Data General

Data General Corporation, Westboro, Massachusetts 01580

Customer Documentation

Using the DG/UX™ Kernel Debugger

093–701075–01

AViiON®
PRODUCT LINE

Using the DG/UX™ Kernel Debugger

093–701075–01

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085–series) supplied with the software.

Notice

Using the DG/UX™ Kernel Debugger
093-701075-01

Revision History:		Effective with:
Original Release	– May 1990	
First Revision	– January 1992	DG/UX Release 5.4.1

A vertical bar in the margin of a page indicates substantive technical change from the previous revision.

Preface

This manual describes the DG/UX™ Kernel Debugger. The Kernel Debugger enables you to examine and modify the state of a DG/UX operating system kernel that is running on an AViiON® station or server.

The debugger functions as an integral part of the DG/UX operating system kernel. You can use the debugger to set breakpoints in the kernel, examine the state of processes, and examine and change a machine's memory and registers.

Who Should Read This Manual?

Users of this manual should be knowledgeable about operating system design topics such as virtual memory. You should be familiar with the Motorola 88000 architecture—particularly the architecture's registers, instruction set, and stack management.

Manual Organization

This manual contains three chapters and two appendixes.

Chapter 1	Introduces the debugger, lists its features, and reviews the debugger installation procedure.
Chapter 2	Explains how to enter the debugger, how to enter commands, and describes the expression parser.
Chapter 3	Explains the purpose of each debugger command and how to interpret the commands' output.
Chapter A	Provides a quick reference for the debugger commands.
Chapter B	Describes context blocks.

Related Documents

The following manuals provide information that you may find useful. The manuals can be ordered using the nine–digit ordering number shown in parentheses (see TIPS information in back of this manual for ordering instructions).

Managing the DG/UX™ System (093–701088).

Discusses the concepts and tasks related to DG/UX system management, providing general orientation to the administrator's job as well as instructions for managing disk resources, user profiles, files systems, printers and tape drives, and other features of the system. The manual approaches system administration through the sysadm facility.

Writing a Standard Device Driver for the DG/UX™ System (093–701053).

Describes how to write a device driver for a DG/UX system running on an AViiON computer. Describes drivers written to address both specific devices and adapters that manage secondary bus access to specific devices.

Programming in the DG/UX™ Kernel Environment (093–701083).

Describes the basics of kernel–level programming on the DG/UX system and provides reference pages for kernel–supplied utility routines.

AViiON® 300D Series Stations: Programming System Control and I/O Registers (014–001823).

Describes the system board architecture and explains how to program the board, including the monochrome and color graphics subsystems, keyboard interface, serial and parallel interfaces, LAN interface, and SCSI.

AViiON® 100, 200, 300, and 400 Series Stations and AViiON® 3000 and 4000 Series Systems: Programming System Control and I/O Registers (014–001800).

Describes the system board architecture and explains how to program the system board, including the monochrome and color graphics subsystems, serial and parallel interfaces, LAN interface, and SCSI.

AViiON® 4600 and 530 Series Stations: Programming System Control and I/O Registers (014–002076).

Describes the system board architecture and explains how to program the system board, including the color graphics controller, keyboard, serial interface, VME interface, LAN interface, and SCSI interface.

AViiON® 5000 and 6000 Series Systems: Programming System Control and I/O Registers (014–001805).

Describes the system board architecture and explains how to program the system board, including the serial and parallel interfaces, the VMEbus, and the associated I/O.

MC88100 RISC Microprocessor User's Manual (MC88100UM/AD).

Describes the Motorola 88100 Central Processing Unit (CPU), including the registers, addressing modes, internal and bus timing, and assembly–language instruction set. Obtain from Motorola Corporation, Phoenix, AZ.

MC88200 Cache/Memory Management Unit User's Manual (MC88200UM/AD).

Describes the Motorola 88200 Cache/Memory Management Unit (CMMU), including the CMMU registers, the cache and cache coherency, memory management and user/supervisor space, the Processor bus (Pbus), and the Memory bus (Mbus). Obtain from Motorola Corporation, Phoenix, AZ.

Reader, Please Note

Data General manuals use certain symbols and styles of type to indicate different meanings. The Data General symbol and typeface conventions used in this manual are defined in the following list. You should familiarize yourself with these conventions before reading the manual.

This manual also presumes the following meanings for the terms "command line," "format line," and "syntax line." A command line is an example of a command string that you should type verbatim; it is preceded by a system prompt and is followed by a delimiter such as the curved arrow symbol for the New Line key. A format line shows how to structure a command; it shows the variables that must be supplied and the available options. A syntax line is a fragment of program code that shows how to use a particular routine; some syntax lines contain variables.

Convention	Meaning
boldface	In command lines and format lines: Indicates text (including punctuation) that you type verbatim from your keyboard. All DG/UX commands, pathnames, and names of files, directories, and manual pages also use this typeface.
`constant width/monospace`	Represents a system response on your screen. Syntax lines and examples of code also use this font.
italic	In format lines: Represents variables for which you supply values; for example, the names of your directories and files, your username and password, and possible arguments to commands. In text: Indicates a term that is defined in the manual's glossary.
[*optional*]	In format lines: These brackets surround an optional argument. Don't type the brackets; they only set off what is optional. The brackets are in regular type and should not be confused with the boldface brackets shown below.
[ ]	In format lines: Indicates literal brackets that you should type. These brackets are in boldface type and should not be confused with the regular type brackets shown above.
...	In format lines and syntax lines: Means you can repeat the preceding argument as many times as desired.
$ and %	In command lines and other examples: Represent the system command prompt symbols used for the Bourne and C shells, respectively. Note that your system might use different symbols for the command prompts.
↵	In command lines and other examples: Represents the New Line key, which is the name of the key used to generate a new line. (Note that on some keyboards this key might be called Enter or Return instead of New Line.) Throughout this manual, a space precedes the New Line symbol; this space is used only to improve readability — you can ignore it.
< >	In command lines and other examples: Angle brackets distinguish a command sequence or a keystroke (such as <**Ctrl–D**>, <**Esc**>, and <**3dw**>) from surrounding text. Note that these angle brackets are in regular type and that you do not type them; there are, however, boldface versions of these symbols (described below) that you do type.
<, >, >>	In text, command lines, and other examples: These boldface symbols are redirection operators, used for redirecting input and output. When they appear in boldface type, they are literal characters that you should type.
□	In command lines and other examples: Represents the cursor, which indicates your current typing position on the screen.

Contacting Data General

Data General wants to assist you in any way it can to help you use its products. Please feel free to contact the company as outlined below.

Manuals

If you require additional manuals, please use the enclosed TIPS order form (United States only) or contact your local Data General sales representative.

Telephone Assistance

If you are unable to solve a problem using any manual you received with your system, free telephone assistance is available with your hardware warranty and with most Data General software service options. If you are within the United States or Canada, contact the Data General Customer Support Center (CSC) by calling 1–800–DG–HELPS. Lines are open from 8:00 a.m. to 5:00 p.m., your time, Monday through Friday. The center will put you in touch with a member of Data General's telephone assistance staff who can answer your questions.

For telephone assistance outside the United States or Canada, ask your Data General sales representative for the appropriate telephone number.

Joining Our Users Group

Please consider joining the largest independent organization of Data General users, the North American Data General Users Group (NADGUG). In addition to making valuable contacts, members receive FOCUS monthly magazine, a conference discount, access to the Software Library and Electronic Bulletin Board, an annual Member Directory, Regional and Special Interest Groups, and much more. For more information about membership in the North American Data General Users Group, call 1–800–932–6663 or 1–508–443–3330.

End of Preface

Contents

Tables

Figures

Figure

Chapter 1
Introduction to the Kernel Debugger

The DG/UX™ kernel debugger enables you to debug interactively the state of an AViiON™ computer and the DG/UX™ operating system kernel that is controlling the computer's resources. The debugger works with all AViiON series machines (both stations and servers) that are running the DG/UX operating system.

You can use the debugger to examine an AViiON computer's program counter, registers and memory. The debugger enables you to set breakpoints, examine the state of virtual processors, and halt the computer's processors to examine their state.

The debugger is a tool for experienced software developers. You should be familiar with the AViiON computer's 88000–based architecture and its instruction set.

You will find the debugger useful in these situations:

- when you are installing and testing device drivers or STREAMS modules.
- when you want to examine the state of a system that is hung or has shut down with a panic.

CAUTION: Because you can change values in an AViiON computer's registers and memory, and because you can halt the computer's processors, you should use the debugger in non–production environments. Changing the kernel stack or data structures could cause a system failure.

Basic Terms

Here is a list of the basic terminology that is unique to the DG/UX operating system and the kernel debugger.

Job processor (JP)	The DG/UX kernel is designed to support computer systems with multiple CPUs (or processors). Each physical processor is called a job processor.
Virtual processor (VP)	Processes on an AViiON system or station run on software abstractions called virtual processors (VPs). VPs use the resources of the underlying JPs and hide the JP implementation from the processes.
Focus	When you are using the debugger, you can stop the system and examine the state of a VP. This is called "focusing" on that VP. You can use debugger commands to focus on any VP.
Program counter (PC)	The PC is the address of a program's currently executing instruction. When you take a breakpoint, the debugger displays the PC's value, which is the point at which you are entering the debugger.
View PC	When the debugger is entered, the view program counter (**view_pc**) is set to the value of the current PC. The **view_pc** is a value kept by the debugger. The **view_pc** value is used as a default memory address in some debugger commands.

Debugger Features

You interact with the debugger through its command–driven interface. Some of the features of the interface, which uses a syntax that is the same as the interface used by the **crash (1M)** utility, are summarized below.

Commands

From the debugger command line, you can invoke commands to perform the following kinds of operations:

Examine and change memory	memory read, memory write, memory search, regular expression search, pattern dump, view, view down, and view up
Evaluate expressions	print a symbol table entry, translate an expression, evaluate an expression, and create and set a global symbol

Control execution	set a breakpoint, delete a breakpoint, proceed, and halt the debugger
Examine and change machine state	display/modify a general register, display/modify a control register, display the state of the Cache/Memory Management Unit (CMMU) chips
Examine kernel state	logical–to–physical address translation, traceback a VP stack, display the status of a VP, display the status of all job processors (JPs), and focus on a VP
General	help, display and set modes, and print

Expression Parser

The debugger contains an expression parser that enables you to supply expressions instead of addresses in debugger commands. The parser resolves the expression to an address. For example, you might want to set a breakpoint on an address that is 16 locations past the beginning of the kernel function named "read." When the parser sees the expression "read+16," it will resolve "read" to a symbol in the kernel symbol table, add 16 to that value, and return the resulting address to the command.

Command Line Editor

The debugger supports a version of the DG/UX operating system's **editread** utility. This utility enables you to retrieve and repeat commands, or retrieve commands and edit them. For example, you can press the uparrow key to retrieve your previous command. Or, you can press Ctrl–R to display your current **editread** configuration and key assignments. The **editread** utility is described in *Using the DG/UX™ System.*

Command Names

Most debugger commands have a "short form" name; these names are listed with the commands' descriptions in Chapter 3.

You can also create your own names for the debugger's commands. For example, the debugger's short name for the **memwrite** command is **mw**. By writing a macro, you could assign another name to the **memwrite** command, such as **memw**. See "Assigning Aliases to Debugger Commands" in Chapter 2.

Relationship to Other Debugging Tools

If you are developing system software for an AViiON computer, you'll find that the kernel debugger will be an important part of your development and testing tools. You may already be familiar with some of the other DG/UX testing tools, such as the Multi–extensible debugger (Mxdb), the symbolic debugger **sdb**, and the **crash** utility.

These tools operate at logically different levels, corresponding to user programs, the kernel, and the hardware. At the highest level are Mxdb and **sdb**, which you use to debug high–level language programs. You can put breakpoints at system calls, but you cannot put a breakpoint inside a system call.

At the next lower level is the **crash** utility, which you can use to examine the state of an AViiON computer and examine system dumps. However, **crash** cannot take control of the kernel or the computer's hardware. For example, you cannot set breakpoints from **crash**. The kernel continues to run while the you use the **crash** utility. Therefore, the state that **crash** provides may be inaccurate, because it can provide only snapshots of the kernel's changing state. You can use **crash** without affecting the machine's users. See the **crash (1M)** manual page for details.

At the lowest logical level is the kernel debugger. The kernel debugger is not a user program. Because the debugger is linked into the kernel, it can take control of the kernel and the computer hardware. Its main purpose is to allow you to debug the kernel, and secondarily, to identify problems with the hardware. The kernel debugger enables you to freeze the state of the entire machine, examine the state of the machine (and perhaps modify it) and restart the machine.

Installing the Kernel Debugger

The information provided in this section assumes that you have already loaded the kernel debugger software onto your system from its release tape. Instructions for loading the software are provided in the kernel debugger release notice.

The debugger will use approximately 1.5 Mbytes of your computer's main memory. This "wired" memory is available for use only by the debugger—it is not paged out or removed.

To link the debugger into the kernel, you must edit the kernel configuration file and rebuild the kernel. You do this from the **sysadm** utility's "System–>Kernel–>Build" menu. (The **sysadm** utility is described in detail in *Managing the DG/UX™ System.*) In the configuration file's "Tuneable Configuration Parameters" section, add the name **DEBUGGER** (in uppercase letters) after the "Parameter Name" heading. No entry is required in the "Value" column.

Whether you install the new kernel automatically or at a later time, you must shutdown and reboot the system for the changes that you made to the configuration file to take effect.

After you rebuild the kernel and boot the system, the debugger will announce to you that it is loaded by displaying its header message.

You cannot remove the debugger from the kernel dynamically. To "remove" the debugger, you must edit the system configuration file and remove the **DEBUGGER** parameter name. (Or, you can place a comment character (#) in front of the **DEBUGGER** parameter name.) Then, you must rebuild the kernel and reboot the machine.

If you are concerned about always using 1.5 Mbytes of main memory for the debugger, you can build two versions of your kernel—one with the debugger linked in, and the other without it linked in. Then, you can copy the kernel that you want to use and reboot your system.

End of Chapter

Chapter 2
Using the Kernel Debugger

The Kernel Debugger will take control of a machine in the following circumstances:

- when the machine is booted and the debugger is initialized
- when the kernel panics
- when the kernel hits a debugger breakpoint
- when you press Ctrl–P from the computer's operator console.

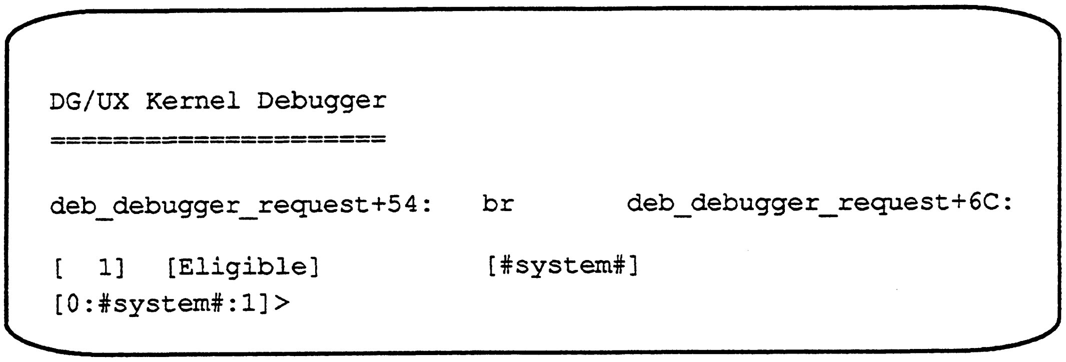
You do not need to have Superuser privileges to use the debugger. The debugger will always be available to you, regardless of the init level.

Note that you cannot see the debugger's display from a window in a window management system. For example, the X Window System ™ intercepts the Ctrl–P, and does not pass it to the debugger.

If you want use the debugger in a window environment, you use the debugger from an asynchronous terminal that is connected to the AViiON computer's serial port. See the description of the debugger's **mode** command in Chapter 3 for information about making the debugger recognize an asynchronous terminal.

Debugger Display

When you enter the debugger, it displays its header message, information about the program counter, information about virtual processors (VPs), and a command–line prompt. Figure 2–1 shows an example of the debugger's initial display.



```
DG/UX Kernel Debugger
=====================

deb_debugger_request+54:   br       deb_debugger_request+6C:

[  1]  [Eligible]          [#system#]
[0:#system#:1]>
```

Figure 2–1 Debugger Display

Program Counter Information

The first line below the header message provides a routine name and program counter information. In this example, the field "deb_debugger_request+54:" is the current value of the hardware program counter (PC) and is the point at which you entered the debugger. In this case, the debugger was entered 54 bytes into the "deb_debugger_request" routine. The second and third fields on this line represent the disassembled instruction at the PC's location.

Virtual Processor Information

The second line below the header message in Figure 2–1 provides virtual processor information. In this example, we are looking at VP number 1, which is in an Eligible state (can be run). The third field provides the name of the program that the VP is running. In this example, the VP is the kernel "idle loop," which is run when no other programs are being serviced—it has the default name of "#system#" because it represents a system process. If you were running a program such as **sh**, you would see the name "sh" in this field.

Prompt Line

The last line in a debugger display, which ends with a right angle bracket (>), is the prompt line. The three values in the square brackets (separated by colons) provide job processor (JP) information. The first value is the number of the JP on which the debugger was entered. For multi–processor machines, the number will be in the range 0–to–(#processors–1). For example, a dual–processor machine's JP number will be 0 or 1. For a single–processor machine, the number will always be 0.

The second value is the name of the program that the VP is running. In this example, the value is "#system#" because it represents a system process; no program is named in the virtual processor line.

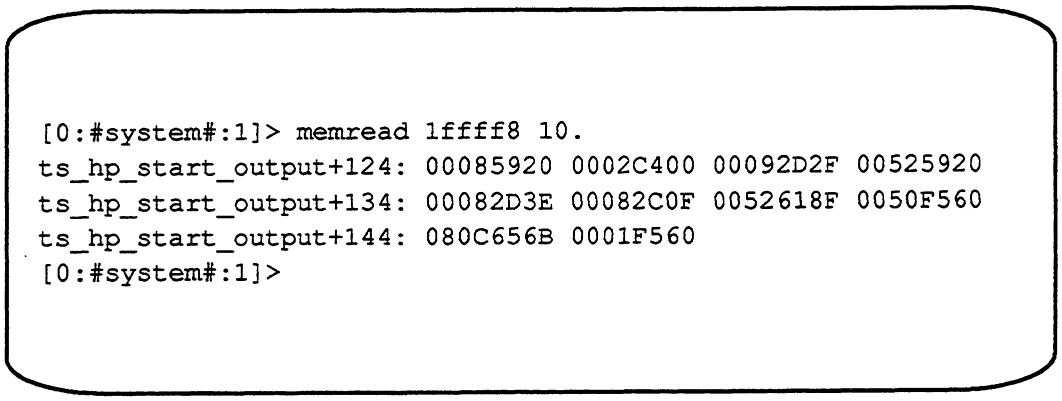
The third value is the number of the VP that is running. The number here and the number in the VP line will match when you enter the debugger. When you are using the debugger, you can focus (look at) different VPs. For example, if you focus on VP number 5, the third value will change to 5.

The VP that was running when the debugger takes control of the kernel is the VP that will restart when you leave the debugger. In other words, you can focus on any of the VPs while you are in the debugger, but the debugger always returns control to the VP that was interrupted when the debugger was entered.

Entering Commands

You enter your commands after the prompt (the right angle bracket). Press the New Line key to invoke the command. Figure 2–2 shows how your screen might look after you entered the following command:

`[0:#system#:1]>` **memread 1fff8 10.** ↲



```
[0:#system#:1]> memread 1ffff8 10.
ts_hp_start_output+124: 00085920 0002C400 00092D2F 00525920
ts_hp_start_output+134: 00082D3E 00082C0F 0052618F 0050F560
ts_hp_start_output+144: 080C656B 0001F560
[0:#system#:1]>
```

Figure 2–2 Sample Memory Read Display

The **memread** command in Figure 2–2 invokes the memory read command to display 10 (decimal) memory elements, starting at logical location 1fff8. The period following the "10" signifies a decimal number.

The debugger will display an error message if you enter an invalid command. You can then enter a valid command or use the command line editor to recall the command and edit it.

Assigning Aliases to Debugger Commands

By using macros, you can create aliases for debugger commands. The format for such a macro is:

macroname( ) *macro_definition*

There should be no whitespace between the *macroname* and the left parenthesis. In this example, **r5** becomes an alias for the debugger command **reg r5**:

`[0:#system#:1]>` **r5() reg r5** ↲

If you type `r5` at the prompt, the kernel debugger returns the value of register 5.

Macros can be recursive and they exist until you end the debugging session or redefine them.

Accessing Memory

Using debugger commands, you can perform operations that enable you to examine and change memory addresses and registers.

When you access memory using a debugger command (such as **memwrite**), you can supply an address. Generally, you supply logical addresses in debugger commands. These logical addresses map on a one–to–one basis with physical addresses in the range from 0 to the size of the physical memory.

If you use a logical address when accessing (or modifying) memory, the system uses its page tables to map the logical address to a physical address and to validate the address. If you supply a physical address (using an option on the command line), you bypass the system table mapping and validation. All commands that access memory allow a "–p" option, which denotes a physical address. In this case, you must be sure that the address you supply is valid. Commands that access memory will fail if the requested address is not valid or is not currently resident.

By default, addresses are assumed to be in kernel (supervisor) space instead of in user space. However, when you access memory using a debugger command, you can supply an address in user space. All commands that access memory allow a "–u" option, which denotes a user space address; the user space address is then translated to a kernel space address. You can also translate a user space address to a kernel space address by using the ":" operator before the user space address. (Operators are described in "The Expression Parser" section later in this chapter.)

A Note About Debugger Modes

The debugger's **mode** command enables you to examine and set global debugger configuration options. For example, you can use the **mode** command to change the debugger's default number radix. The debugger is initialized to work with hexadecimal numbers. You can change to an octal or decimal radix by typing the following commands:

```
[0:#system#:1]> mode oct ⤶

[0:#system#:1]> mode dec ⤶
```

The debugger will display an "Illegal Expression" error message if you enter an address that is in the wrong radix. Note that the debugger's memory commands have a format option that enables you to override the default number radix for one command.

You can examine the debugger's current mode settings by typing the **mode** command with no arguments. The **mode** command is described in detail in Chapter 3.

The Expression Parser

When you enter a debugger command with an expression as an argument, the debugger's parser resolves the expression to a value (an address) and then the debugger command is performed.

Expression Format

A debugger expression can consist of integers, unary operators, binary operators, and symbols. Integers are constants in hexadecimal, octal, or decimal. The operators and the symbols that you can use in an expression are described in the sections below. You can use parentheses to group parts of expressions. An expression may not contain any whitespace.

Operators

You can use both unary and binary operators in expressions. Unary operators perform an operation on a single value, for example *operator value*. The valid unary operators are listed in Table 2–1 below.

Table 2–1. Unary Operators

Unary operator	Operation
!	This operator is the logical NOT operator. If *value* is non–zero, set it to zero. If *value* is zero, set it to 1. This operator is a prefix operator (the usage is **!***value*).
:	Interpret *value* as a user space address, and translate it to the underlying kernel address. This operator is a prefix operator (the usage is **:***value*). Note that you may receive unexpected results if you use this operator in a command that extends over a page boundary in memory.
#number	A 32–bit indirect addressing symbol through a location. This operator is a prefix operator. Example: **#0** indirects through the 32–bit value at location zero.
@number	A 16–bit indirect addressing symbol through a location. This operator is a prefix operator. Example: **@0** indirects through the 16–bit value at location zero.

Binary operators perform an operation on two values, for example *value1 operator value2*. The valid binary operators are listed in Table 2–2.

Table 2–2. Binary Operators

Binary Operator	Operation
+	Add *value1* and *value2*.
–	Subtract *value2* from *value1*.
*	Multiply *value1* and *value2*.
/	Divide *value1* by *value2*.
&	Logical AND *value1* and *value2*.
|	Logical OR *value1* and *value2*.
>	1 if *value1* > *value2* and 0 if *value1* ≤ *value2*.
<	1 if *value1* < *value2* and 0 if *value1* ≥ *value2*.
=	1 if *value1* = *value2* and 0 if *value1* != *value2*.
>=	1 if *value1* ≥ *value2* and 0 if *value1* < *value2*.
<=	1 if *value1* ≤ *value2* and 0 if *value1* > *value2*.

Symbols

You can use four kinds of symbols in expressions:

1. kernel symbols
2. debugger symbols
3. user–defined aliases
4. user–defined global variables

Kernel symbols are located in the symbol table and are part of the kernel image. Debugger symbols are specific to the debugger itself. User–defined aliases are macros that you create as described previously in the section "Assigning Aliases to Debugger Commands." User–defined global variables are values that you define using the debugger commands **global** and **set**, which are described in detail in Chapter 3.

Kernel Symbols

Kernel symbols correspond to the names of C language routines, functions, and global variables in the kernel code. For example, the following **memread** command uses a kernel symbol as an expression:

```
[0:#system#:1]> memread ldm_number_of_registered_disks ⤶
```

The expression parser has a shortcut to specify a kernel symbol—the caret (^). For example, a shortcut form of the above command is:

```
[0:#system#:1]> memread ldm_number_of_r^ ⤶
```

When you use the caret at the end of a kernel symbol, the expression parser takes the element as a regular expression and looks for a match in the symbol table. The match must be unique; if there is more than one match in the symbol table, you will get an error message. If the expression parser finds only one match, the expression is resolved to the symbol table value and the debugger command is performed on that value.

Debugger Symbols

The debugger has a set of built–in symbols that you can use during a debugging session. The debugger symbols are values that are generally useful to look at when debugging. Do not confuse debugger symbols with the kernel program's symbols. The debugger symbols you can use in an expression are described in Table 2–3.

Table 2–3. Debugger Symbols

Symbol	Evaluates to:
. (dot)	The value of the program counter (**view_pc**) when you entered the debugger. If you view an address, the view command sets the **view_pc**, and dot will evaluate to the new **view_pc**. Similarly, viewing up and down resets the **view_pc** and changes the evaluation of dot.
pc	The PC of the current virtual processor (VP).

Symbol	Evaluates to:
~ (tilde)	The name of the current routine. The tilde symbol takes the **view_pc** and truncates the *+nn*. In other words, the tilde is the address of the entry point to the current routine.
r0–r31	General register names.
cr2–cr20	Control register names.
sf0–sf20	The stack frame number after a trace.

For example, programmers often want to know the value of **pc**; they can find the value of **pc** by evaluating that debugger symbol as follows:

```
[0:#system#:1]> eval pc ↵
```

Suppose you are often interested in knowing the value of **pc** plus the contents of the third register. Instead of always typing **eval pc+r2**, you can define an alias to show you that value, as shown in this example:

```
[0:#system#:1]> mypc() eval pc+r2
[0:#system#:1]> mypc
    [octal]          [decimal]          [hex]
[ 000002534661 ]  [      702897 ]     [0x000AB9B1]

              [symbolic]: [deb_debugger_request+55]

[0:#system#:1]>
```

User–defined Global Variables

Two debugger commands, **global** and **set**, enable you to create 32–bit variables that may be used in expressions, which allows you to save values for later use. User–defined global variables contain expressions, such as debugger symbols, kernel symbols, and operators. Once set, you use a global variable as an expression in the debugger commands, much like you use a kernel symbol.

User–defined global variables look much like kernel symbols and, in effect, are a non–permanent extension of the kernel symbols. The details of setting global variables are shown in Chapter 3.

Resolving Expressions

As described above, an expression can consist of a combination of elements: kernel symbols, debugger symbols, operators, integers, and global variables. These elements are resolved, left to right in the command line, in the following order:

1. Debugger symbol values.
2. User–defined global variable values.
3. Kernel symbol values.
4. Integer constants.

If the debugger parser doesn't recognize any of the above elements in an expression, you will receive an error message. Note that a hexadecimal value beginning with a through f (inclusive) can be an ambiguous element; such a value could be resolved to either a variable name or a hexadecimal constant. Given the resolution order above, such an element would be resolved to a variable of that name, which takes precedence over the hexadecimal constant.

Getting Online Help

You can display a list of the debugger's command names and short–form names by typing the following commands at the debugger prompt:

```
[0:#system#:1]> help ↵
```

or

```
[0:#system#:1]> ? ↵
```

Figure 2–3 shows this help display.

```
[0:#system#:1]> help
Memory examination and modification:
          memread (mr), memwrite (mw), memsearch (ms), regsearch (rs),
          patdump (pd), view (vi), down (do), up
Expression evaluation:
          name (nm), translate (ts), eval (ev), global, set
Execution control:
          brk (b), delete (d), proceed (p), halt
Machine state:
          register (reg), control (ctl), cmmu
Kernel state:
          ltop, trace (tr), status, vp, focus (fo)
General:
          help, mode, print, printf

For help about a specific command, type "help <command_name>",
 then the New Line key. Example: "help reg" for the register command.
```

Figure 2–3 Command Name Help

You can display information about a specific command by typing the command name or the short–form name as an argument to help. Figure 2–4 shows the help information for the register (**reg**) command.

```
[0:#system#:1]> help reg

  reg [register_number [register_value] ]

The register command displays or modifies the contents of a general
register.

If <register_value> is specified, then the register named by
<register_name> will have <register_value> stored into it;
otherwise, the contents of <register_name> will be displayed.
The <register_name> is specified as a number in the range 0
through 31 or as a name in the range r0 through r31.  If no
<register_name> is specified, then the contents of all 32
general registers are displayed in order.
```

Figure 2–4 Help for a Specific Command

Exiting from the Debugger

The **proceed** (**p**) command enables you to exit from the debugger and return to the program that you were executing. To exit from the debugger, type the following command:

```
[0:#system#:1]>p ↵
```

NOTE: If you entered the debugger using Ctrl–P and you are using a workstation keyboard (for example, on an AViiON 300 Series station), you must press the Ctrl key after you invoke the **proceed** command to exit from the debugger (the complete sequence is **p**–New Line–Ctrl–New Line). This sequence will reset the workstation's keyboard driver and return to the kernel.Using this key sequence applies only when you are using a workstation keyboard and entered the debugger using Ctrl–P. If you are using an asynchronous terminal or if you are using a workstation keyboard but entered the debugger from a breakpoint (or at initialization), you can use just the **proceed** command.

End of Chapter

Chapter 3
Kernel Debugger Commands

This chapter is a command reference for the kernel debugger commands. If you are familiar with the commands and simply need the command format, Appendix A is a quick reference for the commands described in this chapter.

In this chapter, we explain the debugger commands in the following order:

- Memory examination and modification commands
- Expression evaluation commands
- Execution control commands
- Machine state commands
- Kernel state commands
- General commands

The first two sections describe generic debugger commands that provide memory accessing, access to the symbol table, and expression evaluation commands for the kernel debugger. The next three sections describe specific debugger commands that enable you to access an AViiON system's CPU state, set breakpoints, traceback the stack, and halt the debugger.

Memory Examination and Modification Commands

The memory examination and modification commands enable you to scan through memory locations and modify them selectively. All of the memory commands have the same interface. The defaults and meaning of some of the arguments may differ between individual commands. The format for the memory commands is:

command_name [*–options*] [*memory_address*] [*count*] [*format*]

where:

command_name
: is the name of a command. Valid memory command names are: **memread**, **memwrite**, **memsearch**, **regsearch**, **patdump**, **view**, **down**, and **up**, as described later in this section.

If you enter an invalid command name, the debugger will display the following error message:

`Command Unknown` [*name*]

options
: are given after the command name and are preceded with a dash. The options for the memory commands are:

–p specifies that the memory address is a physical address, not a logical address. This means the read or write physical routine will be called directly to read or write data.

–u specifies that the memory address is an address in user space, not kernel space.

–v is the verify option for the **memwrite** command. See the memory write command for more information.

–n turns off converting labels to their symbolic form while printing memory locations.

–l*n* enables you to specify the number, *n*, of elements to be printed across a line. The default is set by the format you choose.

You can group multiple options in a string that is preceded by a dash. For example, the options are grouped in the command line:

`[0:#system#:1]>` **memwrite –pv** ⤶

memory_address
: is an address, or an expression that evaluates to an address, to be used in the command. This argument is optional. If a memory address is not specified, the default is the current **view_pc**.

NOTE: For the **patdump** command, this argument is required and is a regular expression.

count is a number that specifies how many elements are to be operated on. This argument is optional. The number is assumed to be in the current output base. To specify a decimal number, place a period after the number. You can use the prefix “0x” to specify a hexadecimal number, but that is unnecessary if the mode is **hex**. To specify an octal number, the mode must be **octal**.

This argument may also be an expression that evaluates to a number. The expression elements may be specified as described above.

If the *count* argument is not specified, the default is determined by the command.

format specifies the format of the elements being examined. This argument is optional. If a format is not specified, the default is determined by the command.

The formats supported for the memory commands are listed in Table 3–1 below.

Table 3–1 Formats

Element Formats	Description
character **char** **c**	The memory location is an 8–bit character value.
instruction **i**	The memory location is the start of an instruction.
long **l**	The memory location is a 32–bit value printed in the current output base.
short **s**	The memory location is a 16–bit value printed in the current output base.
byte **b**	The memory location is an 8–bit value printed in the current output base.
decimal **dec** **d**	The memory location is a 16–bit decimal value.
longdec **ld** **D**	The memory location is a 32–bit decimal value.
octal **oct** **o**	The memory location is a 16–bit octal value.
longoct **lo** **O**	The memory location is a 32–bit octal value.
hexadecimal **hex** **h** **x**	The memory location is a 16–bit hexadecimal value.

Element Formats	Description
longhex **lhex** **lh** **lx** **H** **X**	The memory location is a 32–bit hexadecimal value.
ssym	The memory location is a 16–bit value expressed as a symbolic address.
sym	The memory location is a 32–bit value expressed as a symbolic address.
string **str**	The memory location is the start of a string, terminated by a null.
pte	The memory location is a page table entry. The entry is displayed with letters representing the bits that are on.
def	This is equivalent to specifying no format; the default format is used. Each memory command has its own default format.

The memory commands are described below. All examples assume that the radix is hexadecimal.

Memory Read (memread)

The command name for a memory read is **memread**; the short name is **mr**. The command format for **memread** is:

memread [–l*n*] [**–n**] [**–p**] [**–u**] [*memory_address*] [*count*] [*format*]

The **memread** command displays the values of memory locations, starting at the *memory_address* and continuing for the number of elements specified in *count.*

The current memory address is displayed at the beginning of the line. The values are displayed in the given *format*; the number of elements displayed on each line depends on the format selected.

If you use the *format* argument, you must also specify the *memory_address* and *count* arguments. For the **memread** command, the default format is long and the default count is 1. You can change the number of elements displayed on a line using the **–l** option.

The following command specifies a memory read starting at address 100 for a count of 20 in long format:

```
[0:#system#:1]> memread 100 20.1 ⤶
```

The following display shows how your screen might look after you enter this command.

```
[0:#system#:1]> memread 100 20. l
__.init.end+EC      :    80018221  C80004C7  80018221  C80004C5
__.init.end+FC      :    80018221  C80004C3  80018221  C80004C1
__.init.end+10C     :    80018221  C80004BF  80018221  C80004BD
__.init.end+11C     :    80018221  C80004BB  80018221  C80004B9
__.init.end+12C     :    80018221  C80004B7  80018221  C80004B5
[0:#system#:1]>
```

The following command also specifies a read starting at address 100 for a count of 20 in long format:

[0:#system#:1]> **memread –l1 100 20. l** ↵

The following display shows how your screen might look after you enter this command. This display shows the same information as in the previous screen, except the addresses are printed one entry per line as specified by **–l1**. You can use this technique to save the work of calculating the addresses.

```
[0:#system#:1]> memread -l1 100 20. l
__.init.end+EC      :    80018221
__.init.end+F0      :    C80004C7
__.init.end+F4      :    80018221
__.init.end+F8      :    C80004C5
__.init.end+FC      :    80018221
__.init.end+100     :    C80004C3
__.init.end+104     :    80018221
__.init.end+108     :    C80004C1
__.init.end+10C     :    80018221
__.init.end+110     :    C80004BF
__.init.end+114     :    80018221
__.init.end+118     :    C80004BD
__.init.end+11C     :    80018221
__.init.end+120     :    C80004BB
__.init.end+124     :    80018221
__.init.end+128     :    C80004B9
__.init.end+12C     :    80018221
__.init.end+130     :    C80004B7
__.init.end+134     :    80018221
__.init.end+138     :    C80004B5
[0:#system#:1]>
```

The **memread** command also accepts a pipe so that you can search for a string. The following command specifies a read starting at address 100 for a count of 20 in long format with a search of C3:

[0:#system#:1]> **memread 100 20. l | C3** ↵

The following display shows how your screen might look after you enter this command.

```
[0:#system#:1]> memread 100 20. 1 | C3
__.init.end+FC      :    80018221  C80004C3  80018221  C80004C1
[0:#system#:1]>
```

The **sym** and **ssym** formats are useful with the **memread** command to look at symbolic values in tables. For example, the following command specifies the **sym** format to display symbolic values in the system call table:

`[0:#system#:1]>` **memread sc_bcs_system_call_table A sym** ↵

The following display shows how your screen might look after you enter this command.

```
[0:#system#:1]> memread sc_bcs_system_call_table A sym
sc_bcs_system_call_table:      sc_invalid_system_call
sc_bcs_system_call_table+4:    access
sc_bcs_system_call_table+8:    brk
sc_bcs_system_call_table+C:    chdir
sc_bcs_system_call_table+10:   chmod
sc_bcs_system_call_table+14:   chown
sc_bcs_system_call_table+18:   chroot
sc_bcs_system_call_table+1C:   close
sc_bcs_system_call_table+20:   execve
sc_bcs_system_call_table+24:   _exit
[0:#system#:1]>
```

You can use the **memread** command to display the processor's context block. For example, when the debugger takes a breakpoint at **sc_panic**, enter the command:

`[0:#system#:1]>` **mr #sc_base^ 64.1** ↵

This command will display a context block. The information in a context block is described in Appendix B.

Memory Write (memwrite)

The command name for memory write is **memwrite**; the short name is **mw**. The command format for **memwrite** is:

memwrite [–n] [–p] [–u] [–v] *[memory_address] [count] [format]*

The **memwrite** command enables you to view and modify memory locations one at a time. The modification starts at *memory_address* and continues until either *count* elements have been displayed or you enter "**q**" as a value.

Memory elements must be modified in the *format* specified. If you use the *format* argument, you must also specify the *memory_address* and *count* arguments.

For the **memwrite** command, the default format is long and the default count is 1.

Memory write displays the element at the memory address, in the format specified, followed by a prompt (a right angle bracket (>)). If you want to change the value of the element, enter the new value at the prompt and press the New Line key. The valid responses are:

q Exit memory write.

NOTE: If you want to enter q as a character (rather than q to exit), you must enter the value in a format other than character format.

^ Leave this location unchanged, but display the previous element for modification.

NOTE: If you want to enter the caret as a character (rather than display the previous element), you must enter the value in a format other than character format.

New Line Leave this location unchanged, but display the next element for modification.

Expression Resolve the expression, expecting the format specified, and write the results into the memory location. If the verify flag (–v) is set, redisplay this element. Otherwise display the next element for modification.

The following command shows an example of a write starting at address 0 in instruction format:

`[0:#system#:1]>` **memwrite 0 1 i** ⤶

The following display shows how your screen might look after you enter this command. Pressing the New Line key at the right angle bracket prompt leaves the location unchanged.

```
[0:#system#:1]> memwrite 0 1 i
sc_exc_vector       :  or        r0, r0, r0 >
[0:#system#:1]>
```

The following command specifies a write starting at address 0 in long format:

`[0:#system#:1]>` **memwrite 0 9 l** ⤶

The following display shows how your screen might look after you enter this command. Pressing the New Line key at the right angle bracket prompt leaves the location unchanged and displays the next element; typing a caret (^) followed by the New Line key leaves the location unchanged and displays the previous element; and typing **q** followed by the New Line key quits the memory write.

```
[0:#system#:1]> memwrite 0 9 1
sc_exc_vector      :   F4005800 >
sc_exc_vector+4    :   C0000406 >
sc_exc_vector+8    :   C400040D > ^
sc_exc_vector+4    :   C0000406 >
sc_exc_vector+8    :   C400040D > q
[0:#system#:1]>
```

Memory Search (memsearch)

The command name for an memory search is **memsearch**; the short name is **ms**. The command format for **memsearch** is:

memsearch [**–l***n*] [**–n**] [**–p**] [**–u**] [*memory_address*] [*count*] [*format*]

The **memsearch** command searches through memory for a given value in a given *format*. The search starts at *memory_address* and continues for a maximum of *count* elements.

If you use the *format* argument, you must also specify the *memory_address* and *count* arguments. For the **memsearch** command, the default format is long and the default count is 1.

The debugger will prompt you for a search value. You must enter the search value in the format specified. If a value matching the search value is found, a view (see the **view** command below) is performed at the location where the match occurred.

The following command specifies a search starting at address 0 for a count of 20 in byte format:

`[0:#system#:1]>` **memsearch 0 20. b ↵**

At the `Enter Search Value` prompt, enter the value that you want to find, for example **c0**, and press the New Line key. The search value **c0** is a hexadecimal value in byte format.

The following display shows how your screen might look after you enter the **memsearch** command and search value above.

```
[0:#system#:1]> memsearch 0 20. b
Enter Search Value:  c0
sc_exc_vector        :    F4
sc_exc_vector+1      :    00
sc_exc_vector+2      :    58
sc_exc_vector+3      :    00

sc_exc_vector+4      :    C0

sc_exc_vector+5      :    00
sc_exc_vector+6      :    04
sc_exc_vector+7      :    06
sc_exc_vector+8      :    C4
sc_exc_vector+9      :    00
sc_exc_vector+A      :    04
sc_exc_vector+B      :    0D
sc_exc_vector+C      :    80
[0:#system#:1]>
```

Regular Expression Search (regsearch)

The command name for a regular expression search is **regsearch**; the short name is **rs**. The command format for **regsearch** is:

regsearch [**–l***n*] [**–n**] [**–p**] [**–u**] [*memory_address*] [*count*] [*format*]

The **regsearch** command works like the **memread** command, except that **regsearch** pipes the output through a regular expression search. If a match is found, the **view_pc** is set to the location of the match and a view is performed.

The debugger prompts you for the regular expression to use in the search. The search starts at *memory_address* and continues for a maximum of *count* elements.

If you use the *format* argument, you must also specify the *memory_address* and *count* arguments. For the **regsearch** command, the default format is long and the default count is 1.

When prompted for the regular expression, you must enter the search value. If a value matching the regular expression is found, a view (see the **view** command below) is performed at the location where the match occurred.

The following command specifies a regular expression search starting at the **view_pc** (the dot debugger symbol) for a count of 1000 in instruction format:

`[0:#system#:1]>` **regsearch . 1000. i** ⤶

When you enter this command, you will be prompted to enter a regular expression. For this example, enter the search value **sc_panic.*** followed by the New Line key at the `Regular Expression to Search For` prompt.

The following display shows how your screen might look after you enter the **regsearch** command and the regular expression above.

```
[0:#system#:1]> regsearch . 1000. i
Regular Expression to Search For:  sc_panic.*
sc_panic+8          :  st           r30, r31, 20
sc_panic+C          :  addu         r30, r31, 20
sc_panic+10         :  or.u         r3, r0, FFA0
sc_panic+14         :  or           r4, r0, r0
sc_panic+18         :  or           r5, r0, r0
sc_panic+1C         :  or           r6, r0, r0

sc_panic+20         :  bsr.n        sc_panic_with_message

sc_panic+24         :  or           r7, r0, r0
sc_panic+28         :  subu         r31, r30, 20
sc_panic+2C         :  ld           r1, r31, 24
sc_panic+30         :  ld           r30, r31, 20
sc_panic+34         :  jmp.n        r1
sc_panic+38         :  addu         r31, r31, 28

[0:#system#:1]>
```

Pattern Dump (patdump)

The command name for a pattern dump is **patdump**; the short name is **pd**. The command format for **patdump** is:

patdump [**–l***n*] [**–n**] [**–p**] [**–u**] *memory_address* [*format*]

The pattern dump has a similar interface to the **memread** command. The **patdump** command takes a regular expression rather than an expression for the *memory_address*. Note that the regular expression is required. The **patdump** command then searches the symbol table for all matches to the regular expression. If a match occurs, the **memread** command is called using the arguments to the **patdump** command, with the symbol found in the search replacing the regular expression. The *count* argument is not used with the **patdump** command.

The pattern dump command enables you to look at a group of locations that can be described with a regular expression. For example, this command is useful for dumping a set of meters or counts that have a similar name.

For example, you could enter the following **patdump** command:

`[0:#system#:1]>` **pd vm.*count** ↵

The following display shows how your screen might look after entering this command.

```
[0:#system#:1]> pd vm.*count
vm_get_accounting_memory_usage              :     67FF0038
vm_resource_accounting_lock                 :     00000000
vmc_available_frame_resource_counter        :     0000021B
vmc_low_on_available_frames_eventcounter    :     000053CA
vmc_mft_eventcounter_table                  :     001CCE68
vmfs_decrement_reference_count              :     67FF0028
vmfs_increment_reference_count              :     67FF0028
[0:#system#:1]>
```

Note that the **patdump** output makes no distinction between code and data.

View (view)

The command name for view is **view**; the short name is **vi**. The command format for **view** is:

view [**–l***n*] [**–n**] [**–p**] [**–u**] [*memory_address*] [*count*] [*format*]

The **view** command is similar to the **memread** command, but **view** displays elements differently. The **view** command is used to display the element at the *memory_address* surrounded by six elements on either side of the memory address. This command is useful to see the neighboring instructions when looking at code in instruction mode. The default format is instruction mode. The *count* argument is not used and may be ignored. The default memory address is the **view_pc**. If you provide a memory address to the **view** command, that memory address becomes the new **view_pc**.

The following command specifies a view on **sc_panic**:

`[0:#system#:1]>` **view sc_panic** ↵

The following display shows how your screen might look after you enter this command.

```
[0:#system#:1]> view sc_panic
sc_panic_with_message+DC:  ld          r30, r31, 38
sc_panic_with_message+E0:  ld.d        r18, r31, 30
sc_panic_with_message+E4:  ld.d        r16, r31, 28
sc_panic_with_message+E8:  ld.d        r14, r31, 20
sc_panic_with_message+EC:  jmp.n       r1
sc_panic_with_message+F0:  addu        r31, r31, 40

sc_panic            :  subu        r31, r31, 28

sc_panic+4          :  st          r1, r31, 24
sc_panic+8          :  st          r30, r31, 20
sc_panic+C          :  addu        r30, r31, 20
sc_panic+10         :  or.u        r3, r0, FFA0
sc_panic+14         :  or          r4, r0, r0
sc_panic+18         :  or          r5, r0, r0
[0:#system#:1]>
```

View Down (down)

The command name to view down is **down**; the short name is **do**. The command format for **down** is:

down

The **down** command increments the **view_pc** such that sequential executions of this command will produce a continuous listing of elements.

To view down, type:

```
[0:#system#:1]> down ↵
```

The following screen shows a view down from the previous **view** screen.

```
[0:#system#:1]> down
sc_panic+1C          :  or           r6, r0, r0
sc_panic+20          :  bsr.n        sc_panic_with_message
sc_panic+24          :  or           r7, r0, r0
sc_panic+28          :  subu         r31, r30, 20
sc_panic+2C          :  ld           r1, r31, 24
sc_panic+30          :  ld           r30, r31, 20

sc_panic+34          :  jmp.n        r1

sc_panic+38          :  addu         r31, r31, 28
sc_operator_shutdown:   subu         r31, r31, 28
sc_operator_shutdown+4:   st           r1, r31, 24
sc_operator_shutdown+8:   st           r30, r31, 20
sc_operator_shutdown+C:   addu         r30, r31, 20
sc_operator_shutdown+10:    or.u         r2, r0, 10
[0:#system#:1]>
```

View Up (up)

The command name to view up is **up**. The command format for **up** is:

up

The **up** command decrements the **view_pc** such that sequential executions of this command will produce a continuous listing of elements.

To view up, type:

`[0:#system#:1]>` **up** ↵

The following screen shows a view up from the previous screen.

```
[0:#system#:1]> up
sc_panic_with_message+DC:  ld         r30, r31, 38
sc_panic_with_message+E0:  ld.d       r18, r31, 30
sc_panic_with_message+E4:  ld.d       r16, r31, 28
sc_panic_with_message+E8:  ld.d       r14, r31, 20
sc_panic_with_message+EC:  jmp.n      r1
sc_panic_with_message+F0:  addu       r31, r31, 40

sc_panic            :  subu       r31, r31, 28

sc_panic+4          :  st         r1, r31, 24
sc_panic+8          :  st         r30, r31, 20
sc_panic+C          :  addu       r30, r31, 20
sc_panic+10         :  or.u       r3, r0, FFA0
sc_panic+14         :  or         r4, r0, r0
sc_panic+18         :  or         r5, r0, r0
[0:#system#:1]>
```

Expression Evaluation Commands

The expression evaluation commands enable you to search symbol tables; display an expression's value in octal, decimal, hexadecimal, or symbolic format; and set variables to use in expressions.

Print Symbol Table Entry (name)

The command format to print a symbol table entry is:

name [*regular_expression*]

The **name** command searches the symbol table for a match to *regular_expression.* If a match is found, the symbol is printed along with its value and symbol type. The default *regular_expression* is .*, which matches all symbols. The short name for **name** is **nm**.

For example, if you know a routine name starts with **vm** and contains **count**, you can list all routines with that combination by typing:

`[0:#system#:1]>` **name vm.*count** ⤶

The following display shows how your screen might look after you enter this command.

```
[0:#system#:1]> name vm.*count
[  0] vm_get_accounting_memory_usage     0006406C     external
[  0] vm_resource_accounting_lock        00107124     external
[  0] vmc_available_frame_resource_counter  00107124     external
[  0] vmc_low_on_available_frames_eventcounter  001071A4     external
[  0] vmc_mft_eventcounter_table         00107188     external
[  0] vmfs_decrement_reference_count     000D5E54     external
[  0] vmfs_increment_reference_count     000D5E10     external
[0:#system#:1]>
```

Translate an Expression Value to a Symbol (translate)

The command format to translate an expression value to a symbol is:

translate [*expression*]

The **translate** command evaluates the *expression* given and converts it to a symbolic value, using the current set of symbol tables. If a relevant symbol cannot be found, the value is converted, based on the current radix, and the resulting string is printed. The default *expression* is the null expression. The short name for **translate** is **ts.**

The following command translates **sc_panic+50**:

`[0:#system#:1]>` **translate sc_panic+50** ⤶

The following display shows how your screen might look after you enter this command.

```
[0:#system#:1]> translate sc_panic+50
sc_operator_shutdown+14
[0:#system#:1]>
```

This command is similar to the **eval** command below.

Expression Evaluation (eval)

The command format for evaluating an expression is:

eval *expression*

The **eval** command evaluates an *expression* and displays the result in octal, decimal, hex, and symbol formats. The short name for **eval** is **ev.**

The following command evaluates **sc_panic+50**:

`[0:#system#:1]>` **eval sc_panic+50** ⤶

The following display shows how your screen might look after you enter this command. This display shows the **eval** output for the same expression as shown in the **translate** display above. Notice that the **eval** command displays more information about the expression than the **translate** command.

```
[0:#system#:1]> eval sc_panic+50
    [octal]           [decimal]          [hex]
[ 000001640214 ]  [      475276 ]     [0x0007408C]

             [symbolic]: [sc_operator_shutdown+14]

[0:#system#:1]>
```

Symbols can be used in an expression with the **eval** command. These symbols are listed in "The Expression Parser" section in Chapter 2. One of these symbols, **pc**, evaluates to the current VP's program counter. The following command evaluates **pc**:

```
        [0:#system#:1]> eval pc ⤶
```

The following display shows how your screen might look after you enter this command.

```
[0:#system#:1]> eval pc
    [octal]           [decimal]          [hex]
[ 000002534660 ]  [      702896 ]     [0x000AB9B0]

             [symbolic]: [deb_debugger_request+54]

[0:#system#:1]>
```

Global (global)

The **global** command enables you to create a 32-bit variable that may be used in expressions. This allows you to save values and arguments for later use. A global variable will override the evaluation of a symbol of the same name. The format for the **global** command is:

> **global** [{*name expr* | **-d** *name* ...}]

You can initialize a global variable by specifying a *name* and an *expression*. If you do not provide a *name*, the current list of global names with their respective values is printed. The **-d** option deletes a global variable.

For example, you can create a global variable named **scl** to save the value of **sc_panic** using the following command:

```
        [0:#system#:1]> global scl #sc_panic ⤶
```

The next display shows **sc_panic** and the value saved in the global variable **scl**.

```
[0:#system#:1]> memread sc_panic
sc_panic          :      67FF0028
[0:#system#:1]> global scl #sc_panic
[0:#system#:1]> global
            scl:  67FF0028
[0:#system#:1]>
```

Later you can use the saved value. For example, if you changed the value of **sc_panic** and want to re–enter the old, saved value, you can use the saved global variable in the **memwrite** command, as shown in the following display.

```
[0:#system#:1]> memwrite sc_panic
sc_panic          :      67FF0020 > scl
[0:#system#:1]> memread sc_panic
sc_panic          :      67FF0028
[0:#system#:1]>
```

Set (set)

The format for the **set** command is:

set *name expression*

The **set** command enables you to set a global variable, *name*, to the evaluation of an *expression*.

Execution Control Commands

The execution control commands enable you to set and delete breakpoints, continue execution of the system, and halt the processor.

Set Breakpoint (brk)

The **brk** command sets or modifies a breakpoint. By setting a breakpoint, you specify that the debugger should take control of the kernel at a specified place in the code. The debugger will then give you a prompt and wait for user input. You can set a breakpoint with conditions that must be met for the breakpoint to be taken; the breakpoint is passed if the conditions are not met.

For example, you can set a breakpoint with a specified count of ten. This breakpoint will be passed nine times before the breakpoint is actually taken. The debugger increments its counter each time the breakpoint is passed. When the debugger hits the breakpoint the tenth time, the breakpoint is taken (the debugger takes control of the kernel, gives you a prompt, and waits for input).

The command format to set a breakpoint is:

brk [**–t**] [**–l**] [*addr_expression* | **–m** *bp_num*] [**–e** [*expression*]] [**–c** *count*]

The **–t** option sets a temporary breakpoint; the breakpoint is deleted as soon as it is taken.

The **–l** option prints a notification when a breakpoint is passed. You can pass a breakpoint using the **–c** *count* and **–e** *expression* arguments as described below. By default, a notification is given when a breakpoint is taken.

The address expression, *addr_expression*, sets a new breakpoint; the **–m** option specifies the breakpoint number, *bp_num*, to modify an existing breakpoint.

A **–e** *expression* and/or a **–c** *count* can be associated with the breakpoint. The breakpoint will not be taken unless the count is zero and the expression is TRUE, where TRUE means non–zero. The count is decremented each time the breakpoint is executed (but never below zero). To clear the expression on the breakpoint, use the **–e** option without an expression. The default count is 1 and the default expression is always true.

If no arguments are given, then all of the current breakpoints are displayed. The short name for **brk** is **b**.

For example, to set a breakpoint at the **lseek** function with a count and a notification when the breakpoint is passed, type the following command at the debugger prompt:

```
[0:#system#:1]> brk lseek –l –c 3 ↵
```

The following screen shows an example of setting this breakpoint and displaying all of the breakpoints that are set.

```
[0:#system#:1]> brk lseek -l -c 3
[1]  lseek                subu       r31, r31, 40 ; [L] c[3]
[0:#system#:1]> brk
[0]  sc_panic             subu       r31, r31, 28 ;
[1]  lseek                subu       r31, r31, 40 ; [L] c[3]

[0:#system#:1]>
```

To take the above breakpoint, you can type **ps** at the system prompt. The following screen shows an example of the notification messages:

```
% ps
[1]  lseek                subu       r31, r31, 40 ; [L] c[2]
[1]  lseek                subu       r31, r31, 40 ; [L] c[1]
BREAKPOINT [1] TAKEN.

lseek                : subu       r31, r31, 40

[14]  [Eligible]          [ps]
[0:#system#:1]>
```

As shown above, two notifications show that the breakpoint has been passed, then a notification shows that the breakpoint has been taken. When the breakpoint is taken, you enter the debugger and receive the debugger prompt.

Delete Breakpoint (delete)

The command format to delete a breakpoint is:

> **delete** {*bp_num* ... | **–a**}

The **delete** command deletes the breakpoints specified by the breakpoint number(s), *bp_num*, from the list of breakpoints. The **–a** option deletes all of the breakpoints. The short name for **delete** is **d**.

Proceed (proceed)

The **proceed** command enables you to exit from the debugger. The command format to proceed is:

> **proceed**

The short name for **proceed** is **p**. The **proceed** command will cause the debugger to return and continue execution of the system.

NOTE: If you entered the debugger using Ctrl–P and you are using a workstation keyboard (for example, on an AViiON 300 Series station), you must press the Ctrl key after you invoke the **proceed** command to exit from the debugger (the complete sequence is **p**–New Line–Ctrl–New Line). This sequence will reset the workstation's keyboard driver and return to the kernel. Using this key sequence applies only when you are using a workstation keyboard and entered the debugger using Ctrl–P. If you are using an asynchronous terminal or if you are using a workstation keyboard but entered the debugger from a breakpoint (or at initialization), you can use just the **proceed** command.

Halt (halt)

The **halt** command terminates the debugger, stops all of the machine's job processors (JPs), and returns control to the System Control Monitor (SCM). The command format to halt is:

> **halt**

CAUTION: The **halt** command irrevocably kills the kernel. The **halt** command also prevents any buffers from being flushed back to disk, which may require running **fsck** to check file systems and correct inconsistencies. You should use the command only when you are sure that you are finished debugging and want to boot a new kernel from the SCM.

Machine State Commands

The machine state commands allow you to access registers and display the state of the CMMU chips.

Display/Modify Register Contents (register and control)

The command format to display or modify a general register's contents is:

register [*general_register_name* [*register_value*]]

The **register** command displays or modifies the contents of a general register. The *general_register_name* is specified as 0–31 or r0–r31. If *register_value* is specified, the register value is stored in the register. If no arguments are given, all of the registers are displayed. The short name for **register** is **reg**.

To display the contents of register 30, type:

```
[0:#system#:1]> reg r30 ↵
```

To display the contents of all general registers, you type:

```
[0:#system#:1]> reg ↵
```

The following display shows how your screen might look after you type these commands.

```
[0:#system#:1]> reg r30
r30:  FF801CD0
[0:#system#:1]> reg
r 0:  00000000  r 1:  00000000  r 2:  00000001  r 3:  00000000
r 4:  FF801DCC  r 5:  00000000  r 6:  00100030  r 7:  00000000
r 8:  00000011  r 9:  00000000  r10:  00000668  r11:  00000000
r12:  000AB98C  r13:  00100000  r14:  00000010  r15:  00000000
r16:  FFF82800  r17:  00000002  r18:  0009F780  r19:  00000000
r20:  00000000  r21:  00000000  r22:  00000000  r23:  FF802E38
r24:  00000001  r25:  00000002  r26:  00000000  r27:  00000000
r28:  00000000  r29:  00000000  r30:  FF801CD0  r31:  FF801CA0
[0:#system#:1]>
```

The command format to display or modify a control register's contents is:

control [*control_register_name* [*register_value*]]

The **control** command displays or modifies the contents of a control register. The *control_register_name* is specified as cr2–cr20, or by its given name (for example, **sxip** or **vbr**). If *register_value* is specified, the register value is stored in the register. If no arguments are given, all of the registers are displayed. The short name for **control** is **ctl**.

To display the contents of the **snip** control register, type:

```
        [0:#system#:1]> ctl snip ↵
```

To display the contents of all control registers, you type:

```
        [0:#system#:1]> ctl ↵
```

The following display shows how your screen might look after you type these commands.

```
[0:mount:10]> ctl snip
cr 5 [  snip]: 000CCCD2   [read+4 [v]]

[0:mount:10]> ctl
cr 2 [  tpsr]: 900003F0   [ SUP BIGI C MXM FLOAT INTEN]
cr 3 [  ssbr]: 900003F0
cr 4 [  sxip]: 000CCCCE   [read [v]]
cr 5 [  snip]: 000CCCD2   [read+4 [v]]
cr 6 [  sfip]: 000CCCD6   [read+8 [v]]
cr 7 [   vbr]: 00000000
cr 8 [   tr0]: 00000000   [INVALID]
cr 9 [   dr0]: 00000000
cr10 [   ar0]: 00415700   [415700]
cr11 [   tr1]: 00000000   [INVALID]
cr12 [   dr1]: 00000000
cr13 [   ar1]: 00415700   [415700]
cr14 [   tr2]: 00000000   [INVALID]
cr15 [   dr2]: 00000000
cr16 [   ar2]: 0012C1B8   [vm_memory_pool+278]
cr17 [   sr0]: 0007BFA4
cr18 [   sr1]: 00416000
cr19 [   sr2]: 00000000
cr20 [   sr3]: 900003F0
[0:mount:10]>
```

Display CMMUs (cmmu)

The format for the **cmmu** command is:

cmmu

The **cmmu** command displays the state of the virtual address-based management chips, the CMMUs.

To display the state of the CMMU chips, type:

```
[0:#system#:1]> cmmu
```

The following display shows what information is displayed when you type this command.

```
[0:#system#:1]> cmmu
Instruction Cmmu    |    Data Cmmu
   [FFF01000]       |    [FFF00000]
 id    :   1A70000  |   id    :   A70000
 scr   :      8000  |   scr   :     C000
 ssr   :         9  |   ssr   :        9
 sadr  :  FF804000  |   sadr  : FF804000

 sctr  :      8000  |   sctr  :     C000

 sapr  :    1AB001  |   sapr  :   1AB001
 uapr  :         0  |   uapr  :        0

[0:#system#:1]>
```

(For more information, see the *MC88200 Cache/Memory Management Unit User's Manual.*)

Kernel State Commands

The kernel state commands enable you to translate a logical address to its physical address, traceback a process' kernel stack, display the status of JPs and VPs, and focus on a VP.

Logical to Physical Address (ltop)

The format for the **ltop** command is:

ltop *logical_address*

The **ltop** command translates the specified *logical_address* to its physical address. This command is useful for determining if an address is valid.

For example, you can see the logical to physical mapping of **vm_my_process_ptr** by typing the following command:

```
[0:#system#:1]> ltop vm_my_process_ptr
```

The following display shows your screen might look after entering this command.

```
[0:mount:10]> ltop vm_my_process_ptr
Logical: [FF8026C8]  Physical: [3FA6C8]
[0:mount:10]>
```

Traceback a Process' Kernel Stack (trace)

The command format to traceback a process' kernel stack is:

trace [{*vp_num* ... | *vp_num1–vp_num2* | *vp_num–* | *–vp_num* | –}]

The **trace** command displays a traceback of a process' kernel stack. If you specify a virtual process number, *vp_num*, a focus is done on that virtual process before the traceback is performed. If you do not provide an argument, the traceback is performed on the currently focused virtual process. You can also trace a range of VPs using *vp_num1–vp_num2*, *vp_num–*, or *–vp_num*. The hyphen (–) traces all VPs. The short name for **trace** is **tr**.

The traceback displays the return PC and the Frame Pointer for each frame. Arguments are not displayed, since their location is unknown.

The **trace** command lists the routines that a VP has called. The trace stack shows the current PC first, which is the current routine, and traces back through the routines that were called. The first routine can be a page fault, a data exception, or a system call. If you enter the debugger from user space, you start with an exception.

The following display shows an example of a trace of VP 1.

```
[0:#system#:1]> trace

Stack Trace:  []  ID [1]
[1][  0] [deb_debugger_request+54               ]  fp:[FF801CD0]
[1][  1] [sc_enter_debugger+20                  ]  fp:[FF801CF8]
[1][  2] [ts_syscon_input_character+78          ]  fp:[FF801D28]
[1][  3] [ts_gdc_con_get_character+54           ]  fp:[FF801D50]
[1][  4] [ts_kdb_service_interrupt+80           ]  fp:[FF801D90]
[1][  5] [a_io_interrupt_handler+A0             ]  fp:[FF801DC0]
[1][  6] [sc_handle_non_serial_exceptions+8C ]  fp:[FF801E08]
[1][  7] [sc_common_exception_path+50           ]  fp:[FF801E58]
[1][  8] [vp_initialize_vp_mgr+2C4              ]  fp:[FF801E80]
[1][  9] [vp_initialize_subsystem+14            ]  fp:[FF801EA8]
[1][ 10] [init_initialize+20                    ]  fp:[FF801ED0]
[1][ 11] [init_start_system+80                  ]  fp:[FF801ED8]

[0:#system#:1]>
```

The first set of brackets in the trace output contains the VP number. The second set of brackets contains the stack frame number. The third set of brackets shows the routine name. The last set of brackets contains frame pointers for the different routines.

After you have done a trace on a VP, you can use the stack frame number as an argument to the **view** command. For example, the following display shows a view on stack frame number 11 of the previous trace example.

```
[0:#system#:1]> vi sf11
init_initialize+8    :  st          r30, r31, 20
init_initialize+C    :  addu        r30, r31, 20
init_initialize+10   :  bsr         sc_initialize_subsystem
init_initialize+14   :  bsr         sc_enter_debugger
init_initialize+18   :  bsr         uc_initialize_subsystem
init_initialize+1C   :  bsr         vm_initialize_subsystem

init_initialize+20   :  bsr         vp_initialize_subsystem

init_initialize+24   :  bsr         io_initialize_subsystem
init_initialize+28   :  bsr         su_initialize_subsystem
init_initialize+2C   :  bsr         sfm_initialize_subsystem
init_initialize+30   :  bsr         vp_are_interrupts_disabled
init_initialize+34   :  mask        r2, r2, FF
init_initialize+38   :  bcnd        ne0, r2, init_initialize+48

[0:#system#:1]>
```

You can trace more than one VP or specify a range of VPs. The following command lines are equivalent:

```
[0:#system#:1]> trace 0 1 2 3 ↵

[0:#system#:1]> trace 0–3 ↵

[0:#system#:1]> trace –3 ↵
```

You can also trace all VPs (by using just the hyphen) or trace all VPs starting with a particular VP. The first command line below shows how to trace all VPs, and the second shows how to trace VP 4 and all VPs thereafter:

```
[0:#system#:1]> trace – ↵

[0:#system#:1]> trace 4– ↵
```

You do not need to be focused on a VP to perform a trace of that VP. You can perform a trace on VP 2 when you are focused on VP 1. However, once a trace is performed, you are focused on the last VP specified in the **trace** command.

The following display shows an example of a trace of two VPs.

```
[0:mount:10]> tr 7 10

Stack Trace:  []  ID [7]

[7][  0] [vp_suspend_me+40                    ]  fp:[FF801DC0]
[7][  1] [vp_await_ec+1E4                     ]  fp:[FF801E08]
[7][  2] [su_demon+B4                         ]  fp:[FF801E80]
[7][  3] [init_initialize+14C                 ]  fp:[FF801ED0]
[7][  4] [init_start_system+80                ]  fp:[FF801ED8]

Stack Trace:  [mount]  ID [10]

[10][  0] [read                                ]  fp:[FF801EB8]
[10][  1] [sc_common_exception_path+44         ]  fp:[FF801EF8]

[0:mount:10]>
```

Display Status (status)

The format of the **status** command is:

status

The **status** command displays the status of each job processor (JP). This command is used mainly to determine if other processors entered the debugger and to show why the processors entered.

To display the status of the JP(s), type:

[0:#system#:1]> **status** ⤶

The following display shows what information is displayed when you type this command.

```
[0:#system#:1]> status
JP#0  VP:  1  Context:  [0013004C]   ENTER
[0:#system#:1]>
```

Display Virtual Processor (vp)

The format for the **vp** command is:

vp [**–a**] [*vp_number* ...]

The **vp** command displays the status of one or more VPs. The **–a** option gives a more verbose description. If no arguments are given, every virtual process that is bound is displayed.

To display the status of several VPs, you can use a command such as:

```
[0:#system#:1]> vp 8 9 11 19 20 ⤶
```

The following display shows how your screen might look after you type this command. The last two columns show the elapsed time and the processor time.

```
[0:#system#:1]> vp 8 9 11 19 20
[  8]   [Bound_Stopped]     [init]     [40.6]   [20.9]
[  9]   [Eligible]          [-csh]     [ 1.3]   [ 0.2]
[ 11]   [Bound_Stopped]     [update]   [ 3.8]   [ 0.2]
[ 19]   [Bound_Stopped]     [mountd]   [ 0.7]   [ 0.0]
[ 20]   [Bound_Stopped]     [nfsd]     [ 2.2]   [0.5]
[0:#system#:1]>
```

To see the verbose description of VP 8, use the command:

```
[0:#system#:1]> vp –a 8 ⤶
```

The following display shows how your screen might look after you enter this command.

```
[0:#system#:1]> vp -a 8
[  8]  [Bound_Stopped]     [init]     [40.6]   [20.9]
Address   :  0x7F001180   Suspend Count:     0
extension:  0x7F018000   next_ptr: 0xFFA00000
priority: 16584    Eff priority: 16584
VPID:      8    Allowed Set: 0xFFFF

[0:#system#:1]>
```

Focus on a Virtual Processor (focus)

The format for the **focus** command is:

focus [*vp_number*]

The **focus** command changes the address translation in the debugger to make it look as if the virtual process specified by *vp_number* was running. You may trace, look at per–process variables, and look at the machine state for that virtual process. If you do not provide an argument, the current virtual process will be used. The short name for **focus** is **fo**. Note that the address translation (atu) mode on the debugger must be turned on for **focus** to work.

To focus on VP 8, use the command:

```
[0:#system#:1]> fo 8 ⤶
```

The following display shows how your screen might look after you enter this command.

```
[0:#system#:1]> fo 8
[  8]  [Bound_Stopped]     [init]     [40.6]   [20.9]
[0:init:8]>
```

Notice that the prompt now indicates that you are focused on VP 8, the **init** program. As with the **vp** command, the last two columns show the elapsed time and the processor time.

General Commands

Help (help)

The **help** command displays information about a command. The format for the **help** command is:

help [*command_name*]

If you invoke the **help** command without arguments, the complete list of supported commands is printed, separated by subsystem. If a *command_name* is given, the help message for that command is printed. The short name for **help** is ? (a question mark). Examples of the **help** command are shown in Chapter 2.

Mode (mode)

The **mode** command enables you to change parameters for the debugger. The format for the **mode** command is:

mode [**oct** | **hex** | **dec**] [**er** {**on**|**off**}] [**atu** {**on**|**off**}] [**verbose** *n*] [**io** {**duart** | **prom**}]

Using the **mode** command, you set the following options:

- Set the radix to octal, hexadecimal, or decimal.
- Turn **editread** (**er**) on or off.
- Turn address translation (**atu**) on or off.
- Set the verbose mode for the debugger. (The verbose mode is generally only used by Data General systems engineers.)
- Set the console (**io**) to either the machine's PROM (**prom**) or an asynchronous terminal attached to the machine's serial port (**duart**).

If you invoke the **mode** command without arguments, the current settings for the parameters are displayed. To see the current settings, type:

```
[0:#system#:1]> mode ↵
```

The following display shows what information is displayed when you type this command.

```
[0:#system#:1]> mode
base    : hex
er      : on
verbose : 0
atu     : on
io      : prom
[0:#system#:1]>
```

The Prom notation indicates that you are communicating with the kernel through the machine's PROM. If you have an asynchronous terminal attached to your machine's serial port, you can use the **mode** command to tell the debugger to use that port by typing the following command line:

```
[0:#system#:1]> mode io duart ↵
```

If you change the console to the asynchronous terminal, you would then see Duart instead of Prom when you typed the **mode** command with no arguments.

CAUTION: If you do not have an asynchronous terminal connected to your machine, do not enter the **mode io duart** command—you will not be able to communicate with the debugger from your workstation keyboard, and you will have to reboot your machine.

Print (print and printf)

The **print** command provides simple **printf** capabilities. The format for the **print** command is:

print [*expression* ...]

You can use this command to print the evaluation of an *expression*. The expression is printed in the current base. A new line is printed at the end of the print.

Also, using **print** is an easy way to print two or three registers, instead of all 32. (The **reg** command will not enable you to do this). For example, you can print registers 12, 30, and 31 using the following command:

`[0:#system#:1]>` **print r12 r30 r31** ⤶

The following display shows an how your screen might look after entering the command above.

```
[0:#system#:1]> print r12 r30 r31
 AB98C  FF801CA0  FF801CA0
[0:#system#:1]>
```

The **printf** command enables you to print strings in a particular format. This command converts, formats, and prints its *expressions* under control of the *format_string*. The syntax for the **printf** command is:

printf *"format_string"* [*expression* ...]

For example, this command will print each register name and its contents on a separate line:

`[0:#system#:1]>` **printf "Reg 12: %x\nReg 30: %x\nReg 31: %x\n" r12 r30 r31**

The following display shows an how your screen might look after entering the command above.

```
[0:#system#:1]> printf "Reg 12: %x\nReg 30: %x\nReg 31: %x\n" r12 r30 r31
Reg 12: 0
Reg 30: 0
Reg 31: 0
[0:#system#:1]>
```

See the **printf(3S)** man page for a complete description of this command's capabilities.

End of Chapter

Appendix A Quick Reference

This appendix summarizes the DG/UX™ kernel debugger commands. For the full explanation of a command, refer to Chapter 3, "Kernel Debugger Commands."

brk [**–t**] [**–l**] [*addr_expression* | **–m** *bp_num*] [**–e** [*expression*]] [**–c** *count*]

cmmu

control [*control_register_name* [*register_value*]]

delete {*bp_num* ... | **–a**}

down

eval *expression*

focus [*vp_number*]

global [{*name expr* | **–d** *name* ...}]

halt

help [*command_name*]

ltop *logical_address*

memread [**–l***n*] [**–n**] [**–p**] [**–u**] [*memory_address*] [*count*] [*format*]

memsearch [**–l***n*] [**–n**] [**–p**] [**–u**] [*memory_address*] [*count*] [*format*]

memwrite [**–n**] [**–p**] [**–u**] [**–v**] [*memory_address*] [*count*] [*format*]

mode [**oct** | **hex** | **dec**] [**er** {**on**|**off**}] [**atu** {**on**|**off**}] [**verbose** *n*] [**io** {**duart** | **prom**}]

name [*regular_expression*]

patdump [**–l***n*] [**–n**] [**–p**] [**–u**] *regular_expression* [*format*]

print [*expression* ...]

printf *format_string* [*expression* ...]

proceed

register [*general_register_name* [*register_value*]]

regsearch [**–l***n*] [**–n**] [**–p**] [**–u**] [*memory_address*] [*count*] [*format*]

set *global_name expression*

status

trace [{*vp_num* ... | *vp_num1*–*vp_num2* | *vp_num*– | –*vp_num* | –}]

translate [*expression*]

up

view [**–l***n*] [**–n**] [**–p**] [**–u**] [*memory_address*] [*count*] [*format*]

vp [**–a**] [*vp_number* ...]

End of Appendix

Appendix B
Context Blocks

In this appendix, we describe the kernel's context block. A context block is a save area for a process' registers and state. A context block represents the state of a process and VP at the time that a breakpoint is taken or when an interrupt, fault, or exception occurs.

The operating system maintains a stack of context blocks for each process. Context blocks are a useful debugging tool, because you can see the state of the machine after a fault. Or, you can trace backwards through a series of context blocks to see the state of the machine at different times.

Depending on where you take a breakpoint, you may want to look at the second context block in the stack. If a panic was caused by an exception, the second block will contain information about what caused the exception—the first block contains information about the breakpoint (Figure B–1).

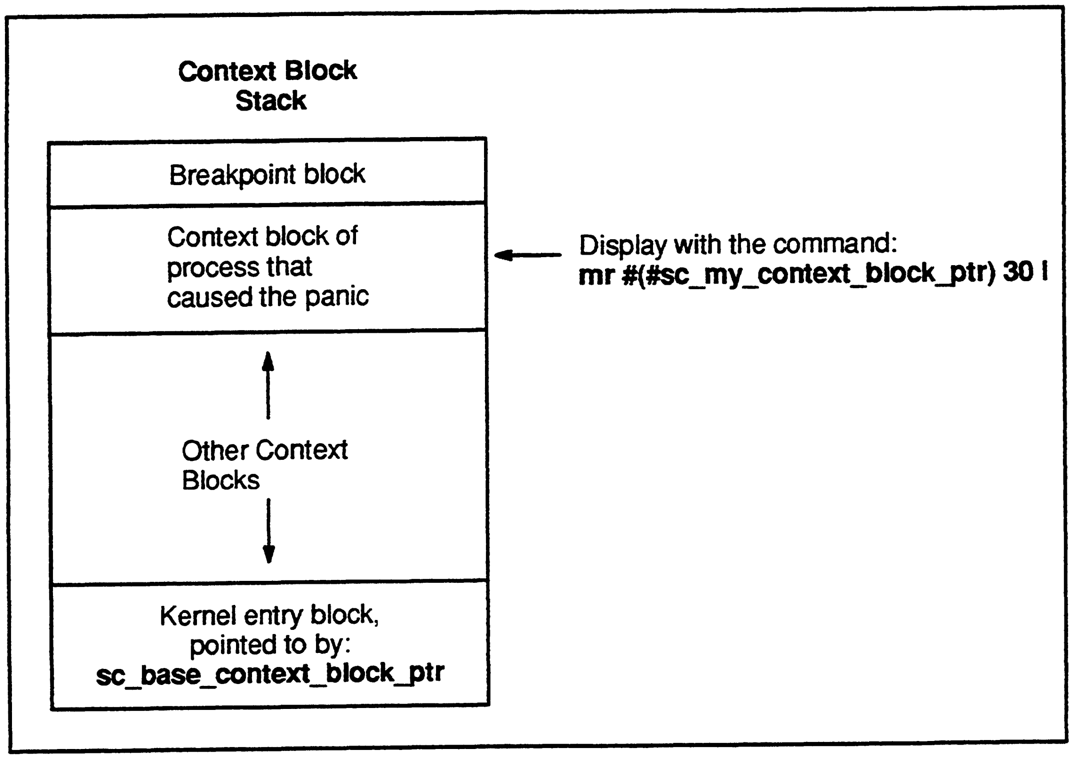

Context Block Stack
Breakpoint block
Context block of process that caused the panic
Display with the command: mr #(#sc_my_context_block_ptr) 30 l
Other Context Blocks
Kernel entry block, pointed to by: sc_base_context_block_ptr

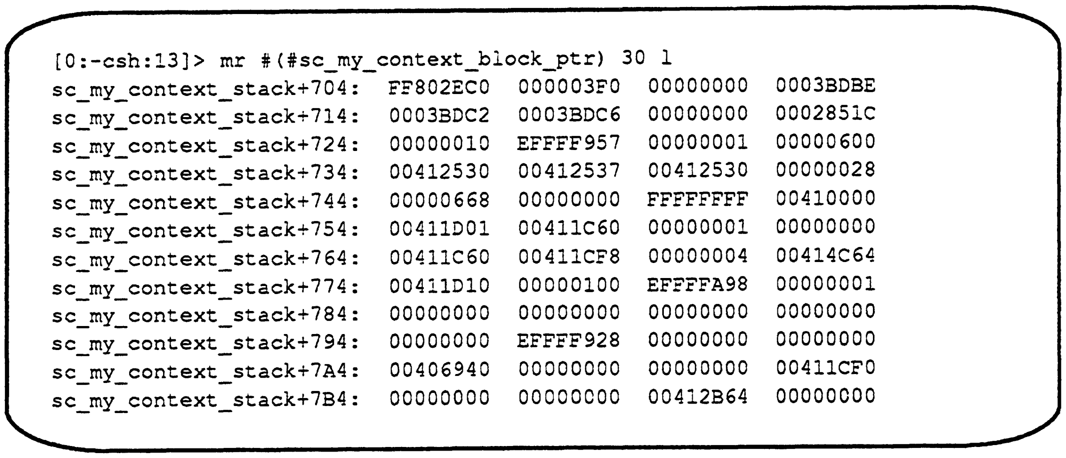
Figure B–1 Context Block Stack

To look at the second context block when a breakpoint at **sc_panic** is taken, use the **memread** command as shown in the following example:

```
[0:#system#:1]> mr #(#sc_my_context_block_ptr) 30 1 ↵
```

This command looks at the address pointed to by the address of the context block header pointer. In other words, the command dereferences through two pointers to get the second context block for that process. As you focus on different processes in the debugger, the header pointer is reset to point to the last context block for the process on which you are focused.

The screen image of an example context block that was taken after a breakpoint is shown below.



```
[0:-csh:13]> mr #(#sc_my_context_block_ptr) 30 1
sc_my_context_stack+704:  FF802EC0  000003F0  00000000  0003BDBE
sc_my_context_stack+714:  0003BDC2  0003BDC6  00000000  0002851C
sc_my_context_stack+724:  00000010  EFFFF957  00000001  00000600
sc_my_context_stack+734:  00412530  00412537  00412530  00000028
sc_my_context_stack+744:  00000668  00000000  FFFFFFFF  00410000
sc_my_context_stack+754:  00411D01  00411C60  00000001  00000000
sc_my_context_stack+764:  00411C60  00411CF8  00000004  00414C64
sc_my_context_stack+774:  00411D10  00000100  EFFFFA98  00000001
sc_my_context_stack+784:  00000000  00000000  00000000  00000000
sc_my_context_stack+794:  00000000  EFFFF928  00000000  00000000
sc_my_context_stack+7A4:  00406940  00000000  00000000  00411CF0
sc_my_context_stack+7B4:  00000000  00000000  00412B64  00000000
```

The parts of the context block are shown in Figure B–2 and described after the figure.

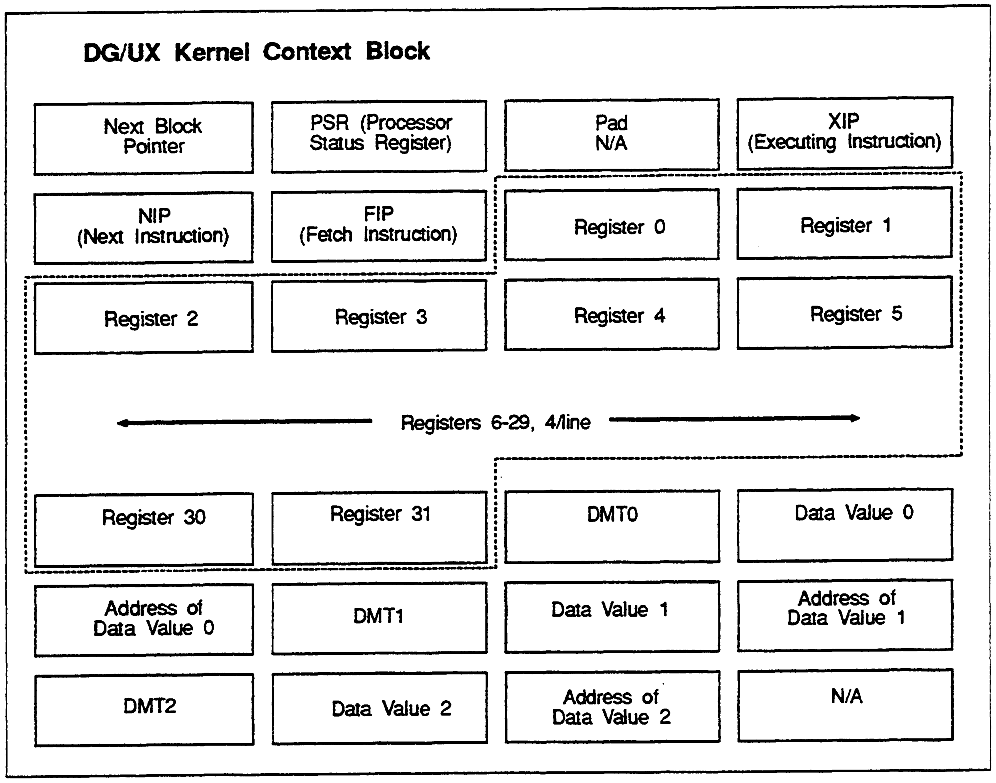


Figure B–2 Parts of a Context Block

Next Block Pointer

The Next Block Pointer points to the next context block in a stack of context blocks. Typically you can get to the next context block through the variables **context_block_ptr** or **context_stack_ptr**. These two variables have identical structures underneath them.

We suggest that you look first at the context block pointer; if the context block pointer doesn't contain relevant information, then look at the context stack pointer. (The context stack reflects the state of the kernel. If a kernel exception occurs, you would want to look at the context stack.)

PSR

The Processor Status Register (PSR) is the state of machine at the time of the fault or exception that this context block was saved. The most significant bit in the PSR is the high bit. If the high bit is set, the process was running in supervisor space when the exception occurred; if the high bit is clear, the process was running in user space.

Pad

The contents of the padding space are not significant.

XIP, NIP, and FIP

XIP, NIP, and FIP are the pipelined instruction registers. XIP is the executing instruction, or the current PC. NIP is the next instruction. FIP is the fetch instruction. These three instruction pointers tell you the state of the pipeline and what was happening with the process. Typically, these three pointers are even, with no exception taking place. Figure B–3 shows the context block formats for XIP, NIP, and FIP.

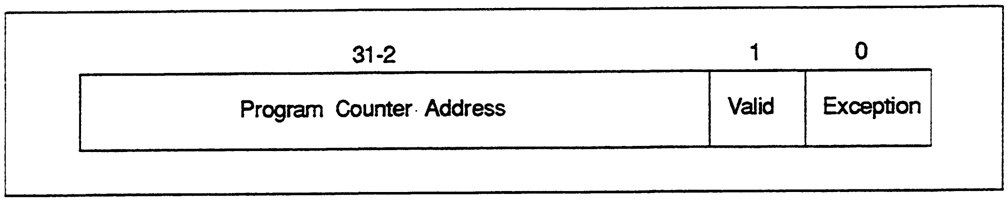


Figure B–3 XIP, NIP, and FIP Format

Often, the current instruction (XIP) is the address that has caused a fault. If the process was running in supervisor space and a fault occurred, you can mask off the two bits in XIP to look at that address (using the **view** command). The debugger will place you at the exact instruction that caused the fault. Typically, if the exception is a serial exception, the address that caused the fault will be the XIP.

Registers

The state of the JP registers are captured in the context block at the time of the fault or exception. For instance, the process was loading instructions that involved three registers and if it tried to load the last register from a null pointer, an exception would occur because the machine tried to reference a non–existent address. You would see that the third register argument had an invalid kernel address.

Note that register zero should always contain the value zero (00000000).

The addresses in the registers correspond to the setting of the PSR high bit. If the process was running in supervisor space, the addresses represent kernel addresses; if the process was running user space, the addresses represent user space addresses.

DMT Registers

Figure B–4 shows the format of the Data Memory Transaction (DMT) registers. The fields in the format are described in the list that follows the figure.

31-16	15	14	13	12	11-7	6	5-2	1	0
Zero	BO	DAS	DOUB1	LOCK	DREG	SD	ENABLE0-3	WRITE	VALID

Figure B–4 Data Memory Transaction Registers

VALID

If the Valid Transaction bit (VALID) is set, the address was valid. If it is clear, the address was invalid.

WRITE

If LOCK (field 12) is set, the Read/Write Transaction bit (WRITE) shows whether the operation was a read (clear) or a write (set). If LOCK is clear, the WRITE bit shows whether the operation is the read (clear) or write (set) access of an **xmem** instruction.

ENABLE0–3

The Byte Enable Bits field (ENABLE0–3) shows the kind of store (**st**, **st.b**, **st.h**, or **st.d**). From these bits, you can decode what the instruction was, as shown below:

0001=byte
0010=byte
0011=half word
0100=byte
1000=byte
1100=half word
1111=word .fi

SD

The Sign–Extended bit (SD) shows part of the **xmem** instruction. If set, the load operation should be sign–extended. If clear, the operation should be zero–extended.

DREG

The Destination Register field (DREG) species the destination register for a memory access.

LOCK

The Bus Lock field bit (LOCK) is clear if the transaction is part of an **xmem** instruction.

DOUB1	The Double Word bit (DOUB1) will be set if the access is the first in a double–word transaction.
DAS	The Data Address Space bit (DAS) is set if the instruction is in supervisor space; it is clear if the instruction is in user space.
BO	The Byte Ordering bit (BO) is set if the instruction is little–endian; it is clear if the instruction is big–endian.

The DMT registers often relate to store instruction faults. If a store instruction causes a fault, you can look backwards from the XIP address to see the last store that occurred (in a DMT). The XIP, NIP, and FIP don't necessarily correlate to DMT0, DMT1, and DMT2. The XIP is the current PC and the DMT0 is where the store exception occurred.

For example, a process could be executing a series of instructions that include a store instruction, instructions that don't access memory, another store instruction, more instructions that don't access memory, and another store instruction. If one of the store instructions tries to store to an address that doesn't exist, the exception might not actually occur until the cache is filled and the system looks up the page table entry. In this case, the XIP, NIP, and FIP would point to addresses far away from the DMT addresses.

Data Values and the Addresses of Data Values

Data Value 0, Data Value 1, and Data Value 2 are the values you are trying to store at their respective addresses. Typically, only the first data value will be valid.

For information about the contents of context blocks, you can also refer to the *MC88100 RISC Microprocessor User's Manual.*

End of Appendix

Index

TIPS ORDERING PROCEDURES

TO ORDER

1. An order can be placed with the TIPS group in two ways:
   a) MAIL ORDER – Use the order form on the opposite page and fill in all requested information. Be sure to include shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the order form.

   Send your order form with payment to:

   Data General Corporation
   ATTN: Educational Services/TIPS G155
   4400 Computer Drive
   Westboro, MA 01581–9973

   b) TELEPHONE – Call TIPS at (508) 870–1600 for all orders that will be charged by credit card or paid for by purchase orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:
   a) Purchase Order – Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany order.
   b) Check or Money Order – Make payable to Data General Corporation.
   c) Credit Card – A minimum order of $20 is required for Mastercard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to the following chart:

Total Quantity	Shipping & Handling Charge
1–4 Units	$5.00
5–10 Units	$8.00
11–40 Units	$10.00
41–200 Units	$30.00
Over 200 Units	$100.00

If overnight or second day shipment is desired, this information should be indicated on the order form. A separate charge will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$1–$149.99	0%
$150–$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all times.

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870–1600 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate DG Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	SHIP TO: (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME__________	COMPANY NAME__________
ATTN:__________	ATTN:__________
ADDRESS__________	ADDRESS (NO PO BOXES)__________
CITY__________	CITY__________
STATE ______ ZIP______	STATE______ ZIP______

Priority Code __________ (See label on back of catalog)

Authorized Signature of Buyer (Agrees to terms & conditions on reverse side)	Title	Date	Phone (Area Code)	Ext.

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TOTAL PRICE

ORDER TOTAL	
Less Discount See B	–
SUB TOTAL	
Your local* sales tax	+
Shipping and handling – See A	+
TOTAL – See C	

Tax Exempt # or Sales Tax (if applicable) __________

A

☐ UPS	ADD
1-4 Items	$ 5.00
5-10 Items	$ 8.00
11-40 Items	$ 10.00
41-200 Items	$ 30.00
200+ Items	$100.00

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.

☐ UPS Blue Label (2 day shipping)

☐ Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0 – $149.99	0%
$150 – $499.99	10%
Over $500.00	20%

C PAYMENT METHOD

☐ Purchase Order Attached ($50 minimum)
P.O. number is__________. (Include hardcopy P.O.)

☐ Check or Money Order Enclosed

☐ Visa ☐ MasterCard ($20 minimum on credit cards)

Account Number ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐ Expiration Date ☐☐☐☐

____
Authorized Signature
(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on all purchases shipped to states where DG maintains a place of business, which covers all 50 states. Please include your local taxes when determining the total value of your order. If you are uncertain about the correct tax amount, please call 508-870-1600.

Form 702
Rev. 8/87

DATA GENERAL CORPORATION TECHNICAL INFORMATION AND PUBLICATIONS SERVICE

TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION
Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub–licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES
Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS
Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY
DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for a period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provided it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY
EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY
A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL
A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict of law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)
Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific to a particular revision of the product. Consequently user programs or systems based on this information and material may be revision–locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes no representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any such use and I and my company (Customer) hold Data General completely harmless therefrom.

Using the DG/UX™
Kernel Debugger

093–701075–01

Cut here and insert in binder spine pocket