TEXAS INSTRUMENTS

Parallel Debug Manager

Addendum to the TMS320C4x and TMS320C5x C Source Debugger User's Guides

Addendum

1993 Microprocessor Development Systems

TEXAS INSTRUMENTS

Printed in U.S.A., April 1993
2617619–9741 revision *

SPRU094

Parallel Debug Manager

Addendum to the TMS320C4x and TMS320C5x C Source Debugger User's Guides

SPRU094
April 1993

IMPORTANT NOTICE

Texas Instruments Incorporated (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 current specifications 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.

Please be aware that TI products are not intended for use in life-support appliances, devices, or systems. Use of TI product in such applications requires the written approval of the appropriate TI officer. Certain applications using semiconductor devices may involve potential risks of personal injury, property damage, or loss of life. In order to minimize these risks, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards. Inclusion of TI products in such applications is understood to be fully at the risk of the customer using TI devices or systems.

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.

Read This First

What Is This Addendum About?

The TMS320C4x and TMS320C5x emulation systems are true multiprocessing debugging systems. Each allows you to debug your entire application by using the parallel debug manager (PDM). The PDM is a command shell that controls and coordinates multiple debuggers, providing you with the ability to:

- 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

You can run multiple debuggers under the control of the PDM only on PCs running OS/2 or Sun workstations running OpenWindows. Before you can use the PDM, you must install the emulator and software as described in the installation guide.

How to Use This Manual

This addendum describes the parallel debug manager (PDM) for the TMS320C4x and TMS320C5x systems.

- **Chapter 1,** *Getting Started With the Parallel Debug Manager,* tells you how to invoke the PDM and individual debuggers and describes execution-related commands. This chapter also includes information about describing your target system in a configuration file.
- **Chapter 2,** *PDM Shell Commands,* describes commands that control how you send commands to the individual debuggers and how you can use the output from the PDM command line.
- **Chapter 3,** *Additional Help,* provides a summary of commands and error messages.

Notational Conventions

This document uses the following conventions.

- The TMS320C40 processor is referred to as the **'C4x**.
- The TMS320C50, TMS320C51, and TMS320C53 processors are referred to collectively as the **'C5x**.
- PDM commands are not case sensitive; you can enter them in lowercase, uppercase, or a combination.
- 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 PDM Display Area
`echo $mask3`	`36+47`

`set`	`dgroup  "proc1 proc2 proc3"`
	`i  "hello world"`
	`val_proc1  "24"`
	`val_proc2"  "34"`
	`val_proc3  "45"`

  In this example, the left column identifies PDM commands that you type in. The right column identifies the result that the PDM displays in the PDM 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:

  echo *string*

  echo is the command. This command has one parameter, indicated by *string*.

- 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:

  help [*command*]

  The HELP command has one parameter, *command*, 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:

  unalias {*alias name* | ***}**

  This provides two choices: **unalias** *alias name* or **unalias ***

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

Related Documentation From Texas Instruments

The following books describe the TMS320C4x and TMS320C5x devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C4x C Source Debugger User's Guide (literature number SPRU054) tells you how to invoke the 'C4x emulator and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints, and includes a tutorial that introduces basic debugger functionality.

TMS320C4x User's Guide (literature number SPRU063) describes the 'C4x 32-bit floating-point processor, developed for digital signal processing as well as parallel processing applications. Covered are its architecture, internal register structure, instruction set, pipeline, specifications, and operation of its six DMA channels and six communication ports. Software and hardware applications are included.

TMS320C5x C Source Debugger User's Guide (literature number SPRU055) tells you how to invoke the 'C5x emulator, SWDS, EVM, and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints, and includes a tutorial that introduces basic debugger functionality.

TMS320C5x User's Guide (literature number SPRU056) describes the TMS320C5x 16-bit, fixed-point, general-purpose digital signal processors. Covered are its architecture, internal register structure, instruction set, pipeline, specifications, DMA, and I/O ports. Software applications are covered in a dedicated chapter.

If You Need Assistance. . .

If you want to. . .	Do this. . .
Request more information about Texas Instruments Digital Signal Processing (DSP) products	Write to: Texas Instruments Incorporated Market Communications Manager, MS 736 P.O. Box 1443 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call the TI Literature Response Center: **(800) 477–8924**
Ask questions about product operation or report suspected problems	Call the DSP hotline: **(713) 274–2320** **FAX: (713) 274–2324**
Report mistakes in this document or any other TI documentation	Send your comments to: Texas Instruments Incorporated Technical Publications Manager, MS 702 P.O. Box 1443 Houston, Texas 77251–1443

Trademarks

OpenWindows is a trademark of Sun Microsystems, Inc.

OS/2 is a trademark of International Business Machines Corp.

UNIX is a registered trademark of Unix System Laboratories, Inc

Contents

Chapter 1

Getting Started With the Parallel Debug Manager

This chapter describes the parallel debug manager (PDM) for the TMS320C4x and TMS320C5x systems, tells you how to invoke the PDM and individual debuggers, and describes execution-related commands. Also included in this chapter is information about describing your target system in a configuration file.

Before you can use the PDM, you must install the emulator and software as described in the installation guide.

Topic **Page**

1.1 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.

- ❑ The ’C4x PPDS (parallel processing development system) comes with a file called *board.dat*. This file describes to the debugger the PPDS device chain and gives the four ’C4xs the names CPU_A, CPU_B, CPU_C, and CPU_D. 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 section and go on to Section 1.2 if you’re using the PPDS.

- ❑ If you’re using an emulation scan path that contains only one ’C5x and no other devices, you can use the *board.dat* file that comes with the ’C5x emulator kit. This file describes to the debugger the single ’C5x in the scan path and gives the ’C5x the name C50_1. 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 section and go on to Section 1.2.

- ❑ If you plan to use a different target system, you must follow these steps:

  Step 1: Create the board configuration file.

  Step 2: Translate the board configuration file to binary so that the debugger can read it.

  Step 3: Specify the configuration file when invoking the debugger.

  These steps are described in the following subsections.

Step 1: Create the board configuration file

To define your target system, you must create a board configuration file that describes your target system (also called an emulation scan path) to the debugger. The file consists of a series of entries, each describing one device on your target system. You must list individual devices on your system in the board configuration file in order for the debugger to work. The configuration file will be referred to as *board.cfg* in this book.

Example 1–1 shows a board.cfg file that describes the ’C4x PPDS. It lists twelve octals named A1–A9, AA, AB, and AC, followed by four ’C4x devices named CPU_A, CPU_B, CPU_C, and CPU_D.

Example 1–1. The 'C4x PPDS Device Chain

(a) A sample board.cfg file

Device Name	Device Type	Comments
"A1"	BYPASS08	;the first device nearest TDO ;(test data out)
"A2"	BYPASS08	;the next device nearest TDO
"A3"	BYPASS08	
"A4"	BYPASS08	
"A5"	BYPASS08	
"A6"	BYPASS08	
"A7"	BYPASS08	
"A8"	BYPASS08	
"A9"	BYPASS08	
"AA"	BYPASS08	
"AB"	BYPASS08	
"AC"	BYPASS08	
"CPU_A"	TI320C4x	;the first 'C4x
"CPU_B"	TI320C4x	
"CPU_C"	TI320C4x	
"CPU_D"	TI320C4x	;the last 'C4x nearest TDI ;(test data in)

(b) The 'C4x PPDS 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. For example, 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 'C4x or 'C5x. 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 ('C4xs or 'C5xs) 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 that are used on the PPDS board. These devices cannot be debugged and must be worked around or "bypassed" when trying to access the 'C4xs or 'C5xs.

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 'C4x or 'C5x. 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:

  - **TI320C4x** describes the 'C4x.
  - **TI320C5x** describes the 'C5x.
  - **BYPASS##** describes devices other than the 'C4x, 'C5x, 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.
  - **SPL** specifies the scan path linker and must be followed by four subpaths, as in this syntax:

    "*device name*" **SPL** {*subpath0*} {*subpath1*} {*subpath2*} {*subpath3*}

    Each *subpath* can contain any number of devices. However, an SPL subpath **cannot** contain another SPL.

Example 1–2 shows a file that contains an SPL.

Example 1–2. A board.cfg File Containing an SPL

Device Name	Device Type	Comments
″A1″	BYPASS08	;the first device nearest TDO
″A2″	BYPASS08	
″CPU_A″	TI320C5x	;the first ’C5x
″HUB″	SPL	;the scan path linker
{		;first subpath
″B1″	BYPASS08	
″B2″	BYPASS08	
″CPU_B″	TI320C5x	;the second ’C5x
}		
{		;second subpath
″C1″	BYPASS08	
″C2″	BYPASS08	
″CPU_C″	TI320C5x	;the third ’C5x
}		
{		;third subpath
″D1″	BYPASS08	
″D2″	BYPASS08	
″CPU_D″	TI320C5x	;the fourth ’C5x
}		
{		;fourth subpath (contains nothing)
}		
″CPU_E″	TI320C5x	;the last ’C5x nearest TDI

Note: The indentation in the file is for readability only.

Step 2: Translate the configuration file to binary

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. 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.

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 target system. The debugger looks for the board.dat file in the current directory and in the directories named with the D_DIR environment variable.

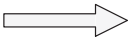
If you used a name other than board.dat or if the board.dat file is not in the current directory or in a directory named with D_DIR, you must use the –f option when you invoke the debugger. The –f option allows you to specify a board configuration file (and pathname) that will be used instead of board.dat. The format for this option is:

–f *filename*

1.2 Invoking a Standalone Debugger

There are two ways to invoke a debugger: you can invoke a standalone debugger, or you can invoke several debuggers that are under control of the PDM. If you want to invoke a debugger that is under control of the PDM, see Section 1.3.

Here's the basic format for the command that invokes a standalone debugger:



'C4x:	**emu4x**	**–n** *processor name*	[*filename*]	[*options*]
'C5x:	**emu5x**	**–n** *processor name*	[*filename*]	[*options*]

- **emu4x** and **emu5x** 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. 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.

  The processor name must match one of the names defined in your board configuration file (see Section 1.1). For example, to invoke the debugger for a 'C5x that you had defined as CPU_A, you would enter:

  `emu5x -n CPU_A` ↵

- *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 are using multiple extensions; you must specify the *entire* filename if the filename has more than one extension.

- –*options* supply the debugger with additional information. Table 1–1 on page 1-10 lists the debugger options that you can use when invoking a debugger, and the subsections that follow the table describe these options. You can also specify filename and option information with the D_OPTIONS environment variable (see *Setting up the environment variables* in your installation guide).

Once you have invoked a standalone debugger, turn to your debugger user's guide; the rest of this addendum describes commands and functions of the PDM.

1.3 Invoking the PDM

The TMS320C4x and TMS320C5x emulation systems are true multiprocessing debugging systems. Each allows you to debug your entire application by using the PDM. The PDM is a command shell that controls and coordinates multiple debuggers, providing you with the ability to:

- 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

You can operate the PDM only on PCs running OS/2 or Sun workstations running OpenWindows. The PDM is invoked and PDM commands are executed from a command shell window under the host windowing system. From the PDM, you can invoke and control debuggers for each of the processors in your multiprocessing system.

The format for invoking the PDM is:

```
pdm
```

Once you invoke the PDM, you will see the PDM command prompt (PDM:1>>) and can begin entering commands.

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

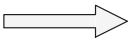
Note: Using the PDM on UNIX Systems

The PDM environment uses the interprocess communication (IPC) features of UNIX (shared memory, message queues, and semaphores) to provide and manage communications between the different tasks. If you are not sure if the IPC features are enabled, see your system administrator. To use the PDM environment, you should be familiar with the IPC status (ipcs) and IPC remove (ipcrm) UNIX commands. If you use the UNIX task kill (kill) command to terminate execution of tasks, you will also need to use the ipcrm command to terminate the shared memory, message queues, and semaphores used by the PDM.

1.4 Invoking Individual Debuggers From the PDM

When you debug a multiprocessing application, each processor must have its own debugger. These debuggers can be invoked individually from the PDM command line.

To invoke a debugger, use the SPAWN command. Here's the basic format for this command:



'C4x: **spawn emu4x –n** *processor name* [*filename*] [*options*]
'C5x: **spawn emu5x –n** *processor name* [*filename*] [*options*]

- **emu4x** and **emu5x** are the executables that invoke the 'C4x or 'C5x version of the debugger. In order to invoke a debugger, the PDM must be able to find the executable file for that debugger. The PDM will first search the current directory and then search the directories listed with the PATH statement.

- **–n** *processor name* supplies a processor name. You *must* use the –n option because the PDM uses processor names to identify the various 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.

  The processor name must match one of the names defined in your board configuration file (see Section 1.1). For example, to invoke a debugger for a 'C4x that you had defined as CPU_A, you would enter:

  ```
  spawn emu4x –n CPU_A ↵
  ```

- *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 are using multiple extensions; you must specify the *entire* filename if the filename has more than one extension.

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

Table 1–1. Summary of Debugger Options

Option	Description
–b[b]	Select the screen size
–f *filename*	Identify a new board configuration file
–i *pathname*	Identify additional directories
–s	Load the symbol table only
–t *filename*	Identify a new initialization file
–v	Load without the symbol table
–x	Ignore D_OPTIONS

Selecting the screen size (–b option)

By default, the debugger uses an 80-character-by-25-line screen. If you'd like to use a different screen size, the method for doing so varies, depending on the type of system that you're using:

- **PC systems.** You can use the –b or –bb option to select one of these preset screen sizes:
  - **–b** Screen size is 80 characters by 43 lines for EGA or VGA displays.
  - **–bb** Screen size is 80 characters by 50 lines for a VGA display only.
- **Sun systems.** 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 provide a preset screen size, or by resizing the screen at run time.
  - **Using a preset screen size.** Use the –b or –bb option to select one of these preset screen sizes:
    - **–b** Screen size is 80 characters by 43 lines.
    - **–bb** Screen size is 80 characters by 50 lines.
  - **Resizing the screen at run time.** You can resize the screen at run time by using your mouse to change the size of the operating-system window that contains the debugger. The maximum size of the debugger screen is 132 characters by 60 lines.

Identifying a new board configuration file (–f option)

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 emu4x –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 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).

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 emuinit.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 installation guide).

1.5 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: Entering Commands From the PDM

Each debugger that runs under the PDM must have a unique processor name. The PDM does not keep track of existing processor names. When you send a command to a debugger, the PDM 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 CPU_A, that debugger's prompt will be CPU_A>.
- ❑ Identifying the individual debuggers on the screen (Sun systems only). 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 must use the –n option when you invoke a debugger. For example, to name one of the 'C5x processors CPU_B, use the following command to invoke the debugger:

```
spawn emu5x -n CPU_B ↵
```

From this point onward, whenever you needed to identify this debugger, you could identify it by its processor name, *CPU_B.*

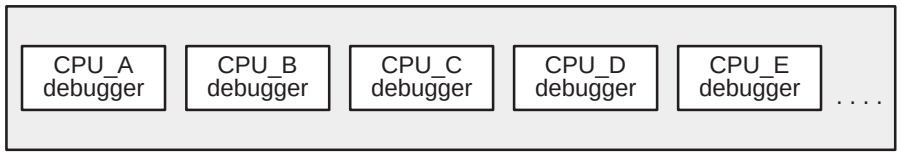
The processor name that you supply can consist of up to eight alphanumeric characters or underscore characters and must begin with an alphabetic character. Note that the name is not case sensitive. The processor name must match one of the names defined in your board configuration file (see Section 1.1).

Organizing processors into groups

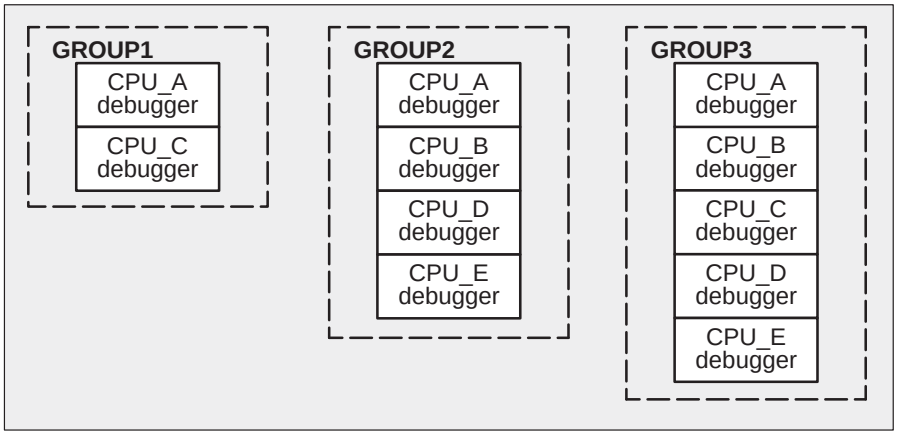
Processors can be organized into groups; these groups are identified by names defined with the SET command. Each processor can belong to any group, all groups, or a group of its own. Figure 1–1 *(a)* shows an example of processors that could exist in a system, and Figure 1–1 *(b)* illustrates three examples of named groups. GROUP1 contains two processors, GROUP2 contains four processors, and GROUP3 contains five processors.

Figure 1–1. Grouping Processors

(a) A system with x processors



(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 characters or underscore characters.

For example, to create the GROUP1 group illustrated in Figure 1–1 *(b)*, you could enter the following on the PDM command line:

```
set GROUP1 = CPU_A CPU_C ↵
```

The result is a group called GROUP1 that contains the processors named CPU_A and CPU_C. 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 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 processors called CPU_B, CPU_D, and CPU_E, enter:

```
set dgroup = CPU_B CPU_D CPU_E ↵
```

The PDM 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 PDM to use the processors listed previously in the group as part of the new list of processors.

Suppose GROUPA contained CPU_C and CPU_D. If you wanted to add CPU_E to the group, you'd enter:

```
set GROUPA = $GROUPA CPU_E ↵
```

After entering this command, GROUPA would contain CPU_C, CPU_D, and CPU_E.

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 PDM lists all of the groups and the processors associated with them:

```
GROUP1   "CPU_A CPU_C"
GROUPA   "CPU_C CPU_D CPU_E"
dgroup   "CPU_C CPU_D CPU_E"
```

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

```
set dgroup ↵
dgroup   "CPU_C CPU_D CPU_E"
```

❑ 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 CPU_A, CPU_C, CPU_D, and CPU_E. If you wanted to remove CPU_E, you could enter:

```
unset GROUPB ↵
set GROUPB = CPU_A CPU_C CPU_D ↵
```

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: Using the UNSET * Command

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 PDM can't find the default group name (dgroup).

1.6 Running and Halting Code

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

- PRUNF starts the processors running free, which means they are disconnected from the emulator.
- PRUN starts the processors running under the control of the emulator.
- PSTEP causes the processors 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 PDM command line:
  - **Without –r**, control is not returned to the command line until each debugger in the group finishes running code. If you want to to break out of a synchronous command and regain control of the PDM command line, press CONTROL C in the PDM window. This will return control to the PDM 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.
- You can specify a *count* for the PSTEP command so that each processor in the group will step for *count* number of times.

Note: Single-Stepping With Breakpoints

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.

Halting processors at the same time

After you enter a PRUNF command, you can use the PHALT 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*}]

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 essentially 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 PDM also lists the current PC value for that processor. 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_A after you've executed a global PRUN, enter:

```
stat –g GROUP_A ↵
```

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

```
[CPU_C] Running
[CPU_D] Halted   PC=200001A
[CPU_E] Running
```

1.7 Exiting a Debugger or the PDM

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 PDM to quit **all** of the debuggers (and also close the PDM).

Chapter 2

PDM Shell Commands

The commands such as PRUN, STAT, and PHALT described in Chapter 1 are execution-related commands. The commands described in this chapter are additional PDM commands that control *how* you send commands to the individual debuggers and how you can use the output from the PDM command line. For example, these shell commands allow you to:

- ❑ Create system variables
- ❑ Execute commands from a batch file
- ❑ Record information from the PDM display area
- ❑ Conditionally execute or loop through PDM commands
- ❑ Define command strings
- ❑ Enter operating-system commands
- ❑ Use the command history

Topic **Page**

2.1 Understanding the PDM's Expression Analysis

The PDM analyzes expressions differently than individual debuggers do (expression analysis for the debugger is described in the *Basic Information About C Expressions* chapter of the debugger user's guide). The PDM uses a simple integral expression analyzer. You can use expressions to cause the PDM to make decisions as part of the @ command and the flow control commands (described in Sections 2.3 and 2.9, respectively).

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

Table 2–1 summarizes the PDM operators. The PDM interprets the operators in the order that they're listed in Table 2–1 (left to right, top to bottom).

Table 2–1. PDM 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

2.2 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 the individual debuggers. 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 PDM command line:

  - **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 PDM command window. These results will be displayed by processor. For example:

    ```
    send ?pc ↵
    [CPU_C]  0x40000000
    [CPU_D]  0x40000004
    [CPU_E]  0x4000000A
    ```

    If you want to break out of a synchronous command and regain control of the PDM command line, press CONTROL C in the PDM window. This will return control to the PDM 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.

    The –r option is useful when you want to exit from a debugger but not from the PDM. 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 PDM waits for a response from all of the debuggers that are quitting. By using –r, you can gain immediate control of the PDM and continue sending commands to the remaining debuggers.

2.3 Using System Variables

You can use the SET, @, and UNSET commands to create, modify, and delete system variables. In addition, you can use the SET command with system-defined variables.

Creating your own system variables

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

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

The *variable name* can consist of up to 128 alphanumeric characters 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 whenever you use the variable result.

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

```
send CPU_D ? $result ↵
```

The dollar sign (**$**) tells the PDM 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 PDM to append the contents of the variable name to a string that precedes or follows the braces. For example (the ECHO command is described on page 2-10):

```
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 PDM 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 differences 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 PDM prompt

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

```
set prompt = 3PROCs ↵
```

After entering this command, the PDM 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 1-17) 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/ENDLOOP command is described on page 2-11):

```
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 1-13), use the UNSET command with an asterisk instead of a variable name:

`unset *` ⏎

Note: Using the UNSET * Command

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

2.4 Evaluating Expressions

The debugger includes a ? command that evaluates an expression and shows the result in the display area of the COMMAND window. The PDM 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 PDM 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 PDM 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 2.1.
- 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	Parameter	Result
*****	Default for the data type	**o**	Octal
c	ASCII character (bytes)	**p**	Valid address
d	Decimal	**s**	ASCII string
e	Exponential floating point	**u**	Unsigned decimal
f	Decimal floating point	**x**	Hexadecimal

Suppose proc0 has two variables defined: j is equal to 5, and k is equal to 17. Also assume that proc1 contains variables j and k: j is equal to 12, and k is equal to 22.

```
set dgroup = proc0 proc1 ↵
eval val = j + k ↵
set ↵
dgroup    ″proc0 proc1″
val_proc0 ″23″
val_proc1 ″34″
```

Notice that the PDM created a system variable for each processor: val_proc0 for proc0 and val_proc1 for proc1.

2.5 Executing PDM Commands From a Batch File

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

take *batch filename*

The *batch filename* **must** have a .pdm extension, or the PDM will not be able to read the file. If you don't supply a pathname as part of the filename, the PDM first looks 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. 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 PDM 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 in Section 2.9) in a batch file and execute the file with the TAKE command.

2.6 Recording Information From the PDM Display Area

By using the DLOG command, you can record the information shown in the PDM display area into a log file. The log file is a system file that contains commands you've entered, their results, and error or progress messages. When using DLOG, the PDM 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 TAKE.

❑ To begin recording the information shown in the PDM display area, 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.

2.7 Echoing Strings to the PDM Display Area

You can display a string in the PDM display area 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 PDM display area.

You can also use ECHO to show the contents of a system variable:

```
echo $var_proc1
34
```

The ECHO command works exactly the same as the ECHO command described in the debugger user's guide, except that you can use it outside of a batch file.

2.8 Pausing the PDM

Sometimes you may want the PDM 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 PDM reads this command in a batch file or during a flow control command segment, the PDM stops execution and displays the following message:

```
<< pause – type return >>
```

To continue processing, press ↵.

2.9 Controlling PDM Command Execution

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

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

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

- If the expression for the IF is nonzero, the PDM executes all commands between the IF and ELIF, ELSE, or ENDIF.
- The ELIF is optional. If the expression for the ELIF is nonzero, the PDM 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 PDM 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*
debugger commands
[**break**]
[**continue**]
endloop

The PDM version of the LOOP/ENDLOOP commands is different from the debugger version of these commands. Instead of accepting any expression, the PDM version of the LOOP command evaluates only Boolean expressions. If the Boolean expression evaluates to true (1), the PDM 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 flow-control commands can be entered interactively or included in a batch file that is executed by the TAKE command. When you enter LOOP or IF from the PDM command line, 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 PDM continues to prompt you for input using the ? until you enter ENDIF (for an IF statement) or ENDLOOP (for a LOOP statement). After you enter ENDIF or ENDLOOP, the PDM immediately executes the IF or LOOP command.

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

The IF/ENDIF and LOOP/ENDLOOP commands can be used together to perform a series of tests. For example, within a batch file, you can create a loop like the following:

```
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 (refer to page 2-9) to examine which test cases failed during the testing session.

2.10 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 PDM version of each of these commands is:

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

The PDM 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 aliased command.
- List all currently defined PDM aliases by entering ALIAS with no parameters.
- Find the definition of a PDM alias by entering ALIAS with only an alias-name parameter.
- Nest alias definitions.
- Redefine an alias.
- Delete a single PDM alias by supplying the UNALIAS command with an alias name, or delete all PDM aliases by entering UNALIAS *.

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

For more information about these features, refer to the *Defining Your Own Command Strings* section in your debugger user's guide.

The PDM 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 PDM versions of debugger commands. For example, the ML debugger command lists the memory ranges that are currently defined. To make a PDM 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 GROUP1:

```
ml GROUP1 ↵
```

- 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 PDM 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 PDM 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 GROUP3 contains CPU_A, CPU_B, and CPU_D. To make GROUP3 the current default group and make the PDM prompt the same name as your default group, enter:

  ```
  switch GROUP3 ↵
  ```

  This causes the default group (dgroup) to contain CPU_A, CPU_B, and CPU_D, and changes the PDM prompt to GROUP3:x>>.

2.11 Entering Operating-System Commands

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

system *operating-system command*

The SYSTEM command is similar to the debugger's SYSTEM command, but there are some differences.

- **Similarities.** You can enter operating-system commands without having to leave the primary environment (in this case, the PDM) and without having to open another operating-system window.
- **Differences.** Unlike the debugger version of the SYSTEM command:
  - The PDM version of the SYSTEM command cannot be entered without an operating-system command parameter. Therefore, you cannot use the command to open a shell.
  - There is no flag parameter; command output is always displayed in the PDM window.

2.12 Using the Command History

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

The PDM command history allows you to re-enter any of the last twenty commands:

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

  !! ↵

- To repeat any of the last twenty commands, use the following command:

  !*number*

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

  ```
  PDM:100>>echo hello
  hello
  PDM:101>>echo goodbye
  goodbye
  PDM:102>>!100
  echo hello
  hello
  ```

  Notice that the PDM displays the command that you are re-entering.

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

  !*string*

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

  ```
  PDM:103>>pstep –g GROUPA
  PDM:104>>send –g GROUPA ?pc
  [CPU_C]  0x40000000
  [CPU_D]  0x40000004
  PDM:103>>pstep –g GROUPB
  PDM:104>>send –g GROUPB ?pc
  [CPU_A]  0x4000001A
  [CPU_E]  0x40000014
  PDM:105>>!p
  pstep –g GROUPB
  ```

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

  history ↵

The command history for the PDM works differently from that of the debugger; the TAB and F2 keys have no command-history meaning for the PDM.

Chapter 3

Additional Help

This chapter summarizes the PDM commands and error messages.

Topic **Page**

3.1 Viewing the Description of a PDM Command

You can use the HELP command to display the syntax and a brief description of a specific command. The syntax for the HELP command is:

help [*command*]

If you omit the *command* parameter, the PDM lists all of the available commands.

3.2 Summary of PDM Commands

This section provides a quick reference to the PDM commands. Refer to the page numbers listed in the tables for more information about these commands.

Invocation commands

To do this	Use this command	See page
Exit any debugger and/or the PDM	**quit**	1-18
Invoke a 'C4x debugger	**spawn emu4x –n** *name* [*filename*] [*options*]	1-9
Invoke a 'C5x debugger	**spawn emu5x –n** *name* [*filename*] [*options*]	1-9
Invoke the PDM	**pdm**	1-8

Group-definition commands

To do this	Use this command	See page
Define a group of processors	**set** *group name* **=** *list of processors*	1-13
Delete a group	**unset** {*group name* | ***}**	1-15
List all groups of processors	**set**	1-15
Set the default group	**set dgroup =** *list of processors*	1-13

Execution-related commands

To do this	Use this command	See page
Find the execution status of a processor or a group of processors	**stat** [**–g** {*group* | *processor name*}]	1-17
Global halt	**phalt** [**–g** {*group* | *processor name*}]	1-17
Halt code execution	**pesc** [**–g** {*group* | *processor name*}]	1-17
Run code globally under the control of the emulator	**prun** [**–r**] [**–g** {*group* | *processor name*}]	1-16
Single-step globally	**pstep** [**–g** {*group* | *processor name*}] [*count*]	1-16
Start the processors running free	**prunf** [**–g** {*group* | *processor name*}]	1-16

Shell commands

To do this	Use this command	See page
Assign a variable to the result of an expression	@ *variable name* **=** *expression*	2-5
Change the PDM prompt	**set prompt =** *new prompt*	2-5
Conditionally execute PDM commands	**if** *expression* *debugger commands* [**elif** *expression* *debugger commands*] [**else** *debugger commands*] **endif**	2-11
Create your own system variables	**set** *variable name* **=** *string*	2-4
Define a custom command string	**alias** [*alias name* [**,** **"***command string***"**]]	2-13
Delete a system variable	**unset** {*variable name* | *****}	2-6
Delete an alias definition	**unalias** {*alias name* | *****}	2-13
Display a string to the PDM display area	**echo** *string*	2-10
Enter an operating-system command	**system** *operating-system command*	2-14
Evaluate an expression in a debugger or group of debuggers and set a variable to the result of the expression	**eval** [**–g** {*group* | *processor*}] *variable***=***expr*[**,** *format*]	2-7
Execute a batch file	**take** *batch filename*	2-8
List the last twenty commands	**history**	2-15
Loop through PDM commands	**loop** *Boolean expression* *debugger commands* [**break**] [**continue**] **endloop**	2-11
Pause the PDM	**pause**	2-10
Record the information shown in the PDM display area	**dlog** *filename* [**,** {**a** | **w**}]	2-9
Send a debugger command to an individual processor or a group of processors	**send** [**–r**] [{**–g** *group* | *processor name*}] *debugger cmd*	2-3
Use the command history	**!**{*prompt number* | *string*}	2-15
View the description of a PDM command	**help** [*command*]	3-2

3.3 Alphabetical Summary of PDM Messages

This section contains an alphabetical listing of the error messages that the PDM might display. Each message contains both a description of the situation that causes the message and an action to take.

> **Note: Errors in Batch Files**
>
> If errors are detected in a TAKE file, the PDM 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 PDM 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 PDM 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 PDM was never established. This usually occurs when you have a problem with your target system.

Action Exit the PDM and go back though the installation instructions in the installation guide. Re-invoke the PDM and try to spawn the debugger again.

Cannot create mailbox

Description The PDM was unable to create a mailbox for the new debugger that you were trying to spawn; the PDM 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 PDM 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 PDM 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 PDM 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 PDM was reading a file, the file was deleted or modified.

Action Be sure that files the PDM reads are not deleted or modified during the read.

Cannot spawn child debugger

Description The PDM couldn't spawn the debugger that you specified, because the PDM couldn't find the debugger executable file (emu4x or emu5x). The PDM 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 in Section 3.2 for a complete list of commands and their syntaxes.

Debugger spawn limit reached

Description The PDM spawned the maximum number of debuggers that it can keep track of in its internal tables. The maximum number of debuggers that the PDM 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 PDM 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 in Section 3.2 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. Refer to Section 2.1, page 2-2, for more information about expression analysis.

Invalid shell variable name

Description The system variable name that you used the SET command to assign 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 PDM.

Action Be sure that you've correctly entered the processor name.

Index

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

X