-bit grayscale | **8** or **8_bit** |
| 16-bit color | **16** or **16_bit** |
| 32-bit color | **32** or **32_bit** |
| YCbCr 4:1:1 | **ycbcr** or **ycbcr4:1:1** |

Note that the color type names are not case sensitive.

- ❑ The *Cb addr* and *Cr addr* parameters are necessary if you select YCbCr 4:1:1 as your color type. The *Cb addr* parameter defines the Cb starting address, and the *Cr addr* parameter defines the Cr starting address.
- ❑ The *frequency* parameter determines how frequently the image is read from memory and redisplayed. For example, if you specify an update rate of 200 and you begin displaying the image at the 50th cycle, the image will be updated at 250 cycles, 450 cycles, etc. The default update frequency is 500 cycles.

Figure 11–1 shows an example of an image display; the image is displayed in an X window instead of in a debugger window. Note that one or more processors must be running code for you to see any activity in the image window.

Figure 11–1. A Displayed Image



**Notes:** 1) Although this example is shown in black and white, the image display feature is designed for use on a color monitor.

2) Your window label and borders may look different from the illustration, depending on which window manager you're using under the X Window System.

## *Manipulating the image*

Once an image is displayed, you can zoom it, change the background color, or request information about specific pixels. If you want these things to happen immediately, you must enter interactive mode; if you don't enter interactive mode, the task you request will not take place until the next time the image is updated.

□ **Enter interactive mode.** To enter interactive mode, press any key but Q, R, or B. The image will be suspended, and any actions that you request will occur immediately. If you are displaying multiple images, entering interactive mode in one window will put you in interactive mode in all of the image displays.

The shape of the cursor tells you whether you're in interactive mode:

| **Interactive mode?** | **Cursor shape** |
|---|---|
| No | arrow : ↖ |
| Yes | crosshair: + |

□ **Change the background color.** Press B (uppercase or lowercase) to toggle the window background between black and white. The background color will change when the image is updated or when you zoom the image. Black is the default background color.

□ **Display pixel coordinates.** If you point to a pixel and press the left mouse button, the window displays the XY coordinates (relative to the upper-left pixel at 0,0) and value of the pixel. You can move the mouse while you're holding the button; you'll see information for each pixel that the mouse cursor passes over.

□ **Zoom the image**. Press the middle mouse button to make the image twice as big. The image will be cropped to fit within the existing window boundaries and shifted so that the center of the new image is at the position where the mouse cursor was when you pressed the button. You can continue zooming until only a single pixel is visible in the window.

□ **Unzoom the image.** Press the right mouse button to unzoom one level (making the image half as large). You cannot make the image smaller than its original size.

- **Exit interactive mode.** Press R (uppercase or lowercase) to stop interactive mode and resume the simulation. (Note that pressing R is meaningless if you are not in interactive mode.) If you are displaying multiple images, pressing R in any image display resumes simulation for all the images.

- **Quit the image.** Press Q (uppercase or lowercase) to quit displaying the image and close the associated window. You can also quit the image by moving your mouse cursor to the border of the window that contains the image and executing Quit or Close from the window-manager menu.

  Note that the image display window will not disappear until you enter another command from the debugger.

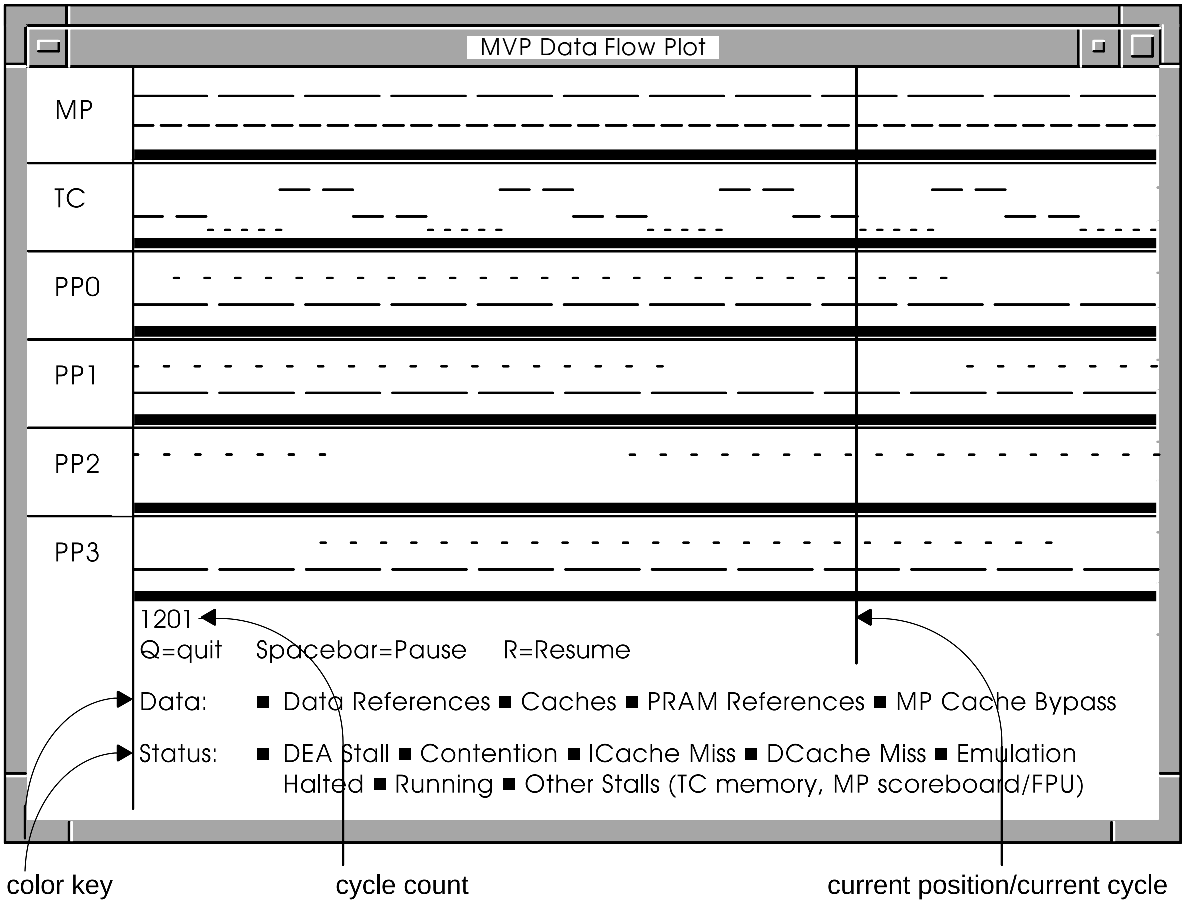
## 11.2 Displaying Processor Status and Data-Flow Information

The data-flow plot provides a cycle-by-cycle display of processor status and memory accesses. To display the data-flow plot, enter:

```
dataplot ↵
```

Figure 11–2 shows an example of a data-flow plot. Note that one or more processors must be running code for you to see any activity in the data-flow plot window.

Figure 11–2. Data-Flow Plot Window



1201
Q=quit Spacebar=Pause R=Resume

Data: ■ Data References ■ Caches ■ PRAM References ■ MP Cache Bypass

Status: ■ DEA Stall ■ Contention ■ ICache Miss ■ DCache Miss ■ Emulation Halted ■ Running ■ Other Stalls (TC memory, MP scoreboard/FPU)

**Notes:** 1) Although this example is shown in black and white, the data-flow plot is designed for use on a color monitor.

2) Your window label and borders may look different from the illustration, depending on which window manager you're using under the X Window System.

The status and memory-access activities are represented by horizontal lines drawn across the data-flow plot; the lines progress with each cycle. When the lines reach the right edge of the window, the plot begins redrawing from the left. (You can also resize the window to show more cycles.) A vertical line moves across the plot to identify the current cycle. A number at the bottom of the window shows the cycle count.

The plot is divided into six areas—one for the MP, one for the TC, and one for each PP. The heavy horizontal line at the bottom of each plot area shows the processor status; the other horizontal lines in each plot area show various memory accesses. This information is color keyed, as described at the bottom of the data-flow plot window.

Each area in the plot is divided into multiple lines, as described in Table 11–1. Each line corresponds to one of the 2K-byte crossbar RAMs; line 0 corresponds to the RAM at crossbar addresses 0x0000–0x07FF, line 1 corresponds to the RAM at crossbar addresses 0x0800–0x0FFF, etc.

Table 11–1. Plot Lines by Area

| Area | Row | Description |
|---|---|---|
| MP | 0–11 | Data RAM references (PP0 data RAM0, PP0 data RAM1, PP0 data RAM2,...PP3 data RAM2) |
| | 12 | MP data cache |
| | 13 | MP cache bypass (see Note 2) |
| | 14 | MP parameter RAM reference |
| | 15–18 | PP parameter RAM references (16=PP0...19=PP3) |
| | 19–22 | PP cache references (20=PP0...23=PP3) |
| | 23 | Run status |
| TC | 0–11 | Data RAM references (PP0 data RAM0, PP0 data RAM1, PP0 data RAM2,...PP3 data RAM2) |
| | 12 | TC parameter RAM reference |
| | 13 | TC cache reference |
| | 14 | Run status |
| PPs | 0–11 | Data RAM references (PP0 data RAM0, PP0 data RAM1, PP0 data RAM2,...PP3 data RAM2) |
| | 12 | Instruction cache |
| | 13–16 | Parameter RAM references |
| | 17 | Run status |

**Notes:** 1) Row 0 is the first row of a plot area.

2) *Cache* refers to a data access at a cached address; *cache bypass* refers to a data access at the actual address of the MP data cache in on-chip memory.

## *Pausing, resuming, or quitting the plot*

key

- ❑ To pause the plot, press SPACE or P.

  Pausing the plot is useful for obtaining a snapshot of processor activity. While the plot is paused, you can find the cycle number associated with any position on the plot by pointing to that position and pressing the left mouse button. When you do this, you'll see a message that shows the cycle number.

- ❑ To resume the plot, press R.

- ❑ To quit plotting and close the window, press Q. You can also quit plotting by moving your mouse cursor to the border of the window that contains the data-flow plot and executing Quit or Close from the window-manager menu.

  Note that the data-flow plot window will not disappear until you enter another command from the debugger.

Chapter 12

# Customizing the Debugger Display

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

**Topics**

## 12.1 Changing the Colors of the Debugger Display

The MVP MP and PP debuggers work best on a color monitor; 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*, $attr_1$ [, $attr_2$ [, $attr_3$ [, $attr_4$] ] ]
**scolor** *area name*, $attr_1$ [, $attr_2$ [, $attr_3$ [, $attr_4$] ] ]

These commands are similar. However, SCOLOR updates the screen immediately, and COLOR doesn't update the screen (the new colors/attributes take effect as soon as the debugger executes another command that updates the screen). Typically, you might use the COLOR command several times, followed by an SCOLOR command to put all of the changes into effect at once.

The *area name* parameter identifies the areas of the display that are affected. The *attributes* identify how the areas are affected. Table 12–1 lists the valid values for the *attribute* parameters.

Table 12–1. Colors and Other Attributes for the COLOR and SCOLOR Commands

(a) Colors

| | | | |
|---|---|---|---|
| black | blue | green | cyan |
| red | magenta | yellow | white |

(b) Other attributes

| | |
|---|---|
| bright | blink |

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

Table 12–2 lists valid values for the *area name* parameters. This is a long list; the subsections following the table further identify these areas.

Table 12–2. Summary of Area Names for the COLOR and SCOLOR Commands

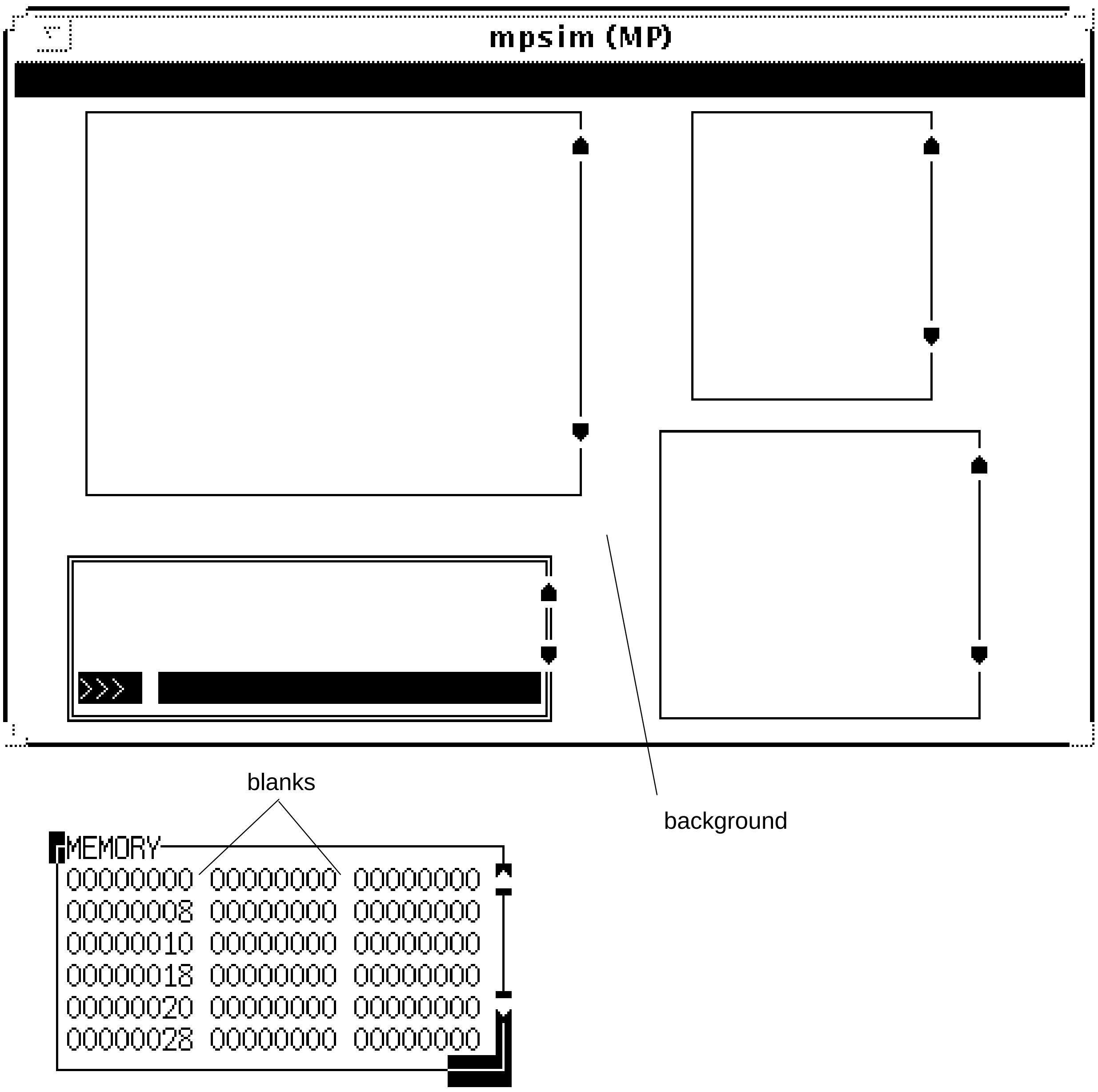
| | | | |
|---|---|---|---|
| menu_bar | menu_border | menu_entry | menu_cmd |
| menu_hilite | menu_hicmd | win_border | win_hiborder |
| win_resize | field_text | field_hilite | field_edit |
| field_label | field_error | cmd_prompt | cmd_input |
| cmd_cursor | cmd_echo | asm_data | asm_cdata |
| asm_label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

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

You don’t have to type an entire *attribute* or *area name*; you need to type only enough letters to uniquely identify either parameter. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they’re listed in Table 12–2 (left to right, top to bottom).

The remainder of this section identifies the areas.

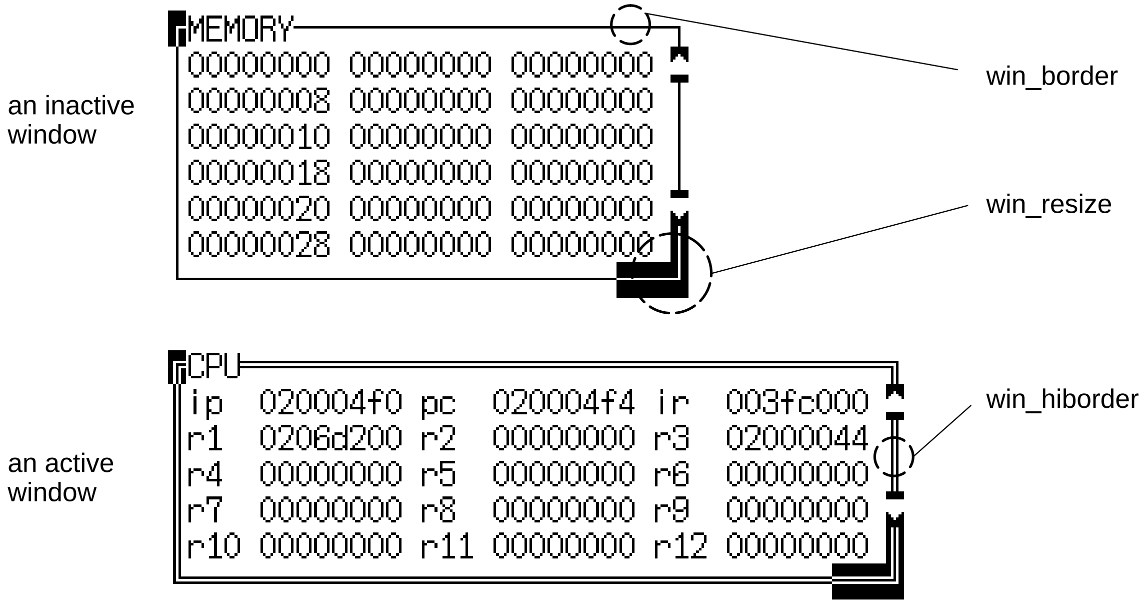
## *Area names: common display areas*



```
MEMORY
00000000 00000000 00000000
00000008 00000000 00000000
00000010 00000000 00000000
00000018 00000000 00000000
00000020 00000000 00000000
00000028 00000000 00000000
```

| **Area identification** | **Parameter name** |
|---|---|
| Screen background (behind all windows) | background |
| Window background (inside windows) | blanks |

## *Area names: window borders*

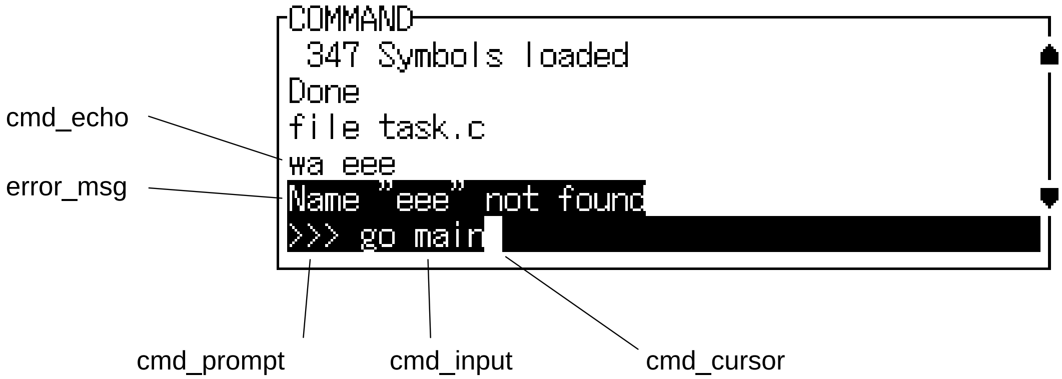


```
MEMORY
00000000 00000000 00000000
00000008 00000000 00000000
00000010 00000000 00000000
00000018 00000000 00000000
00000020 00000000 00000000
00000028 00000000 00000000
```

```
CPU
ip  020004f0 pc  020004f4 ir  003fc000
r1  0206d200 r2  00000000 r3  02000044
r4  00000000 r5  00000000 r6  00000000
r7  00000000 r8  00000000 r9  00000000
r10 00000000 r11 00000000 r12 00000000
```

| Area identification | Parameter name |
|---|---|
| Window border for any window that isn’t active | win_border |
| The reversed “L” in the lower right corner of a resizable window | win_resize |
| Window border of the active window | win_hiborder |

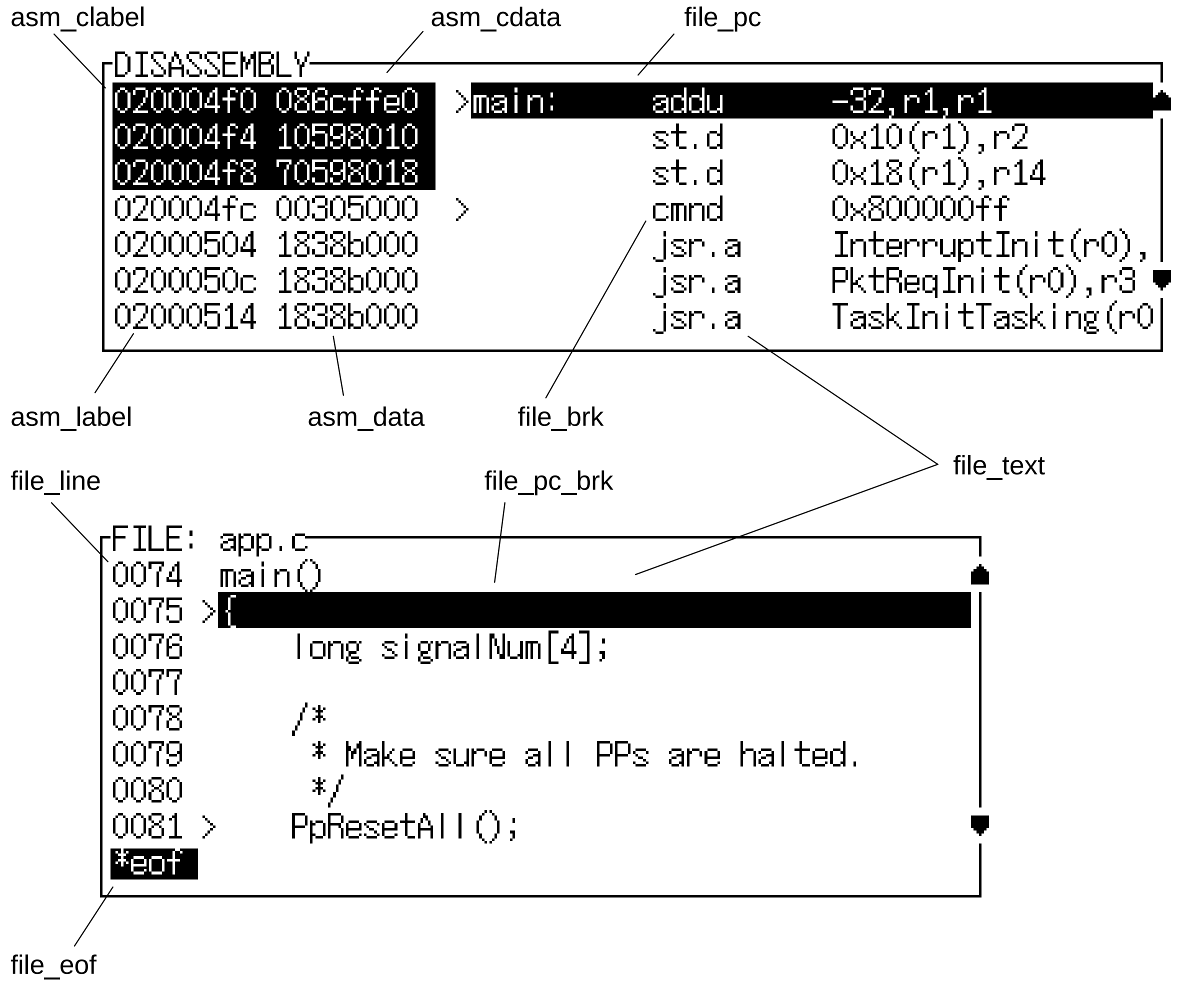
## *Area names: COMMAND window*



```
COMMAND
 347 Symbols loaded
Done
file task.c
wa eee
Name ”eee” not found
>>> go main
```

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

## Area names: DISASSEMBLY and FILE windows

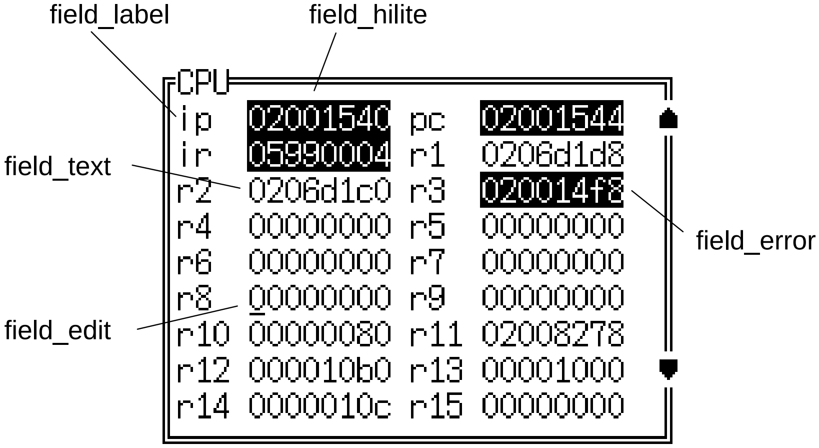


```
DISASSEMBLY
020004f0 086cffe0  >main:     addu     -32,r1,r1
020004f4 10598010             st.d     0x10(r1),r2
020004f8 70598018             st.d     0x18(r1),r14
020004fc 00305000  >          cmnd     0x800000ff
02000504 1838b000             jsr.a    InterruptInit(r0),
0200050c 1838b000             jsr.a    PktReqInit(r0),r3
02000514 1838b000             jsr.a    TaskInitTasking(r0
```

```
FILE: app.c
0074  main()
0075 >{
0076      long signalNum[4];
0077
0078      /*
0079       * Make sure all PPs are halted.
0080       */
0081 >    PpResetAll();
*eof
```

| Area identification | Parameter name |
|---|---|
| Object code in DISASSEMBLY window that is associated with current C statement | asm_cdata |
| Object code in DISASSEMBLY window | asm_data |
| Addresses in DISASSEMBLY window | asm_label |
| Addresses in DISASSEMBLY window that are associated with current C statement | asm_clabel |
| Line numbers in FILE window | file_line |
| End-of-file marker in FILE window | file_eof |
| Text in FILE or DISASSEMBLY window | file_text |
| Breakpointed text in FILE or DISASSEMBLY window | file_brk |
| Current PC in FILE or DISASSEMBLY window | file_pc |
| Breakpoint at current PC in FILE or DISASSEMBLY window | file_pc_brk |

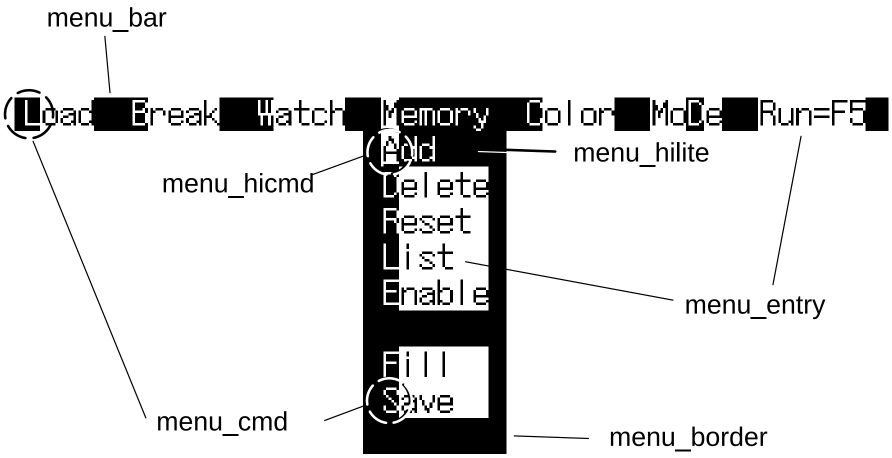
## Area names: data-display windows



```
CPU
ip  02001540 pc  02001544
ir  05990004 r1  0206d1d8
r2  0206d1c0 r3  020014f8
r4  00000000 r5  00000000
r6  00000000 r7  00000000
r8  00000000 r9  00000000
r10 00000080 r11 02008278
r12 000010b0 r13 00001000
r14 0000010c r15 00000000
```

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

## *Area names: menu bar and pulldown menus*



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

## 12.2 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 can change 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/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 |

Here are some examples of the BORDER command. Note that you can skip parameters, if desired.

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

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

## 12.3 Saving and Using Custom Displays

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

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

The debugger supports two commands for saving and restoring custom screen configurations into files. The filenames that you use for restoring configurations must correspond to the filenames that you used for saving configurations. Note that these are binary files, not text files, so you can't edit the files with a text editor.

### *Saving a custom display*

**ssave** Once you've customized the debugger display to your liking, you can use the SSAVE command to save the current screen configuration to a file. The format for this command is:

**ssave** [*filename*]

This saves the screen resolution, border styles, colors, window positions, and window sizes for all debugging modes.

The *filename* parameter names the new screen configuration file. You can include path information (including relative pathnames); if you don't specify path information, the debugger places the file in the current directory. If you don't supply a filename, the debugger saves the current configuration into a file named init.clr.

Note that you can execute this command as the Save selection on the Color pulldown menu.

## Loading a custom display

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

**sconfig** [*filename*]

This restores the screen colors, window positions, window sizes, and border styles for all debugging modes. The *filename* parameter names the new screen configuration file. You can include path information (including relative pathnames); if you don't supply path information, the debugger looks for the file in the current directory. Note that you must save screen configuration files with the SSAVE command in order to use them with the SCONFIG command.

If you don't supply a *filename*, then the debugger looks for init.clr. The debugger searches for the file in the current directory and then in directories named with the D_DIR environment variable.

Note that you can execute this command as the Load selection on the Color pulldown menu.

## Invoking the debugger with a custom display

If you set up the screen in a way that you like and always want to invoke the debugger with this screen configuration, you have two choices for accomplishing this:

❑ Save the configuration in init.clr.

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

## Returning to the default display

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

## 12.4 Changing the Prompt

Whenever you invoke a debugger with a processor name, that processor name is used as the default command-line prompt for the debugger.

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

**prompt** *new prompt*

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

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

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

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

**Note:**

Whenever you select a default group of processors, the group name becomes the command-line prompt for the parallel debug manager. You **cannot** use the PROMPT command to change the parallel debug manager's command-line prompt. To change the parallel debug manager prompt, use the SET command (see page DB:4-28).

Chapter 13

# Using the Analysis Interface

The MVP master processor (MP) and parallel processor (PP) each have an analysis module that allows the emulator to monitor bus addresses and events generated by other processors. The MP and PP debuggers provide you with easy-to-use windows, dialog boxes, and analysis commands that let you set hardware breakpoints on the occurrence of an analysis event.

The debuggers access each analysis module through a special set of pseudoregisters. The dialog boxes described in this chapter provide a transparent means of loading these registers. You will, in most cases, access the analysis features, unlike many of the other debugger features, through dialog boxes rather than through commands. If the dialog boxes do not meet your needs, you can use the special set of aliased commands that deal directly with the analysis pseudoregisters. These commands are described in Appendix C, *Customizing the Analysis Interface*.

## Topics

## 13.1 Introducing the Analysis Interface

The MVP analysis interface provides a detailed look into events occurring in hardware, expanding your debugging capabilities beyond software breakpoints. The analysis interface examines MVP memory-bus cycle information in real time and reacts to this information through hardware breakpoints.

The analysis interface allows you to:

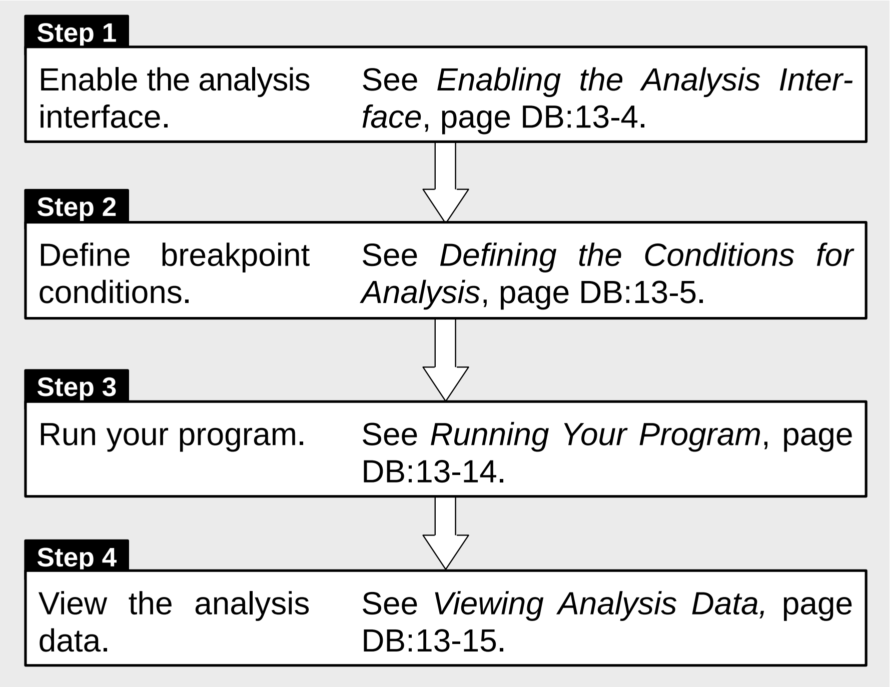
❑ **Set up break events.** The analysis interface allows you to halt the processor during execution of your program by setting hardware breakpoints. The events that cause the processor to stop are called **break events**. A break event can be defined for processor data or instruction bus accesses for the MP and processor global, local or instruction bus accesses for the PP.

  Hardware break events also allow you to set breakpoints in ROM. This enables you to break on events that you cannot break on by using software breakpoints alone. In addition, any of the debugger's basic features available with software breakpoints can also be used with hardware breakpoints. As a result, you can take advantage of all the step and run commands.

❑ **Set up EMU0/1 pins.** In a system of multiple processors connected by EMU0/1 (emulation event) pins, setting up the EMU0/1 pins allows you to create global breakpoints. Using global breakpoints allows you to halt **all** (or some) processors in the system whenever a particular processor reaches a breakpoint (software or hardware). The EMU0/1 pins are connected internally to each MP and PP processor within an MVP. See the *Setting up the EMU0/1 pins* discussion on page DB:13-12 for more information.

## 13.2 An Overview of the Analysis Process

Completing an analysis session consists of four simple steps:



**Step 1**

Enable the analysis interface. See *Enabling the Analysis Interface*, page DB:13-4.

**Step 2**

Define breakpoint conditions. See *Defining the Conditions for Analysis*, page DB:13-5.

**Step 3**

Run your program. See *Running Your Program*, page DB:13-14.

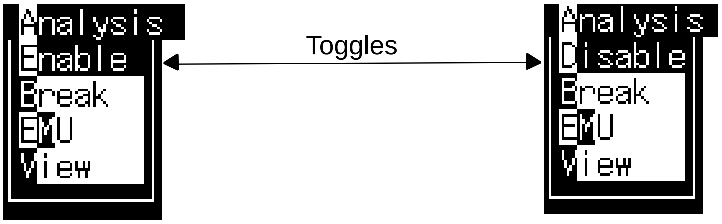
**Step 4**

View the analysis data. See *Viewing Analysis Data,* page DB:13-15.

## 13.3 Enabling the Analysis Interface

To begin tracking hardware events, you must explicitly enable the interface by selecting **Enable** on the Analysis menu. When you select enable, the next time you open the menu, Enable is replaced by **Disable**.

Figure 13–1. Enabling/Disabling the Analysis Interface



Selecting Disable turns the interface off; however, all events you previously enabled remain unchanged. By default, when the debugger comes up, the analysis interface is disabled.

**Note:**

You have to enable the analysis interface only once during a debugging session. It is not necessary to enable the analysis interface each time you run your program.

## 13.4 Defining Conditions for Analysis

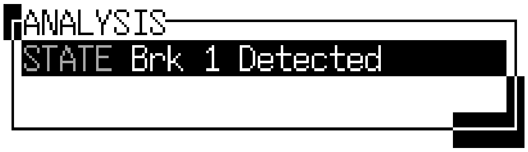
The analysis module detects hardware events and monitors the internal signals of the processor. The interface to the analysis module allows you to define parameters that halt the processor.

First, however, you must define the conditions the analysis interface must meet to track a particular event. To do this, select the events you want to track by enabling the appropriate options in the Analysis break events or Emulator Pins dialog boxes found on the Analysis menu.

### *Halting the processor*

You can set a hardware breakpoint on two basic types of events:

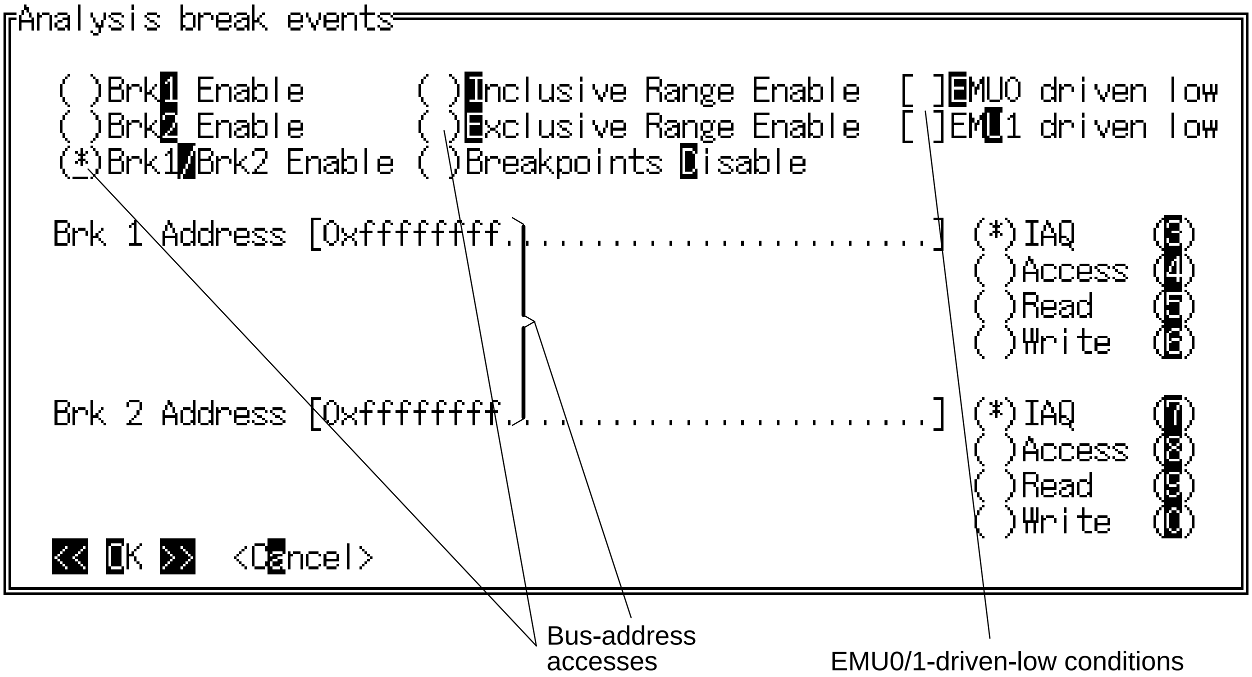
❑ Bus–address accesses
❑ EMU0/1-driven-low conditions



```
ANALYSIS
STATE Brk 1 Detected
```

Figure 13–2 shows the Analysis break events dialog box and the different types of events that you can select.

Figure 13–2. Two Basic Types of Events



```
Analysis break events

 ( )Brk1 Enable      ( )Inclusive Range Enable  [ ]EMU0 driven low
 ( )Brk2 Enable      ( )Exclusive Range Enable  [ ]EMU1 driven low
 (*)Brk1/Brk2 Enable ( )Breakpoints Disable

 Brk 1 Address [0xffffffff.........................] (*)IAQ    (3)
                                                     ( )Access (4)
                                                     ( )Read   (5)
                                                     ( )Write  (6)

 Brk 2 Address [0xffffffff.........................] (*)IAQ    (7)
                                                     ( )Access (8)
                                                     ( )Read   (9)
                                                     ( )Write  (0)

 << OK >>  <Cancel>
```

You enable events in the Analysis break events dialog box in the following two ways:

- The bus-address access events are mutually exclusive; only one event is enabled at a time. When a bus-address access event is enabled, the debugger displays an asterisk inside the parentheses preceding your selection. Use the Breakpoints Disable option to turn off all bus-address access events.
- Enabling the EMU0/1 events is like turning a switch on and off. When an event is enabled, the debugger displays an X in the square brackets preceding the event. You can enable as many of these events as you want.

**Note:**

If you set a hardware and a software breakpoint at the same location, the emulator cannot advance. Every time you enter a RUN or STEP command from this condition, the hardware and software breakpoints toggle, which prevents the pipeline from advancing. To proceed, you must either delete the software breakpoint or disable the hardware breakpoint.

### *Setting up the event comparators*

The analysis module has two breakpoint event comparators for use with either the instruction or data bus for the MP, and with the instruction, global, or local bus for the PP. You can set up the event comparators in the following ways:

- ❑ A single breakpoint on either bus

  The processor halts on accesses to a specified address on a specified bus.

- ❑ Two breakpoints on the same or different buses

  The processor halts on accesses to either of two specified addresses on a specified bus.

- ❑ An inclusive or exclusive range on a single bus
  - ■ An **inclusive** range refers to all of the addresses between the first and second breakpoint addresses entered, inclusive of the breakpoint addresses.

    For example, assume that the first breakpoint address is 0x0200 0050 and the second breakpoint address is 0x0400 0050. The range includes the addresses that are greater than or equal to 0x0200 0050 **and** that are less than or equal to 0x0400 0050.

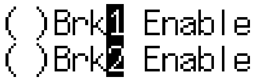
  - ■ An **exclusive** range refers to all of the addresses less than the first breakpoint address entered and greater than the second breakpoint address entered.

    For example, assume that the first breakpoint address is 0x0200 0050 and the second breakpoint is 0x0400 0050. The range includes the addresses that are less than 0x0200 0050 **and** the addresses that are greater than 0x0400 0050.

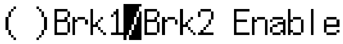
To define hardware breakpoint parameters, select **Break** on the Analysis menu. This brings up the Analysis break events dialog box (shown in Figure 13–3 for the MP and Figure 13–4 for the PP) that you can use to select the bus comparator configuration, address qualification, access qualification, and emulator pins.

- ❑ **Bus comparator configuration**. You can use the bus comparators along with address and access qualification selections to set up three comparator situations:

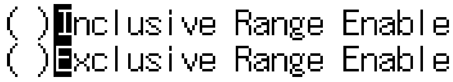
  - ■ To configure the comparators for single-point breakpoints, select one of the following:

  - ■ To configure the comparators as two single-point breakpoints, select the following:

  - ■ To configure the comparators for a range of breakpoints, select one of the following:

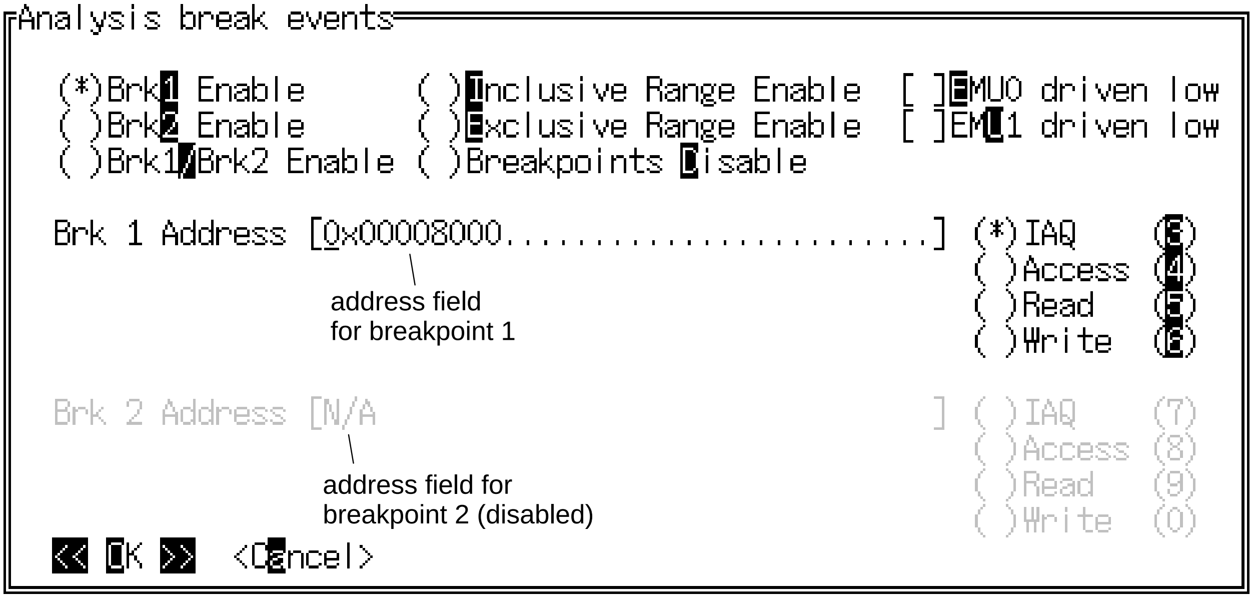
  When you select a comparator configuration, the debugger disables any unnecessary address qualification and access qualification fields. For example, if you select Brk1 Enable, the debugger disables the Brk2 Address field. You can select only one bus-address comparator at a time.

- ❑ **Address qualification**. The breakpoint address fields of the Analysis break events dialog box allow you to enter an address expression (a specific address, symbol, or function name) for each bus comparator. If you select a single-point breakpoint configuration, the address field for the other comparator is disabled (see Figure 13–3).

**Note:**

Both the MP and PP support address masking of breakpoints. The three LSBs of the bus comparator are masked dynamically; the masking is dependent on the bus size of the access. For example, if you set a breakpoint on a byte boundary and the processor accesses memory on the halfword or fullword boundary that contains the byte, the breakpoint event is taken.

Figure 13–3. Single Breakpoint on Instruction Bus



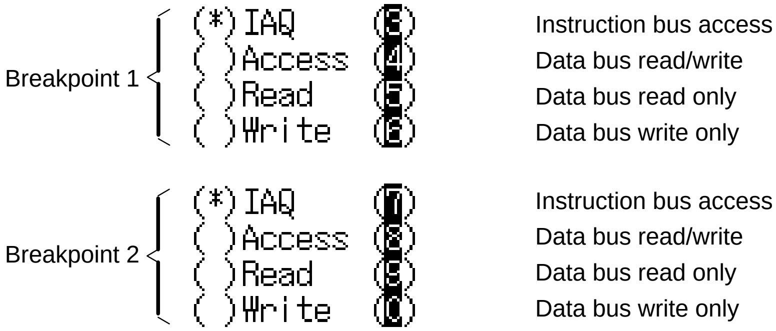
```
Analysis break events

  (*)Brk1 Enable       ( )Inclusive Range Enable  [ ]EMU0 driven low
  ( )Brk2 Enable       ( )Exclusive Range Enable  [ ]EMU1 driven low
  ( )Brk1/Brk2 Enable  ( )Breakpoints Disable

  Brk 1 Address [0x00008000.........................] (*)IAQ    (3)
                                                      ( )Access (4)
                                                      ( )Read   (5)
                                                      ( )Write  (6)

  Brk 2 Address [N/A                                ] ( )IAQ    (7)
                                                      ( )Access (8)
                                                      ( )Read   (9)
                                                      ( )Write  (0)
  << OK >>  <Cancel>
```

address field for breakpoint 1

address field for breakpoint 2 (disabled)

- ❑ **Access qualification**. The access qualification selection determines which bus is affected and what type of access causes the breakpoint. You select the access qualification after enabling a bus comparator and entering the appropriate address qualification. If you select a breakpoint range, the Brk 2 access qualification fields are disabled, even though the Brk 2 Address field remains active.

  - ■ You can select MP instruction or data bus access qualification with the following:



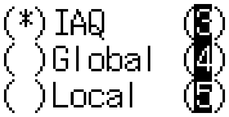
Breakpoint 1

```
(*)IAQ      (3)    Instruction bus access
( )Access   (4)    Data bus read/write
( )Read     (5)    Data bus read only
( )Write    (6)    Data bus write only
```

Breakpoint 2

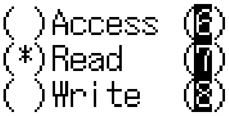
```
(*)IAQ      (7)    Instruction bus access
( )Access   (8)    Data bus read/write
( )Read     (9)    Data bus read only
( )Write    (0)    Data bus write only
```

  - ■ You can select PP instruction, global, or local bus qualification with the following:



```
(*)IAQ      (3)
( )Global   (4)
( )Local    (5)
```

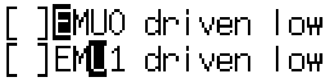
If you selected the PP global or local bus, you can qualify the access with the following:



```
( )Access   (6)
(*)Read     (7)
( )Write    (8)
```

For an example of the Analysis break events dialog box that appears for the PP, see Figure 13–4.

- **Emulator pins**. You can select EMU0/1 (emulation event) breakpoints in addition to a single bus comparator. To configure the analysis module to break on an EMU0/1 event, select one or both of the following:



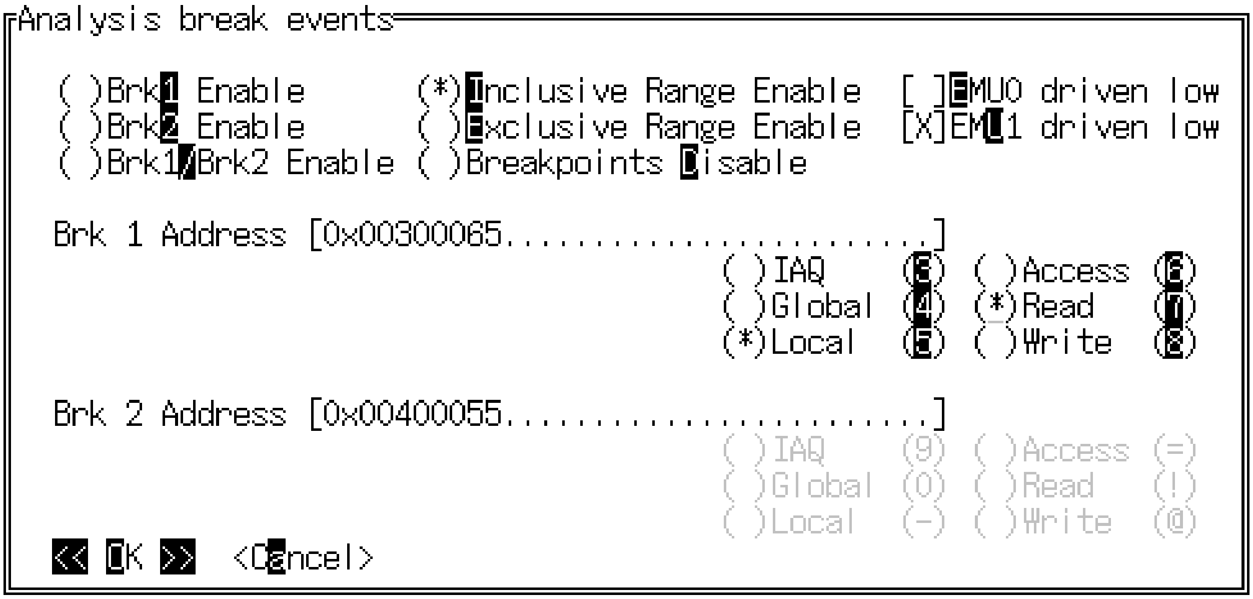
```
[ ]EMU0 driven low
[ ]EMU1 driven low
```

When you have finished making entries in the Analysis break events dialog box, select <OK> to accept the new configuration or <Cancel> to discard the changes and retain the previous configuration. The return or enter key has the same effect as selecting <OK>.

Figure 13–4 shows an example of an inclusive range that is enabled for the PP local bus. Since a range comparator is enabled, the address fields for both breakpoints are active, but the breakpoint 2 access qualification fields are disabled. The processor will halt when a read occurs on any address within the entered range (0x0030 0065 to 0x0040 0055). See page DB:13-7 for more information on inclusive and exclusive ranges.

Also in Figure 13–4, the EMU1-driven-low condition is enabled. When EMU1 is driven low, the processor halts.

Figure 13–4. An Inclusive Range for the PP Local Bus and EMU1 Condition Enabled

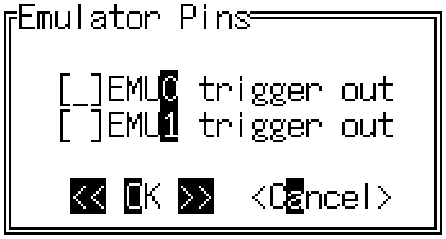


```
Analysis break events

  ( )Brk1 Enable      (*)Inclusive Range Enable  [ ]EMU0 driven low
  ( )Brk2 Enable      ( )Exclusive Range Enable  [X]EMU1 driven low
  ( )Brk1/Brk2 Enable ( )Breakpoints Disable

  Brk 1 Address [0x00300065.........................]
                                        ( )IAQ    (3) ( )Access (6)
                                        ( )Global (4) (*)Read   (7)
                                        (*)Local  (5) ( )Write  (8)

  Brk 2 Address [0x00400055.........................]
                                        ( )IAQ    (9) ( )Access (=)
                                        ( )Global (0) ( )Read   (!)
                                        ( )Local  (-) ( )Write  (@)

  << OK >>  <Cancel>
```

## Setting up the EMU0/1 pins

The analysis interface allows you to access and set up the EMU0/1 (emulation event) pins on your processors. You can use these pins to set global breakpoints. Selecting **EMU** from the Analysis menu opens the Emulator Pins dialog box, shown in Figure 13–5.

Figure 13–5. The Emulator Pins Dialog Box



```
Emulator Pins

  [_]EMU0 trigger out
  [ ]EMU1 trigger out

  << OK >>  <Cancel>
```

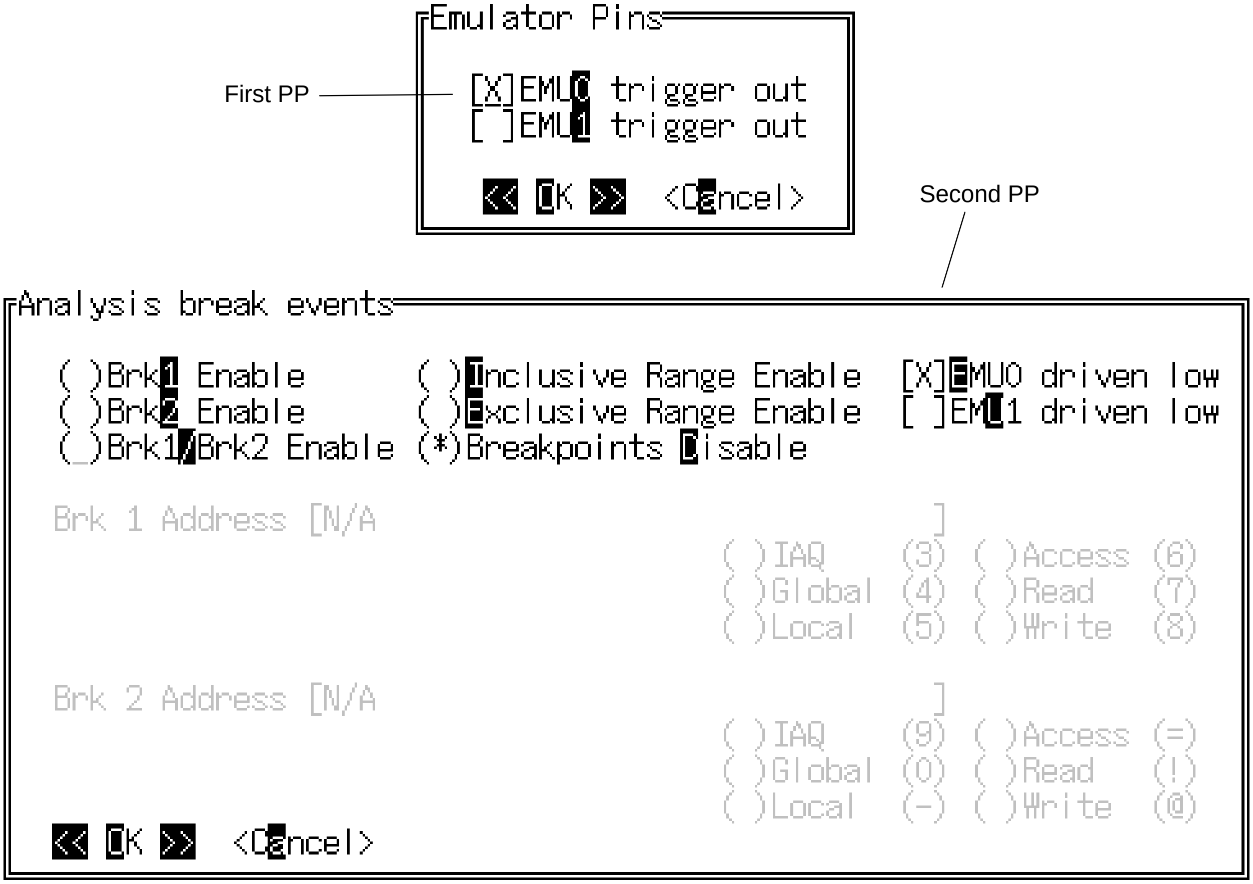
By default, the EMU0/1 pins are set up as input signals; however, you can set them up to be output signals, or **trigger out**, whenever the MP or PP is halted by a software or hardware breakpoint. This is extremely useful when you have multiple MVP processors in a system connected by their EMU0/1 pins. You can select an EMU0/1 pin to trigger out for a **single** MP or PP within the system.

Selecting EMU0/1 does not, however, automatically halt all processors in the system. To do so, you must enable the EMU0/1-driven-low condition in the Analysis break events dialog box for each processor you want halted. For example, if you have a system consisting of two PPs and you want to halt both PPs when the EMU0 pin for the first PP is driven low, you must enable the **EMU0 trigger out** parameter for the first PP. Then you must enable the parameter **EMU0 driven low** in the Analysis break events dialog box for the second PP. See Figure 13–6.

**Note:**

For the MP and PPs within an MVP, the EMU0/1 pins are already connected. If you are using MPs or PPs from multiple MVPs, you must connect the EMU0/1 pins if you want to halt the processors together.

Figure 13–6. Setting Up Global Breakpoints on a System of Two PPs



```
Emulator Pins

  [X]EMU0 trigger out
  [ ]EMU1 trigger out

   << OK >>  <Cancel>
```

```
Analysis break events

 ( )Brk1 Enable      ( )Inclusive Range Enable  [X]EMU0 driven low
 ( )Brk2 Enable      ( )Exclusive Range Enable  [ ]EMU1 driven low
 (_)Brk1/Brk2 Enable (*)Breakpoints Disable

 Brk 1 Address [N/A                        ]
                                 ( )IAQ    (3) ( )Access (6)
                                 ( )Global (4) ( )Read   (7)
                                 ( )Local  (5) ( )Write  (8)

 Brk 2 Address [N/A                        ]
                                 ( )IAQ    (9) ( )Access (=)
                                 ( )Global (0) ( )Read   (!)
                                 ( )Local  (-) ( )Write  (@)
 << OK >>  <Cancel>
```

Setting up the processors in this way creates a global breakpoint so that when the first processor reaches a breakpoint it halts all the other processors in the system.

**Note:**

If code execution stops at a software or hardware breakpoint, the next RUN command will step over the breakpoint before execution continues. Global breakpoints are disabled during the breakpoint stepover process. A side effect of this is that when you STEP from a software breakpoint, it will take two STEP commands before the global breakpoint (EMU0/1 trigger out) is generated.

Refer to subsection 1.9.3, *Using Emulation Pins*, in the *JTAG/MPSD Emulation Technical Reference*, for design considerations concerning EMU0/1 pins.

## 13.5 Running Your Program

Once you have defined your parameters, the analysis interface can begin collecting data as soon as you run your program. It stops collecting data when the defined conditions are met. The analysis interface monitors the progress of the defined events while your program is running. The basic syntax for the RUN command is:

**run** [*expression*]

You can use any of the debugger run commands (STEP, CSTEP, NEXT, etc.) described in Chapter 8, *Loading, Displaying, and Running Code*, except the emulator RUNF (run free) command.

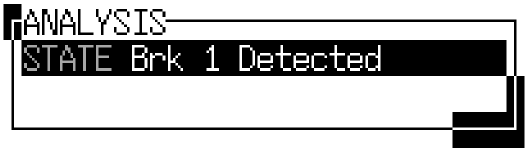
**Note:**

The conditions for the analysis session must be defined **before** your analysis session begins; you cannot change conditions during execution of your program.

## 13.6 Viewing the Analysis Data

You can monitor the status of the analysis interface by selecting **View** on the Analysis menu. This window displays an ongoing progress report of the analysis module's activity. Through this window, you can monitor the status of the breakpoint events. An example of the Analysis window is shown below.

Figure 13–7. Analysis Interface View Window Displaying an Ongoing Status Report



```
ANALYSIS
STATE Brk 1 Detected
```

The STATE field displays a list of the events that caused the processor to halt (such as **Brk 1 Detected**). If the analysis interface itself did not halt the processor but something else (such as a software breakpoint) did, the status line displays **No event detected**.

Multiple events can cause the processor to halt at the same time; these events are reflected in the STATE field of the Analysis window.

Chapter 14

# Summary of Commands and Special Keys

This chapter summarizes the basic debugger and parallel debug manager (PDM) commands and the debugger's special key sequences.

**Topics**

## 14.1 Functional Summary of Debugger and PDM Commands

This section summarizes commands by these categories:

- ❑ **Managing multiple debuggers.** These commands allow you to group debuggers, run code on multiple processors, and send commands to a group of debuggers.
- ❑ **Changing modes.** These commands enable you to switch freely between the three debugging modes (auto, mixed, and assembly). You can select these commands from the Mode pulldown menu, also.
- ❑ **Managing windows.** These commands enable you to select the active window and move or resize the active window. You can perform these functions with the mouse, also.
- ❑ **Performing system tasks.** These commands enable you to perform several system-like functions and provide you with some control over the target system.
- ❑ **Displaying files and loading programs.** These commands enable you to change the displays in the FILE and DISASSEMBLY windows and to load object files into memory. Several of these commands are available on the Load pulldown menu.
- ❑ **Displaying and changing data.** These commands enable you to display and evaluate a variety of data items. Some of these commands are available on the Watch pulldown menu, also.
- ❑ **Managing breakpoints.** These commands provide you with a command-line method for controlling software breakpoints and are also available through the Break pulldown menu. You can also set/clear breakpoints interactively.
- ❑ **Memory mapping.** These commands enable you to define the areas of target memory that the debugger can access or to fill a memory range with an initial value. You can also use the Memory pulldown menu to access these commands.
- ❑ **Customizing the screen.** These commands allow you to customize the debugger display, then save and later reuse the customized displays. You can also use the Color pulldown menu to access these commands.
- ❑ **Running programs.** These commands provide you with a variety of methods for running your programs in the debugger environment. The basic run and single-step commands are available on the menu bar, also.

## Managing multiple debuggers

| To do this | Use this command | See page |
|---|---|---|
| Use the command history | ! | DB:14-11 |
| Assign a variable to the result of an expression | @ | DB:14-11 |
| Define a custom command string | alias | DB:14-13 |
| Record the information shown in the display area of the parallel debug manager | dlog | DB:14-22 |
| Display a string to the display area of the parallel debug manager | echo | DB:14-23 |
| Evaluate an expression in a debugger or group of debuggers and set a variable to the result | eval | DB:14-25 |
| List available PDM commands | help | DB:14-28 |
| List the last 20 commands | history | DB:14-28 |
| Conditionally execute PDM commands | if/elif/else/endif | DB:14-28 |
| Loop through PDM commands | loop/break/ continue/ endloop | DB:14-32 |
| Pause the parallel debug manager | pause | DB:14-39 |
| Halt code execution | pesc | DB:14-39 |
| Global halt | phalt | DB:14-40 |
| Run code globally | prun | DB:14-41 |
| Run free globally | prunf | DB:14-42 |
| Single-step globally | pstep | DB:14-42 |
| Exit any debugger and/or the parallel debug manager | quit | DB:14-43 |
| Send a command to an individual processor or a group of processors | send | DB:14-49 |
| Change the parallel debug manager prompt | set | DB:14-50 |
| Create your own system variables | set | DB:14-50 |
| Define or modify a group of processors | set | DB:14-50 |
| List all system variables or groups of processors | set | DB:14-50 |
| Set the default group | set | DB:14-50 |
| Invoke an individual debugger | spawn | DB:14-54 |
| Find the execution status of a processor or a group of processors | stat | DB:14-57 |
| Enter an operating-system command | system | DB:14-58 |
| Execute a batch file | take | DB:14-58 |
| Delete an alias definition | unalias | DB:14-59 |
| Delete a group or system variable | unset | DB:14-59 |

## Changing modes

| To do this | Use this command | See page |
|---|---|---|
| Put the debugger in assembly mode | asm | DB:14-13 |
| Put the debugger in auto mode for debugging C code | c | DB:14-16 |
| Put the debugger in mixed mode | mix | DB:14-36 |

## Managing windows

| To do this | Use this command | See page |
|---|---|---|
| Reposition the active window | move | DB:14-37 |
| Resize the active window | size | DB:14-55 |
| Select the active window | win | DB:14-62 |
| Make the active window as large as possible | zoom | DB:14-63 |

## Performing system tasks

| To do this | Use this command | See page |
|---|---|---|
| Define your own command string | alias | DB:14-13 |
| Clear all displayed information from the display area of the COMMAND window | cls | DB:14-16 |
| Record the information shown in the display area of the COMMAND window | dlog | DB:14-22 |
| Display a string to the COMMAND window while executing a batch file | echo | DB:14-23 |
| Conditionally execute debugger commands in a batch file | if/else/endif | DB:14-29 |
| Loop debugger commands in a batch file | loop/endloop | DB:14-31 |
| Pause the execution of a batch file | pause | DB:14-39 |
| Exit the debugger | quit | DB:14-43 |
| Reset the target system (emulator only) or the simulator | reset | DB:14-44 |
| Associate a beeping sound with the display of error messages | sound | DB:14-56 |
| Execute commands from a batch file | take | DB:14-58 |
| Delete an alias definition | unalias | DB:14-59 |
| Name additional directories that can be searched when you load source files | use | DB:14-60 |

## Displaying files and loading programs

| To do this | Use this command | See page |
|---|---|---|
| Display C and/or assembly language code at a specific point | addr | DB:14-12 |
| Reopen the CALLS window | calls | DB:14-16 |
| Display assembly language code at a specific address | dasm | DB:14-19 |
| Display a text file in the FILE window | file | DB:14-26 |
| Display a specific C function | func | DB:14-26 |
| Load an object file | load | DB:14-31 |
| Load only the object-code portion of an object file | reload | DB:14-43 |
| Load only the symbol-table portion of an object file | sload | DB:14-56 |

## Displaying and changing data

| To do this | Use this command | See page |
|---|---|---|
| Evaluate and display the result of a C expression | ? | DB:14-10 |
| Display the values in an array or structure or display the value that a pointer is pointing to | disp | DB:14-20 |
| Evaluate a C expression without displaying the results | eval | DB:14-24 |
| Display a different range of memory in the MEMORY window | mem | DB:14-35 |
| Display a pop-up MEMORY window | mem1,<br>mem2,<br>mem3 | DB:14-35 |
| Change the default format for displaying data values | setf | DB:14-52 |
| Continuously display the value of a variable, register, or memory location within the WATCH window | wa | DB:14-61 |
| Delete a data item from the WATCH window | wd | DB:14-62 |
| Show the type of a data item | whatis | DB:14-62 |
| Delete all data items from the WATCH window and close the WATCH window | wr | DB:14-63 |

## Managing breakpoints

| To do this | Use this command | See page |
|---|---|---|
| Add a software breakpoint | ba | DB:14-14 |
| Delete a software breakpoint | bd | DB:14-14 |
| Display a list of all software breakpoints that are set | bl | DB:14-14 |
| Reset (delete) all software breakpoints | br | DB:14-15 |

## Memory mapping

| To do this | Use this command | See page |
|---|---|---|
| Initialize a block of memory | fill | DB:14-26 |
| Add an address range to the memory map | ma | DB:14-33 |
| Enable or disable memory mapping | map | DB:14-35 |
| Delete an address range from the memory map | md | DB:14-35 |
| Display a list of the current memory map settings | ml | DB:14-36 |
| Reset the memory map (delete all ranges) | mr | DB:14-37 |
| Save a block of memory to a system file | ms | DB:14-38 |

## Customizing the screen

| To do this | Use this command | See page |
|---|---|---|
| Change the border style of any window | border | DB:14-15 |
| Change the screen colors, but don’t update the screen immediately | color | DB:14-18 |
| Change the command-line prompt | prompt | DB:14-40 |
| Change the screen colors and update the screen immediately | scolor | DB:14-47 |
| Load and use a previously saved custom screen configuration | sconfig | DB:14-48 |
| Save a custom screen configuration | ssave | DB:14-56 |

## Running programs

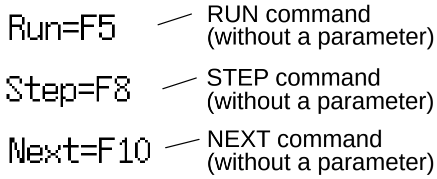
| To do this | Use this command | See page |
|---|---|---|
| Single-step through assembly language or C code, one C statement at a time; step over function calls | cnext | DB:14-17 |
| Single-step through assembly language or C code, one C statement at a time | cstep | DB:14-19 |
| Run a program up to a certain point | go | DB:14-27 |
| Halt the target system after executing a RUNF command (emulator only) | halt | DB:14-27 |
| Single-step through assembly language or C code; step over function calls | next | DB:14-38 |
| Reset the target system (emulator only) or the simulator | reset | DB:14-44 |
| Reset the program entry point | restart | DB:14-44 |
| Execute code in a function and return to the function’s caller | return | DB:14-44 |
| Run a program | run | DB:14-45 |
| Run a program with benchmarking—count the number of CPU clock cycles consumed by the executing portion of code | runb | DB:14-45 |
| Disconnect the emulator from the target system and run free (emulator only) | runf | DB:14-46 |
| Single-step through assembly language or C code | step | DB:14-57 |
| Execute commands from a batch file | take | DB:14-58 |

## 14.2 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.

Remember, you can use the menus with or without a mouse. To access a menu from the keyboard, press the ALT key and the letter that's highlighted in the menu name. (For example, to display the Load menu, press ALT L.) Then, to make a selection from the menu, press the letter that's highlighted in the command you've selected. (For example, on the Load menu, to execute FIle, press F.) If you don't want to execute a command, press ESC to close the menu.

### *Program-execution commands*

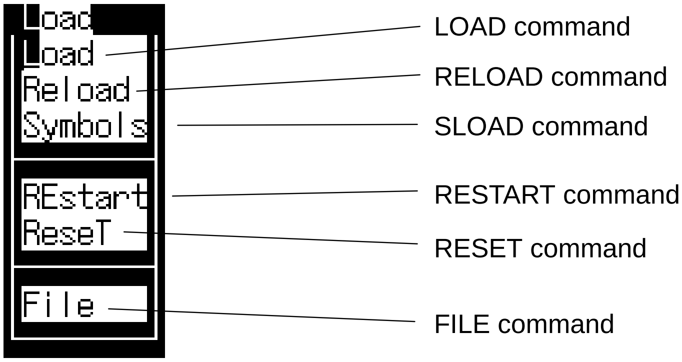


RUN command (without a parameter)

STEP command (without a parameter)

NEXT command (without a parameter)

### *File/load commands*



LOAD command

RELOAD command

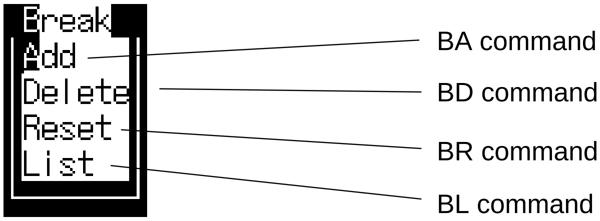
SLOAD command

RESTART command

RESET command

FILE command

### *Breakpoint commands*



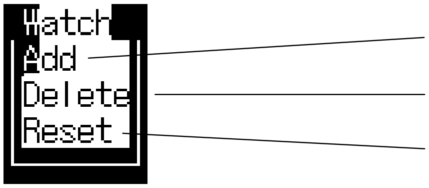
BA command

BD command

BR command

BL command

## *Watch commands*

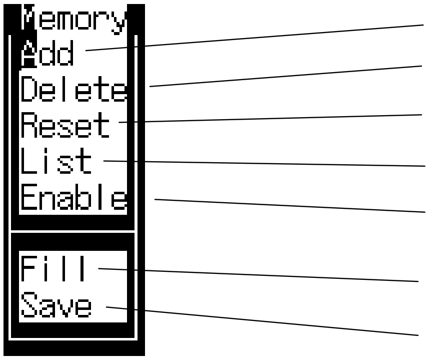


WA command

WD command

WR command

## *Memory commands*



MA command

MD command

MR command

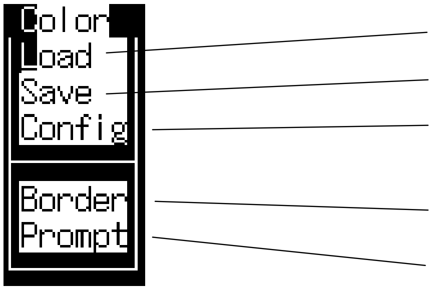
ML command

MAP command

FILL command

MS command

## *Screen-configuration commands*



SCONFIG command

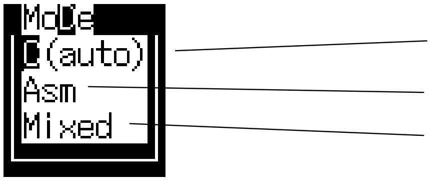
SSAVE command

SCOLOR command

BORDER command

PROMPT command

## *Mode commands*



C command

ASM command

MIX command

## 14.3 Alphabetical Summary of Debugger and PDM Commands

Most of the commands can be used with both the MP and PP debuggers in the basic debugger environment. Other commands can be used only by the parallel debug manager (PDM). A few commands can be used in both environments. Each command description identifies the applicable environments for the command.

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

### ?

Evaluate Expression

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

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**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 (one that modifies the value of the symbols used in the expression); however, you cannot use a string constant or function call in the *expression.* If the result of *expression* is not an array or structure, then the debugger displays the results in the COMMAND window. If *expression* is a structure or array, ? displays the entire contents of the structure or array; you can halt long listings by pressing ESC.

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

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

## !

Use the Parallel Debug Manager Command History

**Syntax** **!**{*prompt number* **|** *string*}
**!!**

**Menu selection** none

**Environments** □ basic debugger ☑ PDM

**Description** The parallel debug manager supports a command history that is similar to the UNIX command history. The parallel debug manager prompt identifies the number of the current command. This number is incremented with every command. The parallel debug manager command history allows you to re-enter any of the last 20 commands.

- The *number* parameter is the number of the parallel debug manager prompt that contains the command that you want to re-enter.
- The *string* parameter tells the parallel debug manager to execute the last command that began with *string.*
- The !! command tells the parallel debug manager to execute the last command that you entered.

## @

Substitute Result of an Expression

**Syntax** @ *variable name* **=** *expression*

**Menu selection** none

**Environments** □ basic debugger ☑ PDM

**Description** Unlike the SET command, the @ command first evaluates the *expression*, and then sets the *variable name* to the result. The *expression* can be any expression that uses the symbols described in Section 4.7, *Understanding the Parallel Debug Manager's Expression Analysis*. The *variable name* can consist of up to 128 alphanumeric characters or underscore characters.

## addr Display Code at Specified Address

**Syntax**

**addr** *address*
**addr** *function name*

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**Description**

Use the ADDR command to display C code or the disassembly at a specific point. ADDR's behavior changes, depending on the current debugging mode:

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

**Note:**

ADDR affects the FILE window only if the specified *address* is in a C function.

## alias

Define Custom Command String

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

**Menu selection** none

**Environments** ☑ basic debugger ☑ PDM

**Description** You can use the ALIAS command to define customized command strings for the debugger and for the parallel debug manager:

- The debugger version of the ALIAS command allows you to associate one or more debugger commands with a single *alias name.*
- The parallel debug manager version of the ALIAS command allows you to associate one or more PDM commands with a single *alias name* or associate one or more debugger commands with a single *alias name*.

You can include as many commands in the *command string* as you like, as long you separate them with semicolons and enclose the entire string of commands in quotation marks. 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 aliases can be included as part of the definition for a new alias.

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

## asm

Enter Assembly Mode

**Syntax** **asm**

**Menu selection** Mo**D**e→**A**sm

**Environments** ☑ basic debugger □ PDM

**Description** The ASM command changes from the current debugging mode to assembly mode. If you’re already in assembly mode, the ASM command has no effect.

## ba — Add Software Breakpoint

**Syntax** **ba** *address*

**Menu selection** **B**reak→**A**dd

**Environments** ☑ basic debugger □ PDM

**Description** The BA command sets a software breakpoint at a specific *address*. This command is useful because it doesn't require you to search through code to find the desired line. The *address* can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label.

## bd — Delete Software Breakpoint

**Syntax** **bd** *address*

**Menu selection** **B**reak→ **D**elete

**Environments** ☑ basic debugger □ PDM

**Description** The BD command clears a software breakpoint at a specific *address*. The *address* can be an absolute address, any C expression, the name of a C function, or the name of an assembly language label.

## bl — List Software Breakpoints

**Syntax** **bl**

**Menu selection** **B**reak→**L**ist

**Environments** ☑ basic debugger □ PDM

**Description** The BL command provides an easy way to get a complete listing of all software breakpoints that are currently set in your program. It displays a table of breakpoints in the display area of the COMMAND window. BL lists all the breakpoints that are set, in the order in which you set them.

## border — Change Style of Window Border

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

**Menu selection** **C**olor→**B**order

**Environments** ☑ basic debugger ☐ PDM

**Description** The BORDER command changes the border style of the active window, the inactive windows, and the border style of any window that you're resizing. The debugger supports nine border styles. Each parameter for the BORDER command must be one of the numbers that identifies these styles:

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

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

## br — Reset Software Breakpoint

**Syntax** **br**

**Menu selection** **B**reak→**R**eset

**Environments** ☑ basic debugger ☐ PDM

**Description** The BR command clears all software breakpoints that are set.

## c

Enter Auto Mode

**Syntax** **c**

**Menu selection** Mo**D**e→**C** (auto)

**Environments** ☑ basic debugger □ PDM

**Description** The C command changes from the current debugging mode to auto mode. If you’re already in auto mode, then the C command has no effect.

## calls

Open CALLS Window

**Syntax** **calls**

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**Description** The CALLS command displays the CALLS window. The debugger displays this window automatically when you are in auto/C or mixed mode. However, you can close the CALLS window; the CALLS command opens the window again.

## cls

Clear Screen

**Syntax** **cls**

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**Description** The CLS command clears all displayed information from the display area of the COMMAND window.

## cnext — Single-Step C, Next Statement

**Syntax** **cnext** [*expression*]

**Menu selection** Next=**F10** (in C code)

**Environments** ☑ basic debugger □ PDM

**Description** The CNEXT command is similar to the CSTEP command. It runs a program one C statement at a time, updating the display after executing each statement. If you're using CNEXT to step through assembly language code, the debugger won't update the display until it has executed all assembly language statements associated with a single C statement. Unlike CSTEP, CNEXT steps over function calls rather than stepping into them—you don't see the single-step execution of the function call.

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page DB:8-18, discusses this in detail).

## color

Change Screen Colors

**Syntax** **color** *area name,* $attr_1$ [,$attr_2$ [,$attr_3$ [,$attr_4$] ] ]

**Menu selection** none

**Environments** ☑ basic debugger ☐ PDM

**Description**

The COLOR command changes the color of specified areas of the debugger display. COLOR doesn't update the display; the changes take effect when another command, such as SCOLOR, updates the display. The *area name* parameter identifies the areas of the display that are affected. The *attributes* identify how the areas are affected. The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Valid values for the *attribute* parameters include:

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

Valid values for the *area name* parameters include:

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

You don't have to type an entire *attribute* or *area name*; you need to type only enough letters to uniquely identify the attribute. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they're listed above (left to right, top to bottom).

## cstep

Single-Step C

**Syntax** **cstep** [*expression*]

**Menu selection** Step=**F8** (in C code)

**Environments** ☑ basic debugger ☐ PDM

**Description**

The CSTEP single-steps through a program one C statement at a time, updating the display after executing each statement. If you're using CSTEP to step through assembly language code, the debugger won't update the display until it has executed all assembly language statements associated with a single C statement.

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

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page DB:8-18, discusses this in detail).

## dasm

Display Disassembly at Specified Address

**Syntax**

**dasm** *address*
**dasm** *function name*

**Menu selection** none

**Environments** ☑ basic debugger ☐ PDM

**Description**

The DASM command displays code beginning at a specific point within the DISASSEMBLY window.

## dataplot

Bring Up Data Flow Plot **Simulator Only**

**Syntax** **dataplot**

**Menu selection** none

**Environments** ☑ basic debugger ☐ PDM

**Description** The DATAPLOT command causes the debugger to display a data flow plot in its own X window. The data flow plot provides a cycle-by-cycle display of processor status and memory accesses.

## disp

Open DISP Window

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

**Menu selection** none

**Environments** ☑ basic debugger ☐ PDM

**Description** The DISP command opens a DISP window to display the contents of an array, structure, or pointer expressions to a scalar type (of the form **pointer*). If the *expression* is not one of these types, then DISP acts like a ? command. You can have up to 120 DISP windows open at the same time.

Once you open a DISP window, you may find that a displayed member is itself an array, structure, or pointer:

| | |
|---|---|
| A member that is an array looks like this | [. . .] |
| A member that is a structure looks like this | {. . .} |
| A member that is a pointer looks like an address | 0x00000000 |

You can display the additional data (the data pointed to or the members of the array or structure) in another DISP window by using the DISP command again, using the arrow keys to select the field and then pressing F9, or pointing the mouse cursor to the field and pressing the left mouse button.

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

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

The *display format* parameter can be used only when you are displaying a scalar type, an array of scalar type, or an individual member of an aggregate type.

You can also use the DISP command with a typecast expression to display memory contents in any format. Here are some examples:

```
disp *0
disp *(float *)123
disp *(char *)0x111
```

This shows memory in the DISP window as an array of locations; the location that you specify with the *expression* parameter is member [0], and all other locations are offset from that location.

## dlog — Record Display Window

**Syntax**

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

**Menu selection** none

**Environments** ☑ basic debugger ☑ PDM

**Description**

The DLOG command allows you to record the information displayed in the debugger's COMMAND window or in the display area of parallel debug manager into a log file.

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

  **dlog** *filename*

  Log files can be executed by using the TAKE command. When you use DLOG to record the information from the display area into a log file called *filename*, the debugger (or parallel debug manager) automatically precedes all error or progress messages and command results with a semicolon to turn them into comments. This way, you can easily re-execute the commands in your log file by using the TAKE command.

- To end the recording session, enter:

  **dlog close** ↵

If necessary, you can write over existing log files or append additional information to existing files. The optional parameters of the DLOG command control how existing log files are used:

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

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

## echo

Echo String to Display Area

**Syntax** **echo** *string*

**Menu selection** none

**Environments** ☑ basic debugger ☑ PDM

**Description** The ECHO command displays *string* in the display area of the COMMAND window or in the display area of the parallel debug manager. You can't use quote marks around the *string,* and any leading blanks in your command string are removed when the ECHO command is executed.

- You can execute the debugger version of the ECHO command only in a batch file.
- You can execute the parallel debug manager version of the ECHO command in a batch file or from the command line.

## elif

Test for Alternate Condition

**Environments** □ basic debugger ☑ PDM

**Description** ELIF provides an alternative test by which you can execute PDM commands in the IF/ELIF/ELSE/ENDIF command sequence. For more information about these commands, see page DB:14-28.

## else

Execute Alternative Commands

**Environments** ☑ basic debugger ☑ PDM

**Description** ELSE provides an alternative list of PDM or debugger commands in the IF/ELIF/ELSE/ENDIF or IF/ELSE/ENDIF command sequences, respectively. For more information about these commands, see pages DB:14-28 and DB:14-29.

## endif

Terminate Conditional Sequence

**Environments** ☑ basic debugger ☑ PDM

**Description** ENDIF identifies the end of a conditional-execution command sequence begun with an IF command. For more information about these commands, see pages DB:14-28 and DB:14-29.

## endloop

Terminate Looping Sequence

**Environments** ☑ basic debugger ☑ PDM

**Description** ENDLOOP identifies the end of the LOOP/ENDLOOP command sequence (for the debugger) or the LOOP/BREAK/CONTINUE/ENDLOOP sequence (for the parallel debug manager). For more information about the LOOP commands, see pages DB:14-31 and DB:14-32.

## eval

Evaluate Expression

**Syntax**

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

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**Description** The EVAL command evaluates an expression like the ? command does, **but does not show the result** in the display area of the COMMAND window. EVAL is useful for assigning values to registers or memory locations in a batch file (where it's not necessary to display the result).

## eval

Evaluate Expression and Set to Variable

**Syntax**

**eval** [**–g** {*group* **|** *processor name*}] *variable name***=***expression*[**,** *format*]

**Menu selection**

none

**Environments**

□ basic debugger ☑ PDM

**Description**

The EVAL command evaluates an expression in a debugger and sets a variable to the result of the expression.

- The **–g** option specifies the group or processor that EVAL should be sent to. If you don't use this option, the command is sent to the default group (dgroup).
- When you send the EVAL command to more than one processor, the parallel debug manager takes the *variable name* that you supply and appends a suffix for each processor. The suffix consists of the underscore character (_) followed by the name that you assigned the processor.
- The *expression* can be any expression that uses the symbols described in Section 4.7, *Understanding the Parallel Debug Manager's Expression Analysis*.
- When you use the optional *format* parameter, the value that the variable is set to will be in one of the following formats:

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

## file

Display Text File

**Syntax** **file** *filename*

**Menu selection** **L**oad→**F**ile

**Environments** ☑ basic debugger ☐ PDM

**Description** The FILE command displays the contents of any text file in the FILE window. The debugger continues to display this file until you run a program and halt in a C function. This command is intended primarily for displaying C source code. You can view only one text file at a time.

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

## fill

Fill Memory

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

**Menu selection** **M**emory→**F**ill

**Environments** ☑ basic debugger ☐ PDM

**Description** The FILL command fills a block of memory with a specified value.

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

## func

Display Function

**Syntax** **func** *function name*
**func** *address*

**Menu selection** none

**Environments** ☑ basic debugger ☐ PDM

**Description** The FUNC command displays a specified C function in the FILE window. You can identify the function by its name or its address. Note that FUNC works the same way FILE works; but with FUNC, you don't need to identify the name of the file that contains the function.

## go

Run to Specified Address

**Syntax** **go** [*address*]

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**Description** The GO command executes code up to a specific point in your program. If you don't supply an *address*, then GO acts like a RUN command without an *expression* parameter.

## halt

Halt Processor **Emulator Only**

**Syntax** **halt**

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**Description** The HALT command halts one or more processors. If you enter a RUNF command from the command line of an MP or PP debugger, you can use HALT to halt the associated processor. When you invoke a debugger, it automatically executes a HALT command. Thus, if you enter a RUNF, quit the debugger, and later reinvoke the debugger, you will effectively reconnect the processor and run the debugger in its normal mode of operation.

**Note:**

If a memory fault is detected by the MP while the MP's memory fault interrupt vector is pointing to a memory location that also generates a memory fault, the MP's instruction pipeline stalls. If the emulator is running when this occurs, the MP does not respond to the emulator's HALT command (ESC or HALT).

If a memory fault is detected by the PP, the PP's instruction pipeline stalls and waits for the MP to service the PP. If the emulator is running when this occurs, the PP does not respond to the emulator's HALT command (ESC or HALT).

## help

List Parallel Debug Manager Commands

**Syntax** **help** [*command*]

**Menu selection** none

**Environments** □ basic debugger ☑ PDM

**Description** The HELP command provides a brief description of the requested PDM *command*. If you omit the *command* parameter, the parallel debug manager lists all of the available PDM commands.

## history

List the Last 20 PDM Commands

**Syntax** **history**

**Menu selection** none

**Environments** □ basic debugger ☑ PDM

**Description** The HISTORY command displays the last 20 PDM commands that you've entered.

## if/elif/else/endif

Conditionally Execute PDM Commands

**Syntax**

**if** *expression*<br>
*PDM commands*<br>
[**elif** *expression*<br>
*PDM commands*]<br>
[**else**<br>
*PDM commands*]<br>
**endif**

**Menu selection** none

**Environments** □ basic debugger ☑ PDM

**Description** These commands allow you to execute PDM commands conditionally in a batch file or from the command line.

- ❑ If the *expression* for the IF is nonzero, the parallel debug manager executes all commands between the IF and ELIF, ELSE, or ENDIF.
- ❑ The ELIF is optional. If the *expression* for the ELIF is nonzero, the parallel debug manager executes all commands between the ELIF and ELSE or ENDIF.

- The ELSE is optional. If the *expressions* for the IF and ELIF (if present) are false (zero), the parallel debug manager executes the commands between the ELSE and ENDIF.

The IF/ELIF/ELSE/ENDIF can be entered interactively or included in a batch file that is executed by the TAKE command. When you enter IF from the command line of the parallel debug manager, a question mark (?) prompts you for the next entry. The parallel debug manager continues to prompt you for input using the ? until you enter ENDIF. After you enter ENDIF, the parallel debug manager immediately executes the IF command.

If you are in the middle of interactively entering an IF statement and want to abort it, type CONTROL C.

## if/else/endif

Conditionally Execute Debugger Commands

**Syntax**

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

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**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. Note that the ELSE portion of the command sequence is optional.

You can substitute a keyword for the *expression*. Keywords evaluate to true (1) or false (0). You can use the following keywords with the IF command:

- **$$EMU$$** (tests for the emulator version of the debugger)
- **$$SIM$$** (tests for the simulator version of the debugger)
- **$$MVP_MP$$** (tests for the MP version of the debugger)
- **$$MVP_PP$$** (tests for the PP version of the debugger)

The conditional commands work with the following provisions:

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

## image Display Image **Simulator Only**

**Syntax** **image** *addr* [**,** *width* [**,** *height* [**,** *color* [**,** *Cbaddr* [**,** *Craddr* [**,** *freq*]]]]]]

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**Description** The IMAGE command causes the debugger to display a grayscale or color representation of an image stored in memory. The image is displayed in its own X window.

If you enter this command without any parameters, the debugger displays a dialog box for you to enter the parameter values.

- The *addr* parameter defines the starting address byte of the image in memory. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.
- The *width* parameter defines the width, in pixels, of the image. The default width is 256 bytes.
- The *height* parameter defines the height, in pixels, of the image. The default height is 256 bytes.
- The *color* parameter defines the type of image that you want to display.

| To display this type of image. . . | Enter this color type |
|---|---|
| 8-bit grayscale | **8** or **8_bit** |
| 16-bit color | **16** or **16_bit** |
| 32-bit color | **32** or **32_bit** |
| YCbCr 4:1:1 | **ycbcr** or **ycbcr4:1:1** |

Note that the color type names are not case sensitive.

- The *Cbaddr* and *Craddr* parameters are necessary if you select YCbCr 4:1:1 as your color type. The *Cbaddr* parameter defines the Cb starting address, and the *Craddr* parameter defines the Cr starting address.
- The *freq* parameter determines how frequently the image is read from memory and redisplayed. For example, if you specify an update rate of 200 and you begin displaying the image at the 50th cycle, the image will be updated at 250 cycles, 450 cycles, etc. The default update frequency is 500 cycles.

## load

Load Executable Object File

**Syntax** **load** *object filename*

**Menu selection** **L**oad→ **L**oad

**Environments** ☑ basic debugger ☐ PDM

**Description** The LOAD command loads both an object file and its associated symbol table into memory. In effect, the LOAD command performs both a RELOAD and an SLOAD. If you don't supply an extension, the debugger looks for *filename*.out. Note that the LOAD command clears the old symbol table and closes the WATCH and DISP windows.

**Note:**

When you use the LOAD command in the MP debugger, the MP data cache is first "cleaned." In other words, all of the modified (dirty) bits are written back to external memory. Then, the instruction and data caches are cleared (all of the DTAG and ITAG registers are cleared).

## loop/endloop

Loop Through Debugger Commands

**Syntax**

**loop** *expression*
*debugger commands*
**endloop**

**Menu selection** none

**Environments** ☑ basic debugger ☐ PDM

**Description** The LOOP/ENDLOOP commands allow you to set up a looping situation in a batch file. These looping commands evaluate in the same method as in the run conditional command expression:

- If you use an *expression* that is not Boolean, the debugger evaluates the *expression* as a loop count.
- If you use a Boolean *expression*, the debugger executes the command repeatedly as long as the *expression* is true.

The LOOP/ENDLOOP commands work under the following conditions:

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

## loop/break/continue/endloop

Loop Through PDM Commands

**Syntax**

**loop** *Boolean expression*
*PDM commands*
[**break**]
[**continue**]
**endloop**

**Menu selection** none

**Environments** □ basic debugger ☑ PDM

**Description**

The LOOP/BREAK/CONTINUE/ENDLOOP commands allow you to set up a looping situation in a batch file or from the command line. Unlike the debugger version of the LOOP/ENDLOOP commands, the parallel debug manager version of the LOOP command evaluates only Boolean expressions:

- If the *Boolean expression* evaluates to true (1), the parallel debug manager executes all commands between the LOOP and BREAK, CONTINUE, or ENDLOOP.
- If the *Boolean expression* evaluates to false (0), the loop is not entered.

The optional BREAK command allows you to exit the loop without having to reach the ENDLOOP. This is helpful when you are testing a group of processors and want to exit if an error is detected.

The CONTINUE command, which is also optional, acts as a goto and returns command flow to the enclosing LOOP command. CONTINUE is useful when the part of the loop that follows is complicated and returning to the top of the loop avoids further nesting.

The LOOP/BREAK/CONTINUE/ENDLOOP commands can be entered interactively or included in a batch file that is executed by the TAKE command. When you enter LOOP from the command line of the parallel debug manager, a question mark (?) prompts you for the next entry. The parallel debug manager continues to prompt you for input using the ? until you enter ENDLOOP. After you enter ENDLOOP, the parallel debug manager immediately executes the LOOP command.

If you are in the middle of interactively entering an LOOP statement and want to abort it, type CONTROL C.

## ma

Add Block to Memory Map

**Syntax** **ma** *address, length, type* [*, pagesize*] [*, bussize*]

**Menu selection** **M**emory→**A**dd

**Environments** ☑ basic debugger □ PDM

**Description** The MA command identifies valid ranges of target memory. Note that a new memory map must not overlap an existing entry; if you define a range that overlaps an existing range, the debugger ignores the new range.

- The *address* parameter defines the starting address of a range in data or program memory. This parameter can be an absolute address, any C expression, the name of a C function, or an assembly language label.
- The *length* parameter defines the length of the range. This parameter can be any C expression and must be a multiple of the *pagesize* that you supply.
- The *type* parameter identifies one the following MVP memory types and its associated read/write characteristics of the memory range. The *type* must be one of these keywords shown in Table 14–1 if you're using the simulator, or Table 14–2 if you're using the emulator.

Table 14–1. Memory-Type Keywords for Use With the Simulator

| To identify this kind of simulator memory | That has these read/write characteristics, | Use this keyword as the *type* parameter |
|---|---|---|
| Pipelined one cycle/column | Read/write memory | **SRAM0** or **DRAM0** |
| Unpipelined one cycle/column (Default memory type) | Read/write memory | **SRAM, DRAM, SRAM1, DRAM1, WR,** or **RAM** |
| | Read-only memory | **R**, **ROM**, or **READONLY** |
| | Write-only memory | **W***,* **WOM***,* or **WRITEONLY** |
| | No-access memory | **PROTECT** |
| Unpipelined two cycle/column | Read/write memory | **SRAM2** or **DRAM2** |
| Unpipelined three cycle/column | Read/write memory | **SRAM3** or **DRAM3** |

Table 14–2. Memory-Type Keywords for Use With the Emulator

| **To identify this kind of emulator memory** | **Use this keyword as the *type* parameter** |
|---|---|
| Read-only memory | **R**, **ROM**, or **READONLY** |
| Write-only memory | **W***,* **WOM***,* or **WRITEONLY** |
| Read/write memory | **WR** or **RAM** |
| No-access memory | **PROTECT** |
| Input port | **IPORT** |
| Output port | **OPORT** |
| Input/output port | **IOPORT** |

❑ The *pagesize* parameter specifies a page boundary for the external memory. You can use this parameter with the simulator only.

| **To select this page boundary (in bytes)** | **Use this as the *pagesize* parameter** |
|---|---|
| 1 to 8† | 0 |
| 2K | 1 |
| 4K | 2 |
| 8K | 3 |
| 16K | 4 |
| 32K | 5 |
| 64K | 6 |
| 128K | 7 |

† To make the page boundary the same as the bus size of the memory access (1, 2, 4, or 8 bytes), use pagesize 0.

By default, the *pagesize* is set to 128K bytes.

❑ The *bussize* parameter specifies the bus size of the memory access. You can use this parameter with the simulator only. Setting the bus size determines the maximum number of bytes that the transfer controller (TC) can transfer during each cycle. You can enter a bus size of 8, 16, 32, or 64 bits. The bus size defaults to 64 bits.

Note that the *pagesize* and *bussize* parameter values are valid only for defining external memory ranges with the simulator. These values are ignored if you're defining an on-chip memory range or if you're using the emulator.

## map

Enable Memory Mapping

**Syntax** **map** {**on** | **off**}

**Menu selection** **M**emory→**E**nable

**Environments** ☑ basic debugger ☐ PDM

**Description** The MAP command enables or disables memory mapping. In some instances, you may want to explicitly enable or disable memory. Note that disabling memory mapping can cause bus fault problems in the target because the debugger may attempt to access nonexistent memory.

## md

Delete Block From Memory Map

**Syntax** **md** *address*

**Menu selection** **M**emory→**D**elete

**Environments** ☑ basic debugger ☐ PDM

**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*]

**Menu selection** none

**Environments** ☑ basic debugger ☐ PDM

**Description** The MEM command identifies a new starting address for the block of memory displayed in the MEMORY window. The optional extension number (#) 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 will be displayed in one of the following formats:

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

## mix

Enter Mixed Mode

**Syntax** **mix**

**Menu selection** Mo**D**e→**M**ixed

**Environments** ☑ basic debugger □ PDM

**Description** The MIX command changes from the current debugging mode to mixed mode. If you're already in mixed mode, the MIX command has no effect.

## ml

List Memory Map

**Syntax** **ml**

**Menu selection** **M**emory→**L**ist

**Environments** ☑ basic debugger □ PDM

**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. If a memory range defines external memory, this command also list the memory type, page size (in bytes), and bus size (in bits) of the range.

## move

Move Active Window

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

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**Description** The MOVE command moves the active window to the specified XY position. If you choose, you can resize the window while you move it (see the SIZE command for valid *width* and *length* values). You can use the MOVE command in one of two ways:

- By supplying a specific *X position* and *Y position*, or
- By omitting the *X position* and *Y position* parameters and using function keys to interactively move the window.

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

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

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

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

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

## mr

Reset Memory Map

**Syntax** **mr**

**Menu selection** **M**emory→**R**eset

**Environments** ☑ basic debugger □ PDM

**Description** The MR command resets the debugger's memory map by deleting all defined memory ranges from the map.

## ms

Save Memory Block to File

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

**Menu selection** **M**emory→**S**ave

**Environments** ☑ basic debugger □ PDM

**Description** The MS command saves the values in a block of memory to a system file; files are saved in COFF format.

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

## next

Single-Step, Next Statement

**Syntax** **next** [*expression*]

**Menu selection** Next=**F10** (in disassembly)

**Environments** ☑ basic debugger □ PDM

**Description** The NEXT command is similar to the STEP command. If you're in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time. Unlike STEP, NEXT never updates the display when executing called functions; NEXT always steps to the next consecutive statement. Unlike STEP, NEXT steps over function calls rather than stepping into them—you don't see the single-step execution of the function call.

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page DB:8-18, discusses this in detail).

## pause

Pause Execution

**Syntax** **pause**

**Menu selection** none

**Environments** ☑ basic debugger ☑ PDM

**Description** The PAUSE command allows you to pause the debugger or parallel debug manager while running a batch file or executing a flow control command. Pausing is especially helpful in debugging the commands in a batch file.

When the debugger or parallel debug manager reads this command in a batch file or during a flow control command segment, the debugger/parallel debug manager stops execution and displays the following message:

```
<< pause – type return >>
```

To continue processing, press ↵.

## pesc

Send ESC Key to Debuggers

**Syntax** **pesc** [**–g** {*group* | *processor name*}]

**Menu selection** none

**Environments** □ basic debugger ☑ PDM

**Description** The PESC command sends the ESC key to an individual debugger or to a group of debuggers. The PESC command halts program execution, but the processors don’t halt at the same real time. When halting a group of processors, the individual processors are halted in the order in which they were added to the group.

The **–g** option identifies the group or processor that the command should be sent to. If you don’t use this option, the ESC key is sent to the default group (dgroup).

## phalt

Halt Processors in Parallel

**Syntax** **phalt** [{**–g** *group* | *processor name*}]

**Menu selection** none

**Environments** □ basic debugger ☑ PDM

**Description** The PHALT command halts one or more processors. If you send a PRUN or PRUNF command to a group or to an individual processor, you can use PHALT to halt the group or the individual processor. Each processor in a group is halted at the same real time. If you don't use the **–g** option to specify a group or a processor name, the PHALT command will be sent to the default group (dgroup).

## prompt

Change Command-Line Prompt

**Syntax** **prompt** *new prompt*

**Menu selection** **C**olor→**P**rompt

**Environments** ☑ basic debugger □ PDM

**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).

## prun — Run Code in Parallel

**Syntax** **prun** [**–r**] [**–g** {*group* **|** *processor name*}]

**Menu selection** none

**Environments** □ basic debugger ☑ PDM

**Description** The PRUN command is the basic command for running an entire program. You enter the command from the command line of the parallel debug manager to begin execution at the same real time for an individual processor or a group of processors.

The **–g** option identifies the group or processor that the command should be sent to. If you don't use this option, then code will run on the default group (dgroup). You can use the PHALT command to stop a global run.

The **–r** (return) option for the PRUN command determines when control returns to the command line of the parallel debug manager:

- **Without –r**, control is not returned to the command line until each debugger in the group finishes running code. If you want to break out of a synchronous command and regain control of the command line, press CONTROL C in the parallel debug manager window. This will return control to the command line. However, no debugger executing the command will be interrupted.
- **With –r**, control is returned to the command line immediately, even if a debugger is still executing a command. You can type new commands, but the processors can't execute the commands until they finish with the current command; however, you can perform PHALT, PESC, and STAT commands when the processors are still executing.

## prunf

Run Free in Parallel **Emulator Only**

**Syntax** **prunf** [**–g** {*group* | *processor name*}]

**Menu selection** none

**Environments** □ basic debugger ☑ PDM

**Description** The PRUNF command starts the processors running freely, which means they are disconnected from the emulator. RUNF synchronizes the debuggers to cause the processors to begin execution at the same real time.

The **–g** option identifies the group or processor that the command should be sent to. If you don't use this option, then code will run on the default group (dgroup).

The PHALT command stops a PRUNF; note that the debugger automatically executes a PHALT when the debugger is invoked.

## pstep

Single-Step in Parallel

**Syntax** **pstep** [**–g** {*group* | *processor name*}] [*count*]

**Menu selection** none

**Environments** □ basic debugger ☑ PDM

**Description** The PSTEP command single-steps synchronously through assembly language code with interrupts disabled. The **–g** option identifies the group or processor that the command should be sent to. If you don't use this option, then code will run on the default group (dgroup). You can use the PHALT command to stop a global run.

You can use the *count* parameter to specify the number of statements that you want to single-step.

**Note:**

If the current statement that a processor is pointing to has a breakpoint, that processor will not step synchronously with the other processors when you use the PSTEP command. However, that processor will still single-step.

## quit

Exit Debugger

**Syntax** **quit**

**Menu selection** none

**Environments** ☑ basic debugger ☑ PDM

**Description** The QUIT command exits the debugger and returns to the operating system. If you enter this command from the parallel debug manager, the parallel debug manager and all debuggers running under the parallel debug manager are exited.

## reload

Reload Object Code

**Syntax** **reload** [*object filename*]

**Menu selection** **L**oad→**R**eload

**Environments** ☑ basic debugger □ PDM

**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.

**Note:**

When you use the RELOAD command in the MP debugger, the MP data cache is first “cleaned.” In other words, all of the modified (dirty) bits are written back to external memory. Then, the instruction and data caches are cleared (all of the DTAG and ITAG registers are cleared).

## reset

Reset Target System

**Syntax** **reset**

**Menu selection** **L**oad→**R**eseT

**Environments** ☑ basic debugger ☐ PDM

**Description** The RESET command resets the target system (emulator only) or the target processor. When you perform a RESET on the MP, an MVP reset is performed. In other words, the MP resets itself and the PPs. When you execute RESET on a PP, the debugger performs a software reset by forcing all registers to their reset values and suspending the current packet request. Note that RESET does not set the PP halt bit in the MP configuration register, so the processor is not halted. As a result, you can resume execution without running MP code to unhalt the PP.

## restart

Reset Program Entry Point

**Syntax** **restart**
**rest**

**Menu selection** **L**oad→R**E**start

**Environments** ☑ basic debugger ☐ PDM

**Description** The RESTART or REST command resets the program to its entry point. (This assumes that you have already used one of the load commands to load a program into memory.)

## return

Return to Function's Caller

**Syntax** **return**
**ret**

**Menu selection** none

**Environments** ☑ basic debugger ☐ PDM

**Description** The RETURN or RET command executes the code in the current C function and halts when execution reaches the caller. Breakpoints do not affect this command, but you can halt execution by pressing the left mouse button or pressing ESC .

## run — Run Code

**Syntax** **run** [*expression*]

**Menu selection** Run=**F5**

**Environments** ☑ basic debugger □ PDM

**Description** The RUN command is the basic command for running an entire program. The *expression* parameter can affect command behavior in these ways:

- If you don’t supply an *expression*, the program executes until it encounters a breakpoint or until you press the left mouse button or press ESC.
- If you supply a logical or relational *expression*, the run becomes conditional (described in detail on page DB:8-18).
- If you supply any other type of *expression*, the debugger treats the expression as a count parameter. The debugger executes *count* instructions, halts, and updates the display.

## runb — Benchmark Code

**Syntax** **runb**

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**Description** The RUNB command executes a specific section of code and counts the number of CPU clock cycles consumed by the execution. In order to operate correctly, **execution must be halted by a software or hardware breakpoint**. After RUNB execution halts, the debugger stores the number of cycles into the CLK pseudoregister. For a complete explanation of the RUNB command and the benchmarking process, see Section 8.7, *Benchmarking*.

For additional information about RUNB, see the *Handling a RUNB emulator resources error* discussion on page DB:A-6.

## runf

Run Free **Emulator Only**

**Syntax** **runf**

**Menu selection** none

**Environments** ☑ basic debugger ☐ PDM

**Description** The RUNF command disconnects the emulator from the target system while code is executing. When you enter RUNF, the debugger clears all breakpoints, disconnects the emulator from the target system, and causes the processor to begin execution at the current IP (for the MP) or current IPE (for the PP). You can quit the debugger, or you can continue to enter commands. However, any command that causes the debugger to access the target at this time produces an error.

The HALT command stops a RUNF; note that the debugger automatically executes a HALT when the debugger is invoked.

## scolor — Change Screen Colors

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

**Menu selection** **C**olor→**C**onfig

**Environments** ☑ basic debugger □ PDM

**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 areas of the display that are affected. The *attributes* identify how the area is affected. The first two *attribute* parameters usually specify the foreground and background colors for the area. If you do not supply a background color, the debugger uses black as the background.

Valid values for the *attribute* parameters include:

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

Valid values for the *area name* parameters include:

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

You don’t have to type an entire *attribute* or *area name*; you need to type only enough letters to uniquely identify the attribute. If you supply ambiguous *attribute* names, the debugger interprets the names in this order: black, blue, bright, blink. If you supply ambiguous *area names*, the debugger interprets them in the order that they’re listed above (left to right, top to bottom).

## sconfig Load Screen Configuration

**Syntax** **sconfig** [*filename*]

**Menu selection** **C**olor→**L**oad

**Environments** ☑ basic debugger □ PDM

**Description** The SCONFIG command restores the display to a specified configuration. This restores the screen colors, window positions, window sizes, and border styles that were saved with the SSAVE command into *filename*. If you don't supply a *filename*, the debugger looks for init.clr. The debugger searches for the specified file in the current directory and then in directories named with the D_DIR environment variable.

## send

Send Debugger Command to Individual Debuggers

**Syntax**

**send** [**–r**] [**–g** {*group* **|** *processor name*}] *debugger command*

**Menu selection**

none

**Environments**

□ basic debugger ☑ PDM

**Description**

The SEND command sends any debugger command to an individual processor or to a group of processors. If the command produces a message, it will be displayed in the COMMAND window for the appropriate debugger(s) and also in the parallel debug manager window.

- The **–g** option specifies the *group* or *processor* that the debugger command should be sent to. If you don't use this option, the command is sent to the default group (dgroup).

- The **–r** (return) option determines when control returns to the command line of the parallel debug manager:

  - **Without –r**, control is not returned to the command line until each debugger in the group finishes running code. Any results that would be printed in the COMMAND window of the individual debuggers will also be echoed in the parallel debug manager command window. These results will be displayed by processor.

    If you want to break out of a synchronous command and regain control of the command line, press CONTROL C in the parallel debug manager window. This will return control to the command line. However, no debugger executing the command will be interrupted.

  - **With –r**, control is returned to the command line immediately, even if a debugger is still executing a command. When you use –r, you **do not** see the results of the commands that the debuggers are executing.

## set — Set a Variable to a String

**Syntax**

**set** [*group name* [**=** *list of processor names*]]
**set** [*variable* [**=** *string value*]]

**Menu selection**

none

**Environments**

□ basic debugger ☑ PDM

**Description**

The SET command allows you to create groups of processors to which you can send commands. With the SET command you can:

- **Define a group of processors.** It is useful to define a group when you plan to send commands to the same set of processors. The commands are sent to the processors in the same order in which you added the processors to the group. To define a group, specify a *group name* and then list the processors you want in the group.

- **Set the default group.** Defining a default group provides you with a shorthand method of maintaining members in a group or of sending commands to the same group. To set up the default group, use the SET command with a special *group name* called dgroup.

- **Modify an existing group or create a group based on another group.** Once you've created a group, you can add processors to it by using the SET command and preceding the existing *group name* with a dollar sign ($) in the list of processors. You can also use a group as part of another group by preceding the existing group's name with a dollar sign. The dollar sign tells the parallel debug manager to use the processors listed previously in the group as part of the new list of processors.

  Suppose GROUPA contains the MP and PP0. If you wanted to add two more PPs to the group, you'd enter:

  ```
  set GROUPA = $GROUPA PP1 PP2 ↵
  ```

  After entering this command, GROUPA would contain MP, PP0, PP1, and PP2.

- **List all groups of processors.** You can use the SET command without any parameters to list all the processors that belong to a group, in the order in which they were added to the group.

You can also use the SET command with system-defined variables to:

- **Change the prompt for the parallel debug manager.** To change the parallel debug manager prompt, use the SET command with the system variable called prompt. For example, to change the parallel debug manager prompt to 3PROCs, enter:

  ```
  set prompt = 3PROCs ↵
  ```

- **Check the execution status of the processors.** In addition to displaying the execution status of a processor or group of processors, the STAT command (described on page DB:14-57) sets a system variable called status. If **all** of the processors in the specified group are running, the status variable is set to 1. If one or more of the processors in the group is halted, the status variable is set to 0.

  You can use this variable when you want an instruction loop to execute until a processor halts (the LOOP/BREAK/CONTINUE/ENDLOOP command sequence is described on page DB:14-32):

- **Create your own system variables.** You can use the SET command to create your own system variables that you can use with PDM commands. For more information about creating your own system variables, see page DB:4-25.

## setf

Set Default Data-Display Format

**Syntax** **setf** [*data type*, *display format* ]

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

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

To return all data types to their default display format, enter:

```
setf * ⏎
```

## spawn — Invoke the MP or PP Debugger

**Syntax**

**spawn** {**mpsim|ppsim**} [**–n** *processor name*] [*options*]
**spawn** {**mpemu|ppemu**} **–n** *processor name* [*options*]

**Menu selection** none

**Environments** □ basic debugger ☑ PDM

**Description**

You must invoke a debugger for each processor that you want the parallel debug manager to control. You can invoke one MP debugger and up to four PP debuggers. Do not attempt to mix the emulator and simulator versions of the debuggers. To invoke a version of the debugger, use the SPAWN command.

- **mpemu, ppemu, mpsim,** and **ppsim** are the executables that invoke the MP or PP version of the debugger. The parallel debug manager associates the *processor name* with the actual processor according to which executable you use. To invoke a debugger, the parallel debug manager must be able to find the executable file for that debugger. The parallel debug manager will first search the current directory and then search the directories listed with the path shell variable.

- **–n** *processor name* supplies a processor name. The parallel debug manager uses processor names to identify the debuggers that are running. The *processor name* can consist of up to eight alphanumeric characters or underscore characters and must begin with an alphabetic character. Note that the name is not case sensitive.

  - **If you're invoking the simulator version of the MP debugger,** you don't need to use the –n option; the *processor name* defaults to MP.

  - **If you're invoking the simulator version of the PP debugger,** the *processor name* that you use *must* end in a 0, 1, 2, or 3 that corresponds to the actual PP in your system. For example, to name the debugger for the PP1, you can use the name PP1. The *processor name* defaults to PP0 if you are invoking a PP version of the debugger and you don't use –n. However, you *must* use the –n option to specify a *processor name* if you want to invoke a debugger for the PP1, PP2, or PP3.

- **If you're invoking the emulator version of either the MP or PP debugger,** the *processor name* that you supply must be a combination of the MVP device name that you defined in your board.dat file and the processor name (MP, PP0, PP1, etc.), separated by an underscore character. (The board.dat file is described in Appendix E, *Describing Your Target System to the Debugger*.) If you're naming a PP, the *processor name* must end in a 0, 1, 2, or 3 that corresponds to the actual PP in your system.

## size — Size Active Window

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

**Menu selection** none

**Environments** ☑ basic debugger ☐ PDM

**Description** The SIZE command changes the size of the active window. You can use the SIZE command in one of two ways:

- ❑ By supplying a specific *width* and *length* or
- ❑ By omitting the *width* and *length* parameters and using function keys to interactively resize the window.

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

If a window is in the middle of the display, you can't size it to the maximum height and width—you can size it only to the right and bottom screen borders. The easiest way to make a window as large as possible is to zoom it, as described on page DB:5-29.

If you enter the SIZE command without *width* and *length* parameters, you can use arrow keys to size the window.

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

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

## sload

Load Symbol Table

**Syntax** **sload** *object filename*

**Menu selection** **L**oad→**S**ymbols

**Environments** ☑ basic debugger □ PDM

**Description** The SLOAD command loads the symbol table of the specified object file. SLOAD is useful in a debugging environment in which the debugger cannot, or need not, load the object code (for example, if the code is in ROM). SLOAD clears the existing symbol table before loading the new one, but does not modify memory or set the program entry point. Note that SLOAD closes the WATCH and DISP windows.

## sound

Enable Error Beeping

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

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**Description** You can cause a beep to sound every time a debugger error message is displayed. This is useful if the COMMAND window is hidden (because you wouldn't see the error message). By default, sound is off.

## ssave

Save Screen Configuration

**Syntax** **ssave** [*filename*]

**Menu selection** **C**olor→**S**ave

**Environments** ☑ basic debugger □ PDM

**Description** The SSAVE command saves the current screen configuration to a file. This saves the screen colors, window positions, window sizes, and border styles. The *filename* parameter names the new screen configuration file. You can include path information (including relative pathnames); if you don't supply path information, the debugger places the file in the current directory. If you don't supply a *filename*, then the debugger saves the current configuration into a file named init.clr and places the file in the current directory.

## stat

Find the Execution Status of Processors

**Syntax** **stat** [{**–g** *group* | *processor name*}]

**Menu selection** none

**Environments** □ basic debugger ☑ PDM

**Description** The STAT command tells you whether a processor is running or halted. If a processor is halted when you execute this command, then the parallel debug manager also lists a PC value. This value is the address of the next instruction to be executed (current IP value for the MP or currrent IPE value for a PP) for that processor. If you don't use the **–g** option, the parallel debug manager displays the status of the processors in the default group (dgroup).

**Note:**

Do not confuse the PC value shown in the parallel debug manager display with the PC registers in either the MP or PP CPU window. The PC value is a parallel debug manager variable that holds the current IP value (for the MP) or the current IPE value (for a PP).

## step

Single-Step

**Syntax** **step** [*expression*]

**Menu selection** Step=**F8** (in disassembly)

**Environments** ☑ basic debugger □ PDM

**Description** The STEP command single-steps through assembly language or C code. If you're in C code, the debugger executes one C statement at a time. In assembly or mixed mode, the debugger executes one assembly language statement at a time.

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

The *expression* parameter specifies the number of statements that you want to single-step. You can also use a conditional *expression* for conditional single-step execution (*Running code conditionally*, page DB:8-18, discusses this in detail).

## system — Enter Operating-System Command

**Syntax** **system** *operating system command*

**Menu selection** none

**Environments** □ basic debugger ☑ PDM

**Description** The SYSTEM command allows you to enter an operating-system command without explicitly exiting the parallel debug manager environment.

## take — Execute Batch File

| **Syntax** | | |
|---|---|---|
| | Basic debugger: | **take** *batch filename* [*, suppress echo flag*] |
| | PDM: | **take** *batch filename* |

**Menu selection** none

**Environments** ☑ basic debugger ☑ PDM

**Description** The TAKE command tells the debugger or the parallel debug manager to read and execute commands from a batch file. The *batch filename* parameter identifies the file that contains commands. If you don't supply a pathname as part of the filename, the parallel debug manager first looks in the current directory and then searches directories named with the D_DIR environment variable.

The *batch filename* for the parallel debug manager version of this command must have a .pdm extension, or the parallel debug manager will not be able to read the file. In addition, the batch file that the parallel debug manager reads can contain only PDM commands.

By default, the debugger echoes the commands to the output area of the COMMAND window and updates the display as it reads the commands from the batch file. For the debugger, you can change this behavior:

- If you don't use the *suppress echo flag* parameter, or if you use it but supply a nonzero value, then the debugger behaves in the default manner.
- If you would like to suppress the echoing and updating, use the value 0 for the *suppress echo flag* parameter.

## unalias

Delete Alias Definition

**Syntax**

**unalias** *alias name*
**unalias** *****

**Menu selection**

none

**Environments**

☑ basic debugger ☑ PDM

**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.

## unset

Delete Group

**Syntax**

**unset** *group name*
**unset** *****

**Menu selection**

none

**Environments**

□ basic debugger ☑ PDM

**Description**

The UNSET command deletes a group of processors. You can use this command in conjunction with the SET command to remove a particular processor from a group. For example, suppose GROUPB contained the MP and the four PPs. If you wanted to remove two of the PPs, you could enter:

```
unset GROUPB ↵
set GROUPB MP PP0 PP1 ↵
```

To delete all groups, enter an asterisk instead of a group name:

```
unset * ↵
```

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

**Note:**

When you use UNSET * to delete all of your system variables and processor groups, variables such as prompt, status, and dgroup are also deleted.

## use

Use New Directory or List Current Source Directories

**Syntax** **use** [*directory name*]

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**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

**Environments** ☑ basic debugger □ PDM

**Description** The VERSION command displays the debugger's copyright date and the current version number of the debugger, silicon, emulator, and simulator.

## wa

Add Item to WATCH Window

**Syntax** **wa** *expression* [,[ *label*] [, *display format*]]

**Menu selection** **W**atch→**A**dd

**Environments** ☑ basic debugger □ PDM

**Description** The WA command displays the value of *expression* in the WATCH window. If the WATCH window isn't open, executing WA opens the WATCH window. The *expression* parameter can be any C expression, including an expression that has side effects.

WA is most useful for watching an expression whose value changes over time; constant expressions serve no useful function in the watch window. The *label* parameter is optional. When used, it provides a label for the watched entry. If you don't use a *label*, the debugger displays the *expression* in the label field.

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

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

If you want to use a *display format* parameter without a *label* parameter, just insert an extra comma. For example:

```
wa pc,,d
```

## wd

Delete Item From WATCH Window

**Syntax** **wd** *index number*

**Menu selection** **W**atch→**D**elete

**Environments** ☑ basic debugger □ PDM

**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.

## whatis

Find Data Type

**Syntax** **whatis** *symbol*

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**Description** The WHATIS command shows the data type of *symbol* in the display area of the COMMAND window. The *symbol* can be any variable (local, global, or static), a function name, structure tag, typedef name, or enumeration constant.

## win

Select Active Window

**Syntax** **win** *WINDOW NAME*

**Menu selection** none

**Environments** ☑ basic debugger □ PDM

**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 of the same types of window are visible on the screen, don't use the WIN command to select one of them. If you supply an ambiguous name (such as C, which could stand for CPU or CALLS), the debugger selects the first window it finds whose name matches the name you supplied. If the debugger doesn't find the window you asked for (because you closed the window or misspelled the name), then the WIN command has no effect.

## wr

Reset WATCH Window

**Syntax** **wr**

**Menu selection** **W**atch→**R**eset

**Environments** ☑ basic debugger ☐ PDM

**Description** The WR command deletes all items from the WATCH window and closes the window.

## zoom

Zoom Active Window

**Syntax** **zoom**

**Menu selection** none

**Environments** ☑ basic debugger ☐ PDM

**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.

# 14.4 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 a window's contents
- ❑ Editing data or selecting the active field

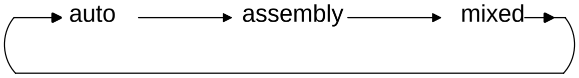
## *Editing text on the command line*

| To do this | Use these function keys |
|---|---|
| Enter the current command (note that if you press the return key in the middle of text, the debugger truncates the input text at the point where you press this key) | ↵ |
| Move back over text without erasing characters | CONTROL H<br>or<br>BACK SPACE |
| Move forward through text without erasing characters | CONTROL L |
| Move back over text while erasing characters | DELETE |
| Move forward through text while erasing characters | SPACE |
| Insert text into the characters that are already on the command line | INSERT |

## Using the command history

| To do this | Use these function keys |
|---|---|
| Repeat the last command that you entered | F2 |
| Move backward, one command at a time, through the command history | TAB |
| Move forward, one command at a time, through the command history | SHIFT TAB |

## Switching modes

| To do this | Use this function key |
|---|---|
| Switch debugging modes in this order:<br> | F3 |

## Halting or escaping from an action

The escape key acts as an end or undo key in several situations.

| To do this | Use this function key |
|---|---|
| ❑ Halt program execution<br>❑ Close a pulldown menu<br>❑ Undo an edit of the active field in a data-display window (pressing this key leaves the field unchanged)<br>❑ Halt the display of a long list of data in the display area of the COMMAND window | ESC |

## *Displaying pulldown menus*

| To do this | Use these function keys |
|---|---|
| Display the Load menu | ALT L |
| Display the Break menu | ALT B |
| Display the Watch menu | ALT W |
| Display the Memory menu | ALT M |
| Display the Color menu | ALT C |
| Display the MoDe menu | ALT D |
| Display an adjacent menu | ← or → |
| Execute any of the choices from a displayed pulldown menu | Press the highlighted letter corresponding to your choice |

## *Running code*

| To do this | Use these function keys |
|---|---|
| Run code from the current IP/IPE (equivalent to the RUN command without an *expression* parameter) | F5 |
| Single-step code from the current IP/IPE (equivalent to the STEP command without an *expression* parameter) | F8 |
| Single-step code from the current IP/IPE; step over function calls (equivalent to the NEXT command without an *expression* parameter) | F10 |

## *Selecting or closing a window*

| To do this | Use these function keys |
|---|---|
| Select the active window (pressing this key makes each window active in turn; stop pressing the key when the desired window becomes active) | F6 |
| Close the CALLS, WATCH, DISP, or additional MEMORY window (the window must be active before you can close it) | F4 |

## *Moving or sizing a window*

You can use the arrow keys to interactively move a window after entering the MOVE or SIZE command without parameters.

| To do this | Use these function keys |
|---|---|
| ❑ Move the window down one line<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 a window's contents*

These descriptions and instructions for scrolling apply to the active window. Some of these descriptions refer to specific windows; if no specific window is named, then the description/ instructions refer to any window that is active.

| To do this | Use these function keys |
|---|---|
| Scroll up through the window contents, one window length at a time | PAGE UP |
| Scroll down through the window contents, one window length at a time | PAGE DOWN |
| Move the field cursor up, one line at a time | ↑ |
| Move the field cursor down, one line at a time | ↓ |
| ❑ *FILE window only:* Scroll left 8 characters at a time | ← |
| ❑ *Other windows:* Move the field cursor left 1 field; at the first field on a line, wrap back to the last fully displayed field on the previous line | |
| ❑ *FILE window only:* Scroll right 8 characters at a time | → |
| ❑ *Other windows:* Move the field cursor right 1 field; at the last field on a line, wrap around to the first field on the next line | |
| *FILE window only:* Adjust the window's contents so that the first line of the text file is at the top of the window | HOME |
| *FILE window only:* Adjust the window's contents so that the last line of the text file is at the bottom of the window | END |
| *DISP windows only*: Scroll up through an array of structures | CONTROL<br>PAGE UP |
| *DISP windows only*: Scroll down through an array of structures | CONTROL<br>PAGE DOWN |

## *Editing data or selecting the active field*

The F9 function key makes the current field (the field that the cursor is pointing to) active. This has various effects, depending on the field.

| To do this | Use these function keys |
|---|---|
| □ *FILE or DISASSEMBLY window:* Set or clear a breakpoint | F9 |
| □ *CALLS window:* Display the source to a listed function | |
| □ *Any data-display window:* Edit the contents of the current field | |
| □ *DISP window:* Open an additional DISP window to display a member that is an array, structure, or pointer | |

Chapter 15

# Basic Information About C Expressions

Many of the debugger commands take C expressions as parameters. This allows the debugger to have a relatively small, yet powerful, instruction set. Because C expressions can have side effects—that is, the evaluation of some types of expressions can affect existing values—you can use the same command to display or to change a value. This reduces the number of commands in the command set.

This chapter contains basic information that you'll need to know in order to use C expressions as debugger command parameters.

**Topics**

## 15.1 C Expressions for Assembly Language Programmers

It’s not necessary for you to be an experienced C programmer to use the debugger. However, in order to use the debugger’s full capabilities, you should be familiar with the rules governing C expressions. You should obtain a copy of *The C Programming Language* (first or second edition) by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey. This book is referred to in the C community, and in Texas Instruments documentation, as **K&R.**

**Note:**

A single value or symbol is a legal C expression.

K&R contains a complete description of C expressions; to get you started, here’s a summary of the operators that you can use in expression parameters.

❑ **Reference operators**

| | | | |
|---|---|---|---|
| –> | indirect structure reference | . | direct structure reference |
| [ ] | array reference | * | indirection (unary) |
| & | address (unary) | | |

❑ **Arithmetic operators**

| | | | |
|---|---|---|---|
| + | addition (binary) | – | subtraction (binary) |
| * | multiplication | / | division |
| % | modulo | – | negation (unary) |
| (*type*) | typecast | | |

❑ **Relational and logical operators**

| | | | |
|---|---|---|---|
| > | greater than | >= | greater than or equal to |
| < | less than | <= | less than or equal to |
| = = | is equal to | != | is not equal to |
| && | logical AND | \|\| | logical OR |
| ! | logical NOT (unary) | | |

### Increment and decrement operators

| | | | |
|---|---|---|---|
| ++ | increment | – – | decrement |

These unary operators can precede or follow a symbol. When the operator precedes a symbol, the symbol value is incremented/decremented before it is used in the expression; when the operator follows a symbol, the symbol value is incremented/decremented after it is used in the expression. Because these operators modify 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 modify a symbol's final value, they have side effects.

# 15.2 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, there are a few limitations, as well as a few additional features not described in K&R C.

## *Restrictions*

The following restrictions apply to the debugger's expression analysis features.

- ❏ The sizeof operator is not supported.
- ❏ The comma operator (,) is not supported (commas are used to separate parameter values for the debugger commands).
- ❏ Function calls and string constants are currently not supported in expressions.
- ❏ The debugger supports a limited capability of type casts; the following forms are allowed:

  **(** *basic type* **)**<br>
  **(** *basic type* ***** *...***)**<br>
  **( [** *structure/union/enum***]** *structure/union/enum tag* **)**<br>
  **( [** *structure/union/enum***]** *structure/union/enum tag* ***** *...* **)**

  Note that you can use up to six *****s in a cast.

## *Additional features*

- All floating-point operations are performed in double precision using standard widening. (This is transparent.) Floats are represented in IEEE floating-point format.
- All registers can be referenced by name.
- Void expressions are legal (treated like integers).
- The specification of variables and functions can be qualified with context information. Local variables (including local statics) can be referenced with the expression form:

  *function name***.***local name*

  This expression format is useful for examining the automatic variables of a function that is not currently being executed. Unless the variable is static, however, the function must be somewhere in the current call stack. Note that if you want to see local variables from the currently executing function, you need not use this form; you can simply specify the variable name (just as in your C source).

  File-scoped variables (such as statics or functions) can be referenced with the following expression form:

  *filename***.***function name*
  or *filename***.***variable name*

  This expression format is useful for accessing a file-scoped static variable (or function) that may share its name with variables in other files.

  Note that in this expression, *filename* **does not include** the file extension; the debugger searches the object symbol table for any source filename that matches the input name, disregarding any extension. Thus, if the variable *ABC* is in file source.c, you can specify it as source.ABC.

  Note that these expression forms can be combined into an expression of the form:

  *filename***.***function name***.***variable name*

- Any integral or void expression can be treated as a pointer and used with the indirection operator (*). Here are several examples of valid use of a pointer in an expression:

  ```
  *123
  *R1
  *(R2 + 123)
  *(I*J)
  ```

  By default, the values are treated as integers (that is, these expressions point to integer values).

❏ Any expression can be typecast to a pointer to a specific type (overriding the default of pointing to an integer, as described above).

**Hint:** You can use casting with the WA and DISP commands to display data in a desired format.

For example, the expression:

```
*(float *)10
```

treats 10 as a pointer to a floating-point value at location 10 in memory. In this case, the debugger fetches the contents of memory location 10 and treats the contents as a floating-point value. If you use this expression as a parameter for the DISP command, the debugger displays memory contents as an array of floating-point values within the DISP window, beginning with memory location 10 as array member [0].

Note how the first expression differs from the expression:

```
(float)*10
```

In this case, the debugger fetches an integer from address 10 and converts the integer to a floating-point value.

You can also typecast to user-defined types such as structures. For example, in the expression:

```
((struct STR *)10)–>field
```

the debugger treats memory location 10 as a pointer to a structure of type STR (assuming that a structure is at address 10) and accesses a field from that structure.

Appendix A

# Data Management and Execution Control

As you are developing code using the MVP debugging tools, you may need additional information about the way processor I/O, memory and cache, breakpoints, and registers are handled. This appendix provides you with more information about these topics.

**Topics**

## A.1 Execution Considerations

This section provides additional information about the following topics:

- Controlling pipeline execution
- Managing interrupts while single-stepping
- Understanding execution differences between RUN and STEP
- Handling a RUNB emulator resources error
- Managing breakpoints
- Modifying the IP/IPE registers

### *Controlling pipeline execution*

The instruction pipeline empties when a software breakpoint is encountered, a PSTEP or single-step command is executed, or you press the ESC key. When the pipeline is emptied, all instructions within the pipeline are executed to completion, any outstanding memory accesses are completed, and any instructions in the floating-point pipeline are completed. There are two major consequences:

- If a CMND instruction that generates a self-interrupt is in the pipeline, the interrupt will occur immediately when execution resumes. Under normal circumstances, the instruction after the CMND instruction will be executed before the interrupt is taken.
- Since all outstanding memory cycles are completed when execution is halted, a scoreboard pipeline stall will never occur in the MP when execution is restarted.

From the parallel debug manager, you can step, run, or halt processors simultaneously. When you run multiple processors in parallel, if you press the ESC key or a breakpoint is executed on one processor, the other processors are not affected. If they are running, they will continue to run.

## *Managing interrupts while single-stepping*

When you enter the STEP or PSTEP command, interrupts that are generated external to the MP or PP are disabled. However, interrupts and traps that are caused by the execution of an instruction (for example, an MP self-interrupt generated by a CMND instruction or the execution of an error trap) are allowed to occur while single-stepping.

Table A–1 lists the MP interrupts and shows which interrupts are internal and which are external.

Table A–1. Master Processor Interrupt Classifications

| **Interrupt** | **Internal or External** |
|---|---|
| Floating-point invalid interrupt | internal |
| Floating-point divide by zero interrupt | internal |
| Integer overflow | internal |
| Floating-point overflow interrupt | internal |
| Floating-point underflow interrupt | internal |
| Floating-point inexact interrupt | internal |
| VC frame timer 0 interrupt when VCOUNT0 = VFTINT0 | external |
| VC frame timer 1 interrupt when VCOUNT1 = VFTINT1 | external |
| MP timer interrupt when TCOUNT decrements to 0 | internal |
| External interrupt 1 | external |
| External interrupt 2 | external |
| Memory fault | |
| ❑ PP instruction-cache errors; PP load/ store/DEA errors; all MP/PP packet request address errors | external |
| ❑ MP instruction-/data-cache errors; MP load/store/DEA errors | internal |
| PP0 message interrupt | external |
| PP1 message interrupt | external |
| PP2 message interrupt | external |
| PP3 message interrupt | external |
| MP message interrupt | internal |
| Packet transfer complete | external |
| Packet transfer busy | external |
| Bad packet transfer (error) | external |

Table A–1. Master Processor Interrupt Classifications (Continued)

| Interrupt | Internal or External |
|---|---|
| External interrupt 3 | external |
| External interrupt 4 | external |
| PP error (illegal instruction) | external |
| Floating-point trap when a floating-point instruction is issued with IE⟦ie⟧ = 0 | internal |
| Error-illegal MP instruction (for example, all 0s or 1s) | internal |
| MP trap instruction | internal |

Table A–2 lists the PP interrupts and shows which interrupts are internal and which are external.

Table A–2. Parallel Processor Interrupt Classifications

| Interrupt | Internal or External |
|---|---|
| Task interrupt | external |
| Packet transfer queued interrupt | external |
| Packet transfer error interrupt | external |
| Bad packet transfer end interrupt | external |
| Message from MP | external |
| Message from PP0 | external† |
| Message from PP1 | external† |
| Message from PP2 | external† |
| Message from PP3 | external† |

† A PP message to itself is an internal interrupt.

## *Understanding execution differences between RUN and STEP*

When you're using single-step execution on a single processor, events caused by running processors can influence the operation of the stepping processor and make it behave differently than if it were executing with the RUN command.

For example, if you execute the code in Example A–1 on the MP with the RUN command to clear a memory fault generated by an illegal PP crossbar memory access, the fault is cleared successfully. However, if you execute the same code with the STEP command while a PP is running, the fault is not cleared. Since the PP and TC are running while the MP is halted between instruction steps, there is sufficient time for the TC to post the fault again. This prevents the memory fault (mf) bit of the MP's INTPEN register from clearing.

Example A–1.   Clear a Memory Fault While Running

```
ld       0x0182000C  (r0), r11     ; Get TC fault status
st       0x0182000C  (r0), r11     ; Clear TC fault status
or       0x80000001, r0, r1        ; Reset PP0
cmnd     r1

rdcr     INTPEN, r1                ; Clear PP mf interrupt
or.tt    0x00004000, r0, r1
wrcr     INTPEN, r1
```

If you swap the code that performs the PP reset with the code that clears the TC fault status register as in Example A–2, the code performs properly under single-step execution.

Example A–2.   Clear a Memory Fault While Stepping

```
or       0x80000001, r0, r1        ; Reset PP0
cmnd     r1
ld       0x0182000C  (r0), r11     ; Get TC fault status
st       0x0182000C  (r0), r11     ; Clear TC fault status


rdcr     INTPEN, r1                ; Clear PP mf interrupt
or.tt    0x00004000, r0, r1
wrcr     INTPEN, r1
```

## Handling a RUNB emulator resources error

If a RUNB command does not have access to the emulator resources necessary to perform an operation, the debugger generates an error, execution does not start, and the debugger does not respond to commands until you execute a HALT command.

## Managing breakpoints

When you set a breakpoint, the opcode in memory is replaced with a special software breakpoint instruction. If the address of the instruction is within the instruction or data cache, the software breakpoint is also written to those locations.

When the debugger encounters a software breakpoint, it halts execution. When you resume execution by using a run or single-step command, the original opcode is loaded back into the instruction cache only, and the instruction is stepped (with interrupts disabled). The breakpoint opcode is then placed back in the instruction cache, and execution is resumed.

## Modifying the IP/IPE registers

When you update the MP's IP register, the PC register is also modified to the next contiguous instruction address. However, if you modify the PC, the IP is not affected.

Likewise, when you update the PP's IPE register, the IPA and PC registers are also modified to the next contiguous instruction addresses. However, if you modify the IPA or PC, the IPE is not affected.

## A.2 Halting Considerations

This section provides additional information about the following topics:

- ❑ Halting a processor while others are running
- ❑ Understanding the difference between debugger and internal halts
- ❑ Modifying register contents when halting

### *Halting a processor while others are running*

You can halt a processor while other processors are running. While a processor is halted, its command bits are latched while the processors that are running execute CMND instructions. The halted processor executes only one command of each type (reset, tsk, msg, etc.) when execution is resumed. This may cause the loss of I/O in the form of CMND instructions. For example, if five message interrupts are issued to a PP that is halted, only one message interrupt will be executed when execution is resumed. To avoid this, use a handshake protocol to wait for a response from a processor that a CMND instruction is issued to. Note that the halt and unhalt CMND instructions are not effected while a processor is halted with the emulator.

### *Understanding the difference between debugger and internal halts*

Each processor (the MP and all PPs) supports a debugger halt and an internal halt. A debugger halt occurs when:

- ❑ You invoke the debugger
- ❑ The debugger encounters a breakpoint
- ❑ You enter a code-execution command such as CSTEP, RUN, etc.
- ❑ You press the ESC key

In general, any time you cause the debugger screen to update, the processor is in the debugger halt state.

If a processor's internal halt is enabled, you can still enter debugger commands. However, these commands do not execute on the processors until you disable the internal halt.

- ❑ To disable the internal halt on the MP, you must manipulate the pins of the device (refer to the *MVP Master Processor User's Guide* for more information).
- ❑ To disable the internal halt on a PP, you must execute the portion of MP code that unhalts the PP.

## *Modifying register contents when halting*

When you use the debugger to halt a processor, the contents of some registers can be modified because the registers latch status signals from other processors or peripherals. The MP registers that can be modified are:

❑ The configuration register (CONFIG)
❑ The interrupt pending register (INTPEN)
❑ The PP error register (PPERROR)
❑ The packet request register (PKTREQ)

The PP registers that can be modified are:

❑ The interrupt pending register (INTFLG)
❑ The communication register (COMM)

## A.3 Data-Management Considerations

This section provides additional information about the following topics:

❑ Managing memory and cache
❑ Detecting memory faults

### *Managing memory and cache*

If you use the MP debugger to read an external memory address, the data cache is always checked for the address. If the address is present, the data is read from the data cache. Otherwise, the data is read from external memory. Data is not read from the instruction cache, because external memory and the instruction cache are assumed to be identical. Note that if the address that is being read is within the data cache, you can't read the corresponding data in external memory without flushing the cache.

If you modify MVP external memory using the MP debugger, data is written to the following locations:

❑ External memory
❑ The data cache if the address is present
❑ The instruction cache if the address is present

When using the PP debugger, data is always read from external memory. Since the instruction cache and external memory are assumed to be identical, data is never read from the instruction cache. During a write to external memory, data is written to external memory and to the instruction cache if the address is present.

### *Detecting memory faults*

If the MP detects a memory fault while the MP's memory fault interrupt vector is pointing to a memory location that also generates a memory fault, the MP's instruction pipeline stalls. If the emulator is running when this occurs, the MP does not respond to the emulator's HALT command (ESC or HALT).

If a PP detects a memory fault, the PP's instruction pipeline stalls and waits for the MP to service the PP. If the emulator is running when this occurs, the PP does not respond to the emulator's HALT command (ESC or HALT).

Appendix B

# Differences Between the Simulator and the Device Specifications

As you are developing code with the current simulator release, you will notice that the simulator is not exactly the same as the MVP device specifications. This appendix explains the differences.

**Topics**

## B.1 Overall Simulation Environment

- ❑ **Timings.** The basic execution pipelines are clock-accurate; however, TC and interrupt operations are not as accurate.
  - ■ While the TC does simulate the time consumed by page-mode RAMs and DRAM refresh, the start-up times for packet requests and cache misses may vary by one or two cycles.
  - ■ Interrupt-related timings may vary one or two cycles in either direction.
- ❑ **Accessing nonexistent registers.** If you attempt to access a nonexistent register, you will not see an error message.
- ❑ **Endian ordering.** The MVP simulator currently supports only big-endian ordering, in which the most significant part of the wide word occupies the first of the consecutive locations.

## B.2 Master Processor

❑ **Floating-point implementation.** The simulator uses the host floating-point implementation, so the exact values returned may differ from those in the real device; however, these cases should be rare because both the simulator and the host use IEEE floating-point conventions. Timings of complicated interactions of multiple floating-point instructions may differ slightly.

❑ **Operating mode.** The MP always operates in privileged mode; the user-mode bit in the ie register is ignored.

❑ **External interrupts.** External interrupts are not yet implemented.

❑ **Command-word data-cache flushes.** Command-word data-cache flushes are not yet implemented. You can achieve the same effect by using a collection of dflush instructions.

❑ **PC/IP register values.** The PC/IP register advance during long-immediates is not quite as specified, and interrupts are not locked out during the annulled cycle of branches. These differences will not be noticeable except when you examine the exact values of EPC and EIP after an interrupted long-immediate or annulled branch (or jump) instruction.

❑ **Bit 1 of the EIP register.** When taking an interrupt, bit 1 of the EIP register is set if the execute-stage instruction is annulled. This bit is not used in the real hardware; the actual device handles this situation in another way.

❑ **Bits in INTPEN register.** In the simulator version of the MP debugger, you can set the bits in the INTPEN register by writing 1s to INTPEN; you can clear the bits by writing 0s to INTPEN. In the emulator version of the MP debugger, you cannot set the bits in the INTPEN register, and you can clear the bits only by writing 1s to INTPEN.

## B.3 Parallel Processors

❑ **Conditional registers.** PP conditional registers invert the low bit. This differs from the actual device in that the low bits are inverted on **any** condition, rather than the restricted form of even register only, condition nonnegative only.

❑ **Illegal operations.** PP illegal operations cause the simulator to abort rather than setting the corresponding bit in the MP's PPERROR register.

## B.4 Transfer Controller

❑ **Priority of requests.** The priority given to requests is the same as that in the device specifications, except that MP cache requests are always given higher priority, as if MP interrupts were disabled.

❑ **Memory faults.** Memory faults cause the simulator to abort rather than perform the actions described in the device specification. Off-chip memory is simulated from 0x0200 0000 to 0xFFFF FFFF, so the only invalid addresses are in the on-chip address space.

❑ **Attempt to fill from crossbar memory.** Cache misses that attempt to fill from the crossbar memory cause the simulator to abort because they are usually caused by an invalid return address or a missing interrupt vector.

## B.5 Video Controller

Only a small part of the video controller is implemented. Only the counters of the frame timers work, and they generate interrupts to the MP.

## B.6 Crossbar Memory

❑ **Contention checking.** There is no contention checking on instruction ports. All competing accesses will complete successfully and correctly, but no processor will stall.

❑ **CONFIG register R bit.** The R (round-robin) bit in the MP CONFIG register is ignored; the crossbar round-robin is always enabled.

Appendix C

# Customizing the Analysis Interface

The interface to the MVP analysis module is register based. In most cases, the Analysis break events dialog box provides a sufficient means of setting hardware breakpoints. In some cases, however, you may want to define more complex conditions for the processor to detect. Or, you may want to write a batch file that defines breakpoint conditions. In either case, you can accomplish these tasks by accessing the analysis registers through the debugger. This appendix explains how to set these registers.

**Topics**

## C.1 Enable Analysis Register

You can enable and disable the analysis module by using the ANAENBL register. Set the bit to 1 to enable and 0 to disable.

Table C–1. ANAENBL Register Bits

| Bit Number | Description |
|---|---|
| 0–1 | enable analysis module |
| 2 | enable automatic re-arm of analysis prior to execute command. (Bit 2 is automatically set when you use the analysis menu to enable analysis.) |
| 3 | reserved (set to 0) |
| 4 | enable EMU0 output |
| 5 | enable EMU1 output |

## C.2 Analysis Configuration Register for the MP

You can select a breakpoint comparator configuration, bus selection, and access qualifications for the MP by using the MP ANACONFIG register. Set the bit to 1 to enable or to 0 to disable, or set as indicated to enable the desired settings.

**Note:**

If you select the instruction bus with write-only qualification, a breakpoint never occurs.

Table C–2. MP ANACONFIG Register Bits

(a) Breakpoint Event Enable Bits

| Bit Number | Description |
|---|---|
| 0 | reserved (set to 0) |
| 1 | reserved (set to 0) |
| 2 | enable breakpoint 1 |
| 3 | enable breakpoint 2 |
| 4 | enable range breakpoints |
| 5 | enable EMU0 driven low |
| 6 | enable EMU1 driven low |
| 7–23 | reserved |

Table C–2. MP ANACONFIG Register Bits (Continued)

(b) Comparator Configuration Bits

| Bit Number | Description |
|---|---|
| 24 | Comparator 1 bus selection:<br>0 = program bus<br>1 = data bus |
| 26,25 | Comparator 1 bus qualification:<br>00 = read or write<br>01 = read only<br>10 = write only<br>11 = invalid |
| 27 | Comparator 2 bus selection:<br>0 = program bus<br>1 = data bus |
| 29,28 | Comparator 2 bus qualification:<br>00 = read or write<br>01 = read only<br>10 = write only<br>11 = invalid |
| 30 | Breakpoint type:<br>0 = single point<br>1 = range |
| 31 | Range type:<br>0 = inclusive range<br>1 = exclusive range |

## C.3 Analysis Configuration Register for the PP

You can select a breakpoint comparator configuration, bus selection, and access qualifications for the PP by using the PP ANACONFIG register. Set the bit as indicated to enable the desired settings.

**Note:**

If you select the instruction bus with write-only qualification, a breakpoint never occurs.

Table C–3. PP ANACONFIG Register Bits

(a) Breakpoint Event Enable Bits

| Bit Number | Description |
|---|---|
| 0 | reserved (set to 0) |
| 1 | reserved (set to 0) |
| 2 | enable breakpoint 1 |
| 3 | enable breakpoint 2 |
| 4 | enable range of breakpoints |
| 5 | enable EMU0 driven low |
| 6 | enable EMU1 driven low |
| 7–22 | reserved |

Table C–3. PP ANACONFIG Register Bits (Continued)

(b) Comparator Configuration Bits

| Bit Number | Description |
|---|---|
| 23,22 | Comparator 1 bus selection:<br>00 = program bus<br>01 = global bus<br>10 = local bus<br>11 = invalid |
| 25,24 | Comparator 1 bus qualification:<br>00 = read or write<br>01 = read only<br>10 = write only<br>11 = invalid |
| 27,26 | Comparator 2 bus selection:<br>00 = program bus<br>01 = global bus<br>10 = local bus<br>11 = invalid |
| 29,28 | Comparator 2 bus qualification:<br>00 = read or write<br>01 = read only<br>10 = write only<br>11 = invalid |
| 30 | Breakpoint type:<br>0 = single point<br>1 = range |
| 31 | Range type:<br>0 = inclusive range<br>1 = exclusive range |

## C.4 Analysis Status Register

The ANASTATUS register records the occurrence of events that are enabled with bits 2–6 of the ANACONFIG register. During a run command, if an event is taken, that event’s bit in the status register is cleared. In other words, the exclusive OR of bits 2–6 of the ANACONFIG and ANASTATUS registers indicates that an event has occurred.

Table C–4. ANASTATUS Register Bits

| Bit Number | Description |
|---|---|
| 0 | reserved (set to 0) |
| 1 | reserved (set to 0) |
| 2 | breakpoint 1 |
| 3 | breakpoint 2 |
| 4 | range of breakpoints |
| 5 | EMU0 driven low |
| 6 | EMU1 driven low |

## C.5 Breakpoint Address Registers

Use the brk1 and brk2 registers to set 32-bit hardware breakpoint addresses.

Example C–1. The brk1 and brk2 Registers

```
>?brk1 = 0x00230020
>?brk2 = 0x03000040
```

# C.6 Analysis Programming Example

To program the analysis module for a single breakpoint, type the following commands on the command line.

Example C–2. Analysis Program for a Single Breakpoint

```
>?anaenbl = 1
>?anaconfig = 4
>?anastatus(should report back 4)
>?brk1 = 0x02000020 (or any address in your program)
>run(processor will halt when breakpoint taken)
>?anastatus(should report back 0 – bit 2 cleared)
```

Appendix D

# What the Debugger Does During Invocation

In some circumstances, you may find it helpful to know the steps that the debugger goes through during the invocation process. These are the steps, in order, that the debugger performs. Note that the PDM executes the first step. (For more information on the environment variables mentioned below, refer to your installation guide.)

1) Establishes the connection between the processor name that you provide and the actual processor. The processor is halted at this time.

2) Reads options from the command line.

3) Reads any information specified with the D_OPTIONS environment variable.

4) Reads information from the D_DIR, D_SRC, DISPLAY (if present), and LD_LIBRARY_PATH environment variables.

5) Looks for the init.clr screen configuration file.

   (The debugger searches for the screen configuration file in directories named with D_DIR.)

6) Initializes the debugger screen and windows but initially displays only the COMMAND window.

7) 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:

   a) When you invoke the debugger, it checks to see if you've used the –t debugger option. If it finds the –t option, the debugger reads and executes the specified file.

   b) If you don't use the –t option, the debugger looks for the default initialization batch file called *init.cmd*. If the debugger finds the file, it reads and executes the file.

8) Loads any object filenames specified with D_OPTIONS or specified on the command line during invocation.

9) Determines the initial mode (auto, assembly, or mixed) and displays the appropriate windows on the screen.

At this point, the debugger is ready to process any commands that you enter.

Appendix E

# Describing Your Target System to the Debugger

In order for the debugger to understand how you have configured your target system, you must supply a file for the debugger to read.

❑ If you're using an emulation scan path that contains only one MVP and no other devices, you can use the *board.dat* file that comes with the MVP emulator kit. This file describes to the debugger the single MVP in the scan path and gives the MVP the name MVP1. Since the debugger automatically looks for a file called board.dat in the current directory and in the directories specified with the D_DIR environment variable, you can skip this appendix.

❑ If you plan to use a different target system, you must follow these steps:

**Step 1:** Create the board configuration text file.

**Step 2:** Translate the board configuration text file to a binary, structured format so that the debugger can read it.

**Step 3:** Specify the formatted configuration file when invoking the debugger.

These steps are described in this appendix.

**Topics**

# E.1 Step 1: Create the Board Configuration Text File

To describe the emulation scan path of your target system to the debugger, you must create a board configuration file. The file consists of a series of entries, each describing one device on your scan path. You must list, in order, the individual devices on your system in the board configuration file for the debugger to work. The text version of the configuration file will be referred to as *board.cfg* in this appendix.

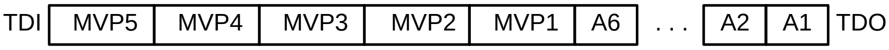
Example E–1 shows a board.cfg file that describes a possible MVP device chain. It lists six octals named A1–A6, followed by five MVP devices named MVP1, MVP2, MVP3, MVP4, and MVP5.

Example E–1. A Sample MVP Device Chain

(a) A sample board.cfg file

| **Device Name** | **Device Type** | **Comments** |
|---|---|---|
| `″A1″` | `BYPASS08` | `;the first device nearest TDO`<br>`;(test data out)` |
| `″A2″` | `BYPASS08` | `;the next device nearest TDO` |
| `″A3″` | `BYPASS08` | |
| `″A4″` | `BYPASS08` | |
| `″A5″` | `BYPASS08` | |
| `″A6″` | `BYPASS08` | |
| `″MVP1″` | `TI320C8x` | `;the first MVP` |
| `″MVP2″` | `TI320C8x` | |
| `″MVP3″` | `TI320C8x` | |
| `″MVP4″` | `TI320C8x` | |
| `″MVP5″` | `TI320C8x` | `;the last MVP nearest TDI`<br>`;(test data in)` |

(b) A sample MVP device chain



The order in which you list each device is important. The emulator scans the devices, assuming that the data from one device is followed by the data of the next device on the chain. Data from the device that is closest to the emulation header’s TDO reaches the emulator first. Moreover, in the board.cfg file, the devices should be listed in the order in which their data reaches the emulator. The device whose data reaches the emulator first is listed first in the board.cfg file; the device whose data reaches the emulator last is listed last in the board.cfg file.

The board.cfg file can have any number of each of the three types of entries:

- **Debugger devices** such as the MVP. These are the only devices that the debugger can recognize.
- The **TI ACT8997 scan path linker**, or **SPL**. The SPL allows you to have up to four secondary scan paths that can each contain debugger devices (MVPs) and other devices.
- **Other devices**. These are any other devices in the scan path. For example, you can have devices such as the TI BCT 8244 octals. These devices cannot be debugged and must be worked around or “bypassed” when trying to access the MVPs.

Each entry in the board.cfg file consists of at least two pieces of data:

- **The name of the device.** The device name always appears first and is enclosed in double quotes:

  *”device name”*

  This is the same name that you use with the –n debugger option, which tells the debugger the name of the MVP. The *device name* can consist of up to eight alphanumeric characters or underscore characters and must begin with an alphabetic character.

- **The type of the device.** The debugger supports the following device types:

  - **TI320C8x** is an example of a debugger device type. TI320C8x describes the MVP. TI320C4x describes the ’C4x.
  - **SPL** specifies the scan path linker and must be followed by four subpaths, as in this syntax:

    ”*device name*” **SPL** {*spath0*} {*spath1*} {*spath2*} {*spath3*}

    Each *spath* (subpath) can contain any number of devices. However, an SPL subpath **cannot** contain another SPL.

- **BYPASS##** describes devices other than the debugger devices or SPL. The ## is the hexadecimal number that describes the number of bits in the device’s JTAG instruction register. For example, TI BCT 8244 octals have a device type of BYPASS08.

Example E–2 shows a file that contains an SPL.

Example E–2. A board.cfg File Containing an SPL

```
Device
Name       Device Type  Comments
″A1″       BYPASS08     ;the first device nearest TDO
″A2″       BYPASS08
″MVP1″     TI320C8x     ;the first MVP
″HUB″      SPL          ;the scan path linker
  {                     ;first subpath
  ″B1″     BYPASS08
  ″B2″     BYPASS08
  ″MVP2″   TI320C8x     ;the second MVP
  }
  {                     ;second subpath
  ″C1″     BYPASS08
  ″C2″     BYPASS08
  ″MVP3″   TI320C8x     ;the third MVP
  }
  {                     ;third subpath (contains nothing)
  }
  {                     ;fourth subpath
  ″D1″     BYPASS08
  ″D2″     BYPASS08
  ″MVP4″   TI320C8x     ;the fourth MVP
  }
″MVP5″     TI320C8x     ;the last MVP nearest TDI
```

**Note:** The indentation in the file is for readability only.

## E.2 Step 2: Translate the Configuration File to a Debugger-Readable Format

After you have created the board.cfg file, you must translate it from text to a binary, conditioned format so that the debugger can understand it. To translate the file, use the composer utility that is included with the emulator kit. At the system prompt, enter the following command:

**composer** [*input file* [*output file*] ]

- The *input file* is the name of the board.cfg file that you created in step 1; if the file isn't in the current directory, you must supply the entire pathname. If you omit the input filename, the composer utility looks for a file called board.cfg in your current directory.
- The *output file* is the name that you can specify for the resulting binary file; ideally, use the name board.dat. If you want the output file to reside in a directory other than the current directory, you must supply the entire pathname. If you omit an output filename, the composer utility creates a file called board.dat and places it in the current directory.

To avoid confusion, use a .cfg extension for your text filenames and a .dat extension for your binary filenames. If you enter only one filename on the command line, the composer utility assumes that it is an input filename.

## E.3 Step 3: Specify the Configuration File When Invoking the Debugger

When you invoke a debugger (either from the PDM or at the system prompt), the debugger must be able to find the board.dat file so that it knows how you have set up your scan path. The debugger looks for the board.dat file in the current directory and in the directories named with the D_DIR environment variable.

If you used a name other than board.dat or if the board.dat file is not in the current directory nor in a directory named with D_DIR, you must use the –f option when you invoke the debugger. The –f option allows you to specify a board configuration file (and pathname) that will be used instead of board.dat. The format for this option is:

**–f** *filename*

Appendix F

# Debugger and Parallel Debug Manager 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.

**Topics**

## F.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.

## F.2 Alphabetical Summary of Debugger Messages

### Symbols

**‘]’ expected**

*Description* This is an expression error—it means that the parameter contained an opening bracket symbol “**[**” but didn’t contain a closing bracket symbol “**]**”.

*Action* See Section F.4.

**‘)’ expected**

*Description* This is an expression error—it means that the parameter contained an opening parenthesis symbol “**(**” but didn’t contain a closing parenthesis symbol “**)**”.

*Action* See Section F.4.

### 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.

**Attempt to fill cache from crossbar memory**

*Description* The PC has been loaded with a crossbar memory address and a cache miss operation was attempted.

**Attempt to use reserved FMOD code (14 or 15)**

*Description* (PPs only) FMOD codes 14 and 15 are reserved.

*Action* Modify your code.

## B

**Breakpoint already exists at** ***address***

*Description* During single-step execution, the debugger attempted to set a breakpoint where one already existed. (This isn’t necessarily a breakpoint that you set—it may have been an internal breakpoint that was used for single-stepping.)

*Action* None should be required; you may want to reset the program entry point (RESTART) and re-enter the single-step command.

**Breakpoint table full**

*Description* 200 breakpoints are already set, and there was an attempt to set another. The maximum limit of 200 breakpoints includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.

*Action* Enter a BL command to see where breakpoints are set in your program. Use the BR command to delete all software breakpoints, or use the BD command to delete individual software breakpoints.

### 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 barrel-shift a Non-D register

*Description* (PPs only) The input to the barrel rotator must be a D register.

*Action* Modify your code.

### Cannot detect target power

*Description* This hardware error occurs after resetting the emurst command. Follow the steps described below and then restart your emulator.

*Action*

- ❑ Check the emulator board to be sure it is installed snugly.
- ❑ Check the cable connecting your emulator and target system to be sure it is not loose.
- ❑ Check your target board to be sure it is getting the correct voltage.
- ❑ Check your emulator scan path to be sure it is uninterrupted.
- ❑ Ensure that your port address is set correctly:
  - ■ Check to be sure the –p option used with the D_OPTIONS environment variable matches the I/O address defined by your switch settings. (Refer to the *TMS320C8x Workstation Emulator Installation Guide* for more information.)
  - ■ Check to see if you have a conflict in address space with another bus setting. If you have a conflict, change the switches on your board to one of the alternate settings listed in the installation guide. Modify the –p option of the D_OPTIONS environment variable to reflect the change in your switch settings.

### Cannot edit field

*Description* Expressions that are displayed in the WATCH window cannot be edited.

*Action* If you attempted to edit an expression in the WATCH window, you may have actually wanted to change the value of a symbol or register used in the expression. Use the ? or EVAL command to edit the actual symbol or register. The expression value will automatically be updated.

### Cannot find/open initialization file

*Description* The debugger can't find the init.cmd file.

*Action* Be sure that init.cmd is in the appropriate directory. If it isn't, copy it from the debugger product tape. If the file is already in the correct directory, verify that the D_DIR environment variable is set up to identify the directory. See *Setting Up the Debugger Environment* in the installation guide.

### Cannot halt the processor

*Description* This is a fatal error—for some reason, pressing ESC didn't halt program execution.

*Action* Exit the debugger. Invoke the .cshrc file; then invoke the debugger again.

### Cannot initialize target system

*Description* This error occurs while you are invoking the debugger with the emulator. Any combination of events may cause this error to occur.

*Action*

- Check the cable connecting the emulator to the target system to be sure it is not loose.
- Ensure that your port address is set correctly:
  - Check to be sure the –p option used with the D_OPTIONS environment variable matches the I/O address defined by your switch settings.
  - Check to see if you have a conflict in address space with another bus setting. If you have a conflict, change the switches on your board to one of the alternate settings listed in the installation guide. Modify the –p option of the D_OPTIONS environment variable to reflect the change in your switch settings.
- Check the end of your .cshrc file for the emurst.exe command. Execute this command *after* powering up the target board.

For more details, refer to the *TMS320C8x Workstation Emulator Installation Guide.*

### Cannot map into reserved memory: ?

*Description* The debugger tried to access unconfigured/reserved/nonexistent memory.

*Action* Remap the reserved memory accesses.

### Cannot open config file

*Description* The SCONFIG command can’t find the screen-customization file that you specified.

*Action* Be sure that the filename was typed correctly. If it wasn’t, re-enter the command with the correct name. If it was, re-enter the command and specify full path information with the filename.

### Cannot open “*filename*”

*Description* The debugger attempted to show *filename* in the FILE window but could not find the file.

*Action* Be sure that the file exists as named. If it does, enter the USE command to identify the file’s directory.

### Cannot open object file: “*filename*”

*Description* The file specified with the LOAD, SLOAD, or RELOAD command is not an object file that the debugger can load.

*Action* Be sure that you’re loading an actual object file. Be sure that the file was linked.

### Cannot open new window

*Description* A maximum of 127 windows can be open at once. The last request to open a window would have made 128, which isn’t possible.

*Action* Close any unnecessary windows. Windows that can be closed include WATCH, CALLS, DISP, and additional MEMORY windows. To close any of these windows, make the desired window active and press F4.

### Cannot read processor status

*Description* This is a fatal error—for some reason, pressing ESC didn’t halt program execution.

*Action* Exit the debugger. Invoke the .cshrc, then invoke the debugger again. If you are using the emulator, check the cable connections, also.

### Cannot reset the processor

*Description* This is a fatal error—for some reason, pressing ESC didn’t halt program execution.

*Action* Exit the debugger. Invoke the .cshrc file, then invoke the debugger again. If you are using the emulator, there may be a problem with the target system; check the cable connections.

### Cannot restart processor

*Description* If a program doesn't have an entry point, then RESTART won't reset the IP/IPE to the program entry point.

*Action* Don't use RESTART if your program doesn't have an explicit entry point.

### Cannot set/verify breakpoint at *address*

*Description* Either you attempted to set a breakpoint in read-only or protected memory, or there are hardware problems with the target system. This may also happen when you enable or disable on-chip memory while using breakpoints.

*Action* Check your memory map. If the address that you wanted to breakpoint wasn't in ROM, see Section F.5.

### Cannot step

*Description* There is a problem with the target system.

*Action* See Section F.5.

### 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 F.4.

### Command "*cmd*" not found

*Description* The debugger didn't recognize the command that you typed.

*Action* Re-enter the correct command. For a list of valid debugger commands, see Chapter 14, *Summary of Commands and Special Keys*.

### Command timed out, emulator busy

*Description* There is a problem with the target system.

*Action* See Section F.5.

### 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 re-enter the MA command. If the existing block is necessary, re-enter the MA command with parameters that will not overlap the existing block.

### Corrupt call stack

*Description* The debugger tried to update the CALLS window and couldn’t. This may be because a function was called that didn’t return. Or it could be that the program stack was overwritten in target memory. Another reason you may have this message is that you are debugging code that has optimization enabled (for example, you did not use the –g compile switch); if this is the case, ignore this message—code execution is not affected.

*Action* If your program called a function that didn’t return, then this is normal behavior (as long as you intended for the function not to return). Otherwise, you may be overwriting program memory.

## E

### Emulator I/O address is invalid

*Description* The debugger was invoked with the –p option, and an invalid *port address* was used.

*Action* For valid *port address* values, refer to the *TMS320C8x Workstation Emulator Installation Guide*.

### Error in expression

*Description* This is an expression error.

*Action* See Section F.4.

### Execution error

*Description* There is a problem with the target system.

*Action* See Section F.5.

## F

### File not found

*Description* The filename specified for the FILE command was not found in the current directory or any of the directories identified with D_SRC.

*Action* Be sure that the filename was typed correctly. If it was, re-enter the FILE command and specify full path information with the filename.

### File not found : “*filename*”

*Description* The filename specified for the LOAD, RELOAD, SLOAD, or TAKE command was not found in the current directory or any of the directories identified with D_SRC.

*Action* Be sure that the filename was typed correctly. If it was, re-enter the command and specify full path information with the filename.

**File too large (*filename*)**

*Description* You attempted to load a file that was more than 65,518 bytes long.

*Action* Try loading the file without the symbol table (SLOAD), or relink the program with fewer modules.

**Float not allowed**

*Description* This is an expression error—a floating-point value was used incorrectly.

*Action* See Section F.4.

**Function required**

*Description* The parameter for the FUNC command must be the name of a function in the program that is loaded.

*Action* Re-enter the FUNC command with a valid function name.

## I

**Illegal addressing mode**

*Description* An illegal MVP addressing mode was encountered.

*Action* Refer to Section 5.6, *Accessing Illegal Addresses*, in the *MVP Master Processor User's Guide* and to Section 2.4, *Access Capabilities,* in the *MVP Transfer Controller User's Guide* for valid addressing modes.

**Illegal cast**

*Description* This is an expression error—the expression parameter uses a cast that doesn't meet the C language rules for casts.

*Action* See Section F.4.

**Illegal control transfer instruction**

*Description* The instruction following a delayed branch/call instruction was modifying the program counter.

*Action* Modify your source code.

### Illegal left hand side of assignment

*Description* This is an expression error—the left-hand side of an assignment expression doesn't meet C language assignment rules.

*Action* See Section F.4.

### Illegal loop counter combination

*Description* (PPs only) The LCTL register has been loaded with an invalid loop counter assigned in one of the LE0, LE1, or LE2 fields.

*Action* Modify your code.

### Illegal memory access

*Description* Your program tried to access unmapped memory.

*Action* Modify your source code.

### Illegal opcode

*Description* An invalid MVP instruction was encountered.

*Action* Modify your source code.

### Illegal operand of &

*Description* This is an expression error—the expression attempts to take the address of an item that doesn't have an address.

*Action* See Section F.4.

### Illegal operation

*Description* Your code contained an illegal instruction or attempted an illegal branch.

*Action* Modify your code.

### Illegal pointer math

*Description* This is an expression error—some types of pointer math are not valid in C expressions.

*Action* See Section F.4.

### 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 F.4.

### Illegal structure reference

*Description* This is an expression error—either the item being referenced as a structure is not a structure, or you are attempting to reference a nonexistent portion of a structure.

*Action* See Section F.4.

### Illegal use of structures

*Description* This is an expression error—the expression parameter is not using structures according to the C language rules.

*Action* See Section F.4.

### Illegal use of void expression

*Description* This is an expression error—the expression parameter does not meet the C language rules.

*Action* See Section F.4.

### Integer not allowed

*Description* This is an expression error—the command did not accept an integer as a parameter.

*Action* See Section F.4.

### Internal access using VRAM or transparency mode

*Description* (TC only) Attempt to access on-chip memory using VRAM or transparency mode.

*Action* Modify your code.

### 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* Re-enter the command with valid parameters. Refer to the appropriate command description in Chapter 14, *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* Re-enter the COLOR or SCOLOR command with a valid *area name* parameter. Valid area names are listed in Table 12–2, *Summary of Area Names for the COLOR and SCOLOR Commands*.

### 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* Re-enter the COLOR or SCOLOR command with a valid color parameter. Valid color attributes are listed in Table 12–1, *Colors and Other Attributes for the COLOR and SCOLOR Commands*.

### Invalid crossbar address

*Description* A processor or the TC attempted to access an invalid crossbar address.

*Action* Modify your code.

### Invalid data size in Move instruction

*Description* (PPs only) Attempt to execute a Move instruction with the size field set to reserved code.

*Action* Modify your code.

### 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* Re-enter the MA command. Use one of the following valid parameters to identify the memory type:

| | |
|---|---|
| SRAM0, DRAM0 | (pipelined one cycle/column, read/write memory) |
| R, ROM, READONLY | (unpipelined one cycle/column, read-only memory) |
| W, WOM, WRITEONLY | (unpipelined one cycle/column, write-only memory) |
| SRAM, DRAM, SRAM1, DRAM1, WR, RAM | (unpipelined one cycle/column, read/write memory) |
| PROTECT | (unpipelined one cycle/column, no-access memory) |
| SRAM2, DRAM2 | (unpipelined two cycle/column, read/write memory) |
| SRAM3, DRAM3 | (unpipelined three cycle/column, read/write memory) |

### Invalid msize in mf_mask

*Description* (PPs only) You attempted to use an invalid msize to select the MF bits in a C-port mask operation. The msize codes can be only one of the following: 8, 16, or 32.

*Action* Modify your code.

### Invalid object file

*Description* Either the file specified with the LOAD, SLOAD, or RELOAD command is not an object file that the debugger can load, or it has been corrupted.

*Action* Be sure that you’re loading an actual object file. Be sure that the file was linked. If the file you attempted to load was a valid executable object file, then it was probably corrupted.

### Invalid packet-request options

*Description* (TC only) A packet request was programmed illegally.

*Action* Modify your code.

### Invalid watch delete

*Description* The debugger can’t delete the parameter supplied with the WD command. Usually, this is because the watch index doesn’t exist or because a symbol name was typed in instead of a watch index.

*Action* Re-enter the WD command. Be sure to specify the watch index that matches the item you’d like to delete (this is the number in the left column of the WATCH window). Remember, you can’t delete items symbolically—you must delete them by number.

### Invalid window position

*Description* The debugger can’t move the active window to the XY position specified with the MOVE command. Either the XY parameters are not within the screen limits, or the active window may be too large to move to the desired position.

*Action*

- You can use the mouse to move the window.
- If you don’t have a mouse, enter the MOVE command without parameters; then use the arrow keys to move the window. When you’re finished, you *must* press ESC or ↵.
- If you prefer to use the MOVE command with parameters, the minimum XY position is 0,1; the maximum position depends on which screen size you’re using.

### Invalid window size

*Description* The width and length specified with the SIZE or MOVE command may be too large or too small. If valid width and length were specified, then the active window is already at the far right or bottom of the screen and so cannot be made larger.

*Action*

- □ You can use the mouse to size the window.
- □ If you don't have a mouse, enter the SIZE command without parameters; then use the arrow keys to move the window. When you're finished, you *must* press ESC or 🖱.
- □ If you prefer to use the SIZE command with parameters, the minimum size is 4 by 3; the maximum size depends on which screen size you're using.

## L

### Load aborted

*Description* This message always follows another message.

*Action* Refer to the message that preceded *Load aborted*.

### Lost power (or cable disconnected)

*Description* Either the target cable is disconnected, or the target system is faulty.

*Action* Check the target cable connections. If the target seems to be connected correctly, see Section F.5.

### Lost processor clock

*Description* Either the target cable is disconnected, or the target system is faulty.

*Action* Check the target cable connections. If the target seems to be connected correctly, see Section F.5.

### Lval required

*Description* This is an expression error—an assignment expression was entered that requires a legal lefthand side.

*Action* See Section F.4.

## M

**Memory access error at** ***address***

*Description* Either the processor is receiving a bus fault, or there are problems with target system memory.

*Action* See Section F.5.

**Memory map table full**

*Description* Too many blocks have been added to the memory map. This will rarely happen unless blocks are added word by word (which is inadvisable).

*Action* Stop adding blocks to the memory map. Consolidate any adjacent blocks that have the same memory attributes.

**Message dflush not yet implemented**

*Description* This action is not yet implemented.

**Multiple-ALU error: asize is not 4, 5, or 6**

*Description* (PPs only) The asize codes can be only one of the following: 4, 5, or 6.

*Action* Modify your code.

## N

**Name “*name*” not found**

*Description* The command cannot find the object named *name*.

*Action*

❑ If *name* is a symbol, be sure that it was typed correctly. If it wasn’t, re-enter the command with the correct name. If it was, then be sure that the associated object file is loaded.

❑ If *name* was some other type of parameter, refer to the command’s description for a list of valid parameters.

## O

### On-chip access using SRT-mode packet request

*Description* (TC only) Illegal on-chip access.

*Action* Modify your code.

## P

### Packet request guide table not properly aligned

*Description* (TC only) Packet-request guide table entries must be double-word aligned.

*Action* Modify your code.

### Packet request started at an address not 64-byte aligned

*Description* Packet requests must start at addresses that are 64-byte aligned.

*Action* Modify your code.

### Pointer not allowed

*Description* This is an expression error.

*Action* See Section F.4.

## R

### Register access error

*Description* Either the processor is receiving a bus fault, or there are problems with target-system memory.

*Action* See Section F.5.

### Register must be even for 64-bit load

*Description* (MP only) Attempt to use an odd register for a double-word load instruction.

*Action* Modify your code.

### Register must be even for 64-bit store

*Description* (MP only) Attempt to use an odd register for a double-word store instruction.

*Action* Modify your code.

## S

### Shift rotate register value must be even when C=0 and emask=0

*Description* This is a restriction of the MP Shift instruction.

*Action* Modify your code.

### Source and destination counts do not match

*Description* (TC only) The destination and source lengths are different.

*Action* Modify your code.

### 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, it is possible to specify only the first address of a defined block.

### Structure member name required

*Description* This is an expression error—a symbol name followed by a period but no member name.

*Action* See Section F.4.

### Structure member not found

*Description* This is an expression error—an expression references a nonexistent structure member.

*Action* See Section F.4.

### Structure not allowed

*Description* This is an expression error—the expression is attempting an operation that cannot be performed on a structure.

*Action* See Section F.4.

### Take file stack too deep

*Description* Batch files can be nested up to 10 levels deep. Batch files can call other batch files, which can call other batch files, and so on. Apparently, the batch file that you are TAKEing calls batch files that are nested more than 10 levels.

*Action* Edit the batch file that caused the error. Instead of calling another batch file from within the offending file, you may want to copy the contents of the second file into the first. This will remove a level of nesting.

### Too many breakpoints

*Description* 200 breakpoints are already set, and there was an attempt to set another. Note that the maximum limit of 200 breakpoints includes internal breakpoints that the debugger may set for single-stepping. Under normal conditions, this should not be a problem; it is rarely necessary to set this many breakpoints.

*Action* Enter a BL command to see where breakpoints are set in your program. Use the BR command to delete all software breakpoints or use the BD command to delete individual software breakpoints.

### Too many paths

*Description* More than 20 paths have been specified cumulatively with the USE command, D_SRC environment variable, and –i debugger option.

*Action* Don’t enter the USE command before entering another command that has a *filename* parameter. Instead, enter the second command and specify full path information for the *filename*.

## U

### 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* Re-enter 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:

- **CA**LLS
- **CP**U
- **DISP**
- **CO**MMAND
- **DISA**SSEMBLY
- **F**ILE
- **M**EMORY
- **W**ATCH

## F.3 Alphabetical Summary of Parallel Debug Manager Messages

This section contains an alphabetical listing of the error messages that the parallel debug manager might display. Each message contains both a description of the situation that causes the message and an action to take.

**Note:**

If errors are detected in a TAKE file, the parallel debug manager aborts the batch file execution, and the file line number of the invalid command is displayed along with the error message.

### C

**Cannot communicate with** *“name”*

*Description* The parallel debug manager cannot communicate with the named debugger, because the debugger either crashed or was exited.

*Action* Spawn the debugger again.

**Cannot communicate with the child debugger**

*Description* This error occurs when you are spawning a debugger. The parallel debug manager was able to find the debugger executable file, but the debugger could not be invoked for some reason, and the communication between the debugger and the parallel debug manager was never established. This usually occurs when you have a problem with your target system.

*Action* Exit the parallel debug manager and go back though the installation instructions in the installation guide. Re-invoke the parallel debug manager and try to spawn the debugger again.

### Cannot create mailbox

*Description* The parallel debug manager was unable to create a mailbox for the new debugger that you were trying to spawn; the parallel debug manager must be able to create a mailbox in order to communicate with each debugger. This message usually indicates a resource limitation (you have more debuggers invoked than your system can handle).

*Action* If you have numerous debuggers invoked and you’re not using all of them, close some of them. If you are under a UNIX environment, use the ipcs command to check your message queues; use ipcrm to clean up the message queues.

### Cannot open log file

*Description* The parallel debug manager cannot find the filename that you supplied when you entered the DLOG command.

*Action*

❑ Be sure that the file resides in the current directory or in one of the directories specified by the D_DIR environment variable.

❑ Check to see if you mistyped the filename.

### Cannot open take file

*Description* The parallel debug manager cannot find the batch filename supplied for the TAKE command. You will also see this message if you try to execute a batch file that does not have a .pdm extension.

*Action*

❑ Be sure that the file resides in the current directory or in one of the directories specified by the D_DIR environment variable.

❑ Check to see if you mistyped the filename.

❑ Be sure that the batch filename has a .pdm extension.

❑ Be sure that the file has executable rights.

### Cannot open temporary file

*Description* The parallel debug manager is unable create a temporary file in the current directory.

*Action* Change the permissions of the current directory.

### Cannot seek in file

*Description* While the parallel debug manager was reading a file, the file was deleted or modified.

*Action* Be sure that files the parallel debug manager reads are not deleted or modified during the read.

### Cannot spawn child debugger

*Description* The parallel debug manager couldn’t spawn the debugger that you specified, because the parallel debug manager couldn’t find the debugger executable file (mpsim or ppsim). The parallel debug manager will first search for the file in the current directory and then search the directories listed with the PATH statement.

*Action* Check to see if the executable file is in the current directory or in a directory that is specified by the PATH statement. Modify the PATH statement if necessary, or change the current directory.

### Command error

*Description* The syntax for the command that you entered was invalid (for example, you used the wrong options or arguments).

*Action* Re-enter the command with valid parameters. Refer to the command summary on page DB:14-3*,* for a complete list of parallel debug manager commands and their syntaxes.

## D

### Debugger spawn limit reached

*Description* The parallel debug manager spawned the maximum number of debuggers that it can keep track of in its internal tables. The maximum number of debuggers that the parallel debug manager can track is 2048. However, your system may not have enough resources to support that many debuggers.

*Action* Before trying to spawn an additional debugger, close any debuggers that you don’t need to run.

## I

### Illegal flow control

*Description* One of the flow control commands (IF/ELIF/ELSE/ENDIF or LOOP/BREAK/CONTINUE/ENDLOOP) has an error. This error usually occurs when there is some type of imbalance in one of these commands.

*Action* Check the flow command construct for such problems as an IF without an ENDIF, a LOOP without an ENDLOOP, or a BREAK that does not appear between a LOOP and an ENDLOOP. Edit the batch file that contains the problem flow command, or interactively re-enter the correct command.

### Input buffer overflow

*Description* The parallel debug manager is trying to execute or manipulate an alias or shell variable that has been recursively defined.

*Action* Use the SET and/or ALIAS commands to check the definitions of your aliases and system variables. Modify them as necessary.

### Invalid command

*Description* The command that you entered was not valid.

*Action* Refer to the command summary on page DB:14-3 for a complete list of commands and their syntax.

### Invalid expression

*Description* The expression that you used with a flow control command or the @ command is invalid. You may see specific messages before this one that provide more information about the problem with the expression. The most common problem is the failure to use the $ character when evaluating the contents of a system variable.

*Action* Check the expression that you used. For more information about expression analysis, see Section 4.7, *Understanding the Parallel Debug Manager's Expression Analysis*.

### Invalid shell variable name

*Description* The system variable name that you assigned with the SET command is invalid. Variable names can contain any alphanumeric characters or underscore characters.

*Action* Use a different name.

## M

### Maximum loop depth exceeded

*Description* The LOOP/ENDLOOP command that you tried to execute had more than 10 nested LOOP/ENDLOOP constructs. LOOP/ENDLOOP constructs can be nested up to 10 deep.

*Action* Edit the batch file that contains the LOOP/ENDLOOP construct, or re-enter the LOOP/ENDLOOP command interactively.

### Maximum take file depth exceeded

*Description* The batch file that you tried to execute with the TAKE command called or nested more than 10 other batch files. The TAKE command can handle only batch files that are nested up to 10 deep.

*Action* Edit the batch file.

## U

### Unknown processor name *“name”*

*Description* The processor name that you specified with the –g option or a processor name within a group that you specified with the –g option does not match any of the names of the debuggers that were spawned under the parallel debug manager.

*Action* Be sure that you’ve correctly entered the processor name.

## F.4 Additional Instructions for Expression Errors

Whenever you receive an expression error, you should re-enter 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.

## F.5 Additional Instructions for Hardware Errors

If you continue to receive the messages that send you to this section, this indicates persistent hardware problems.

❑ If a bus fault occurs, the emulator may not be able to access memory.

❑ The MVP must be reset before you can use the emulator. Most target systems reset the MVP at power-up; your target system may not be doing this.

Appendix G

# Glossary

## A

**aggregate type:** A C data type, such as a structure or an array, in which a variable is composed of multiple other variables, called members. See also *scalar type*, *DISP window*

**aliasing:** A method of customizing debugger or other shell commands, providing a shorthand method for entering often-used command strings.

**assembler:** A software utility that creates a machine-language program from a source file. There are two assemblers associated with the MVP: a mnemonic-based RISC-type assembler for the MP and an algebraic assembler for the PP.

**assembly mode:** In the debugger, a debugging mode that shows assembly language code in the DISASSEMBLY window and doesn't show the FILE window, regardless of what type of code is currently running.

**auto mode:** In the debugger, a context-sensitive debugging mode that automatically switches between showing assembly code in the DISASSEMBLY window and C code in the FILE window, depending on the type of code that is currently running.

## B

**benchmarking:** A type of program execution that allows you to track the number of CPU cycles consumed by a specific section of code.

# C

**C compiler:** A program that translates C source statements into assembly language source statements or object code.

**cache:** A fast memory into which frequently used data or instructions from slower memory are copied for fast access. Fast access is facilitated by the cache’s high speed and its on-chip proximity to the CPU.

**CALLS window:** A debugger window that lists the functions called by your program.

**casting:** A feature of C expressions that allows you to use one type of data as if it were a different type of data.

**children:** Secondary windows within a debugger that are opened for aggregate data types that are members of a parent aggregate type displayed in an existing DISP window.

**CLK:** A pseudoregister that shows the number of CPU cycles consumed during benchmarking.

**code-display windows:** Debugger windows that show code, text files, or code-specific information.

**COFF:** *Common object file format*. An object file format that promotes modular programming by supporting sections.

**common object file format:** See *COFF*

**CPU window:** A debugger window that displays the contents of MVP on-chip registers, including the program counter, status register, and general-purpose registers.

**crossbar:** A generally configurable, high-speed bus switching network for a multiprocessor system, permitting any of several processors to connect to any of several memory modules.

**cross-reference lister:** A debugging tool that accepts linked object files as input and produces cross-reference listings as output.

## D

**data cache:** The MP's two SRAM banks that hold cached data needed by the MP. Data RAMs for the PPs are not cached.

**data-display windows:** Debugger windows for observing and modifying various types of data.

**debugger:** A window-oriented software interface that helps you to debug MVP programs running on an MVP emulator or simulator.

**disassembly:** Assembly language code formed from the reverse assembly of the contents of memory.

**DISASSEMBLY window:** A debugger window that displays the disassembly of memory contents.

**DISP window:** A debugger window that displays the members of an aggregate data type.

**D_OPTIONS:** An environment variable that you can use for identifying often-used debugger options.

**double-precision floating-point:** A floating-point number with 64 bits plus an additional hidden bit.

**doubleword:** A 64-bit value.

## E

**emulator:** A debugging tool that is external to the target system and that provides direct control over the MVP that is in the target system.

**environment variable:** 1) A special system symbol that the debugger uses for finding directories or obtaining debugger options. 2) A system symbol that can be used to modify command-line input for the assembler or linker, or to modify the environment.

**executive:** The portion of a multitasking software system that is responsible for executing application tasks, providing communications among tasks, and managing shared resources.

**external address:** See *off-chip address*

## F

**FILE window:** A debugger window that displays C code statements. It can be used to display any text file.

## H

**halfword:** A 16-bit value.

**hex conversion utility:** A tool that translates COFF object files into one of several standard ASCII hexadecimal formats suitable for loading into an EPROM programmer.

## I

**internal address:** See *on-chip address*

**IP:** *Instruction pointer.* The MP register that points to the instruction currently in the fetch stage of the pipeline.

## L

**linker:** A software tool that combines object files to form an object module that can be allocated into system memory and executed by the device.

**LSB:** *Least significant bit.* The bit having the smallest effect on the value of a binary numeral, usually the rightmost bit. The MVP numbers the bits in a word from 0 to 31, where bit 0 is the LSB.

## M

**master processor:** See *MP*

**memory map:** A map of target system memory space that is partitioned into functional blocks.

**MEMORY window:** A debugger window that displays the contents of memory.

**mixed mode:** A debugging mode that simultaneously shows both assembly language code in the DISASSEMBLY window and C code in the FILE window.

**MP:** *Master processor.* A general-purpose RISC processor that coordinates the activity of the other processors on the MVP. The MP includes an IEEE-754 floating-point hardware unit.

**mpcl:** A shell utility that invokes the MVP master processor compiler, assembler, and linker to create an executable object file version of your MP program.

**MSB:** *Most significant bit*. The bit having the greatest effect on the value of a binary numeral. It is the leftmost bit. The MVP numbers the bits in a word from 0 to 31, where bit 31 is the MSB.

**multimedia video processor:** See *MVP*

**MVP:** *Multimedia video processor*. A single-chip multiprocessor device that accelerates applications such as video compression and decompression, image processing, and graphics. The multimedia video processor contains a master processor and from one to eight parallel processors, depending on the device version. For example, the TMS320C80 device contains four PPs.

**MVP multitasking executive:** See *executive*

## O

**off-chip address:** An address external to the MVP chip. Addresses from 0x0200 0000 to 0xFFFF FFFF are off-chip addresses. See also *on-chip address*

**on-chip address:** An address internal to the MVP chip. Addresses from 0x0000 0000 to 0x1FFF FFFF are on-chip addresses. See also *off-chip address*

## P

**parallel processor:** See *PP*

**parameter RAM:** A general-purpose 2K-byte RAM that is associated with a specific processor, part of which is dedicated to packet transfer information and the processor interrupt vectors.

**PC register:** The 32-bit register that contains the address of the next instruction (PC field). In the PP, the PC register also includes the G and L control bits.

**PDM:** *Parallel debug manager.* A program used for invoking and controlling multiple debuggers for the purpose of debugging code in a parallel-processing environment.

**port address:** The PC I/O memory space that the debugger uses for communicating with the emulator. The port address is selected via switches on the emulator board and communicated to the debugger with the –p debugger option.

**PP:** *Parallel processor.* The MVP's advanced digital signal processor that is used for video compression/ decompression (P × 64 or MPEG), still-image compression/ decompression (JPEG), 2-D and 3-D graphic functions such as line draw, trapezoid fill, antialiasing, and a variety of high-speed integer operations on image data. An MVP single-chip multiprocessor device may contain from one to eight PPs, depending on the device version.

**ppcl:** A shell utility that invokes the MVP's PP compiler, assembler, and linker to create an executable object file version of your PP program.

**PP command interface:** The software interface through which the MP (or other client processor) issues commands to be executed by a server PP.

## R

**refresh:** A method of restoring the charge capacitance to a memory device (such as a DRAM or VRAM) or of restoring memory contents.

**reset:** A means to bring processors to known states by setting registers and control bits to predetermined values and signaling execution to start at a specified address. At reset, the MP loads the address 0xFFFF FFF8 into the PC register.

## S

**scalar type:** A C type in which the variable is a single unit, not an aggregate type. See also *aggregate type*

**side effects:** A feature of C expressions that enables you to use an assignment operator in an expression to affect the value of one of the components in the expression (for example, one of the components could be incremented).

**simulator:** A development tool that simulates the operation of the MVP and allows you to execute and debug application programs using the MVP debugger.

**single-precision floating-point:** 32-bit floating-point number.

**symbol:** A string of alphanumeric characters that represents an address or a value.

**symbol table:** A portion of a COFF object file that contains information about the symbols that are defined and used by the file.

**symbolic debugging:** The ability of a software tool to retain symbolic information so that it can be used by a debugging tool such as a simulator or an emulator.

## T

**target system:** An MVP board that works with the emulator.

**TC:** *Transfer controller*. The MVP's on-chip DMA controller for servicing the cache and for transferring one-, two-, and three-dimensional data blocks between each processor on the MVP and its external memory.

**transfer controller:** See *TC*

**trap:** An exceptional condition caused by the currently executing instruction that forces a program to be interrupted before execution of the next instruction begins. After the processor has serviced the trap, it typically resumes execution of the interrupted program at the instruction that immediately follows the instruction that caused the trap.

## V

**VC:** *Video controller*. The portion of the MVP responsible for the video interface.

**video controller:** See *VC*

## W

**WATCH window:** A debugger window that displays the values of selected expressions, symbols, addresses, and registers.

**window:** A defined rectangular area of virtual space on the display.

**word:** A sequence of 32 adjacent bits that constitutes a register or memory value. The PP supports 32-bit words. The MP also supports doublewords of 64 bits for loads and stores.

# Index