digital

VAX-11
Symbolic Debugger
Reference Manual

Order No. AA-D026A-TE

VAX11

August 1978

This document describes the VAX-11 Symbolic Debugger, a program used in locating errors in executable user images. The information in this document is particularly pertinent to programmers using the VAX-11 MACRO assembly language.

VAX-11
Symbolic Debugger
Reference Manual

Order No. AA-D026A-TE

9

SUPERSESSION/UPDATE INFORMATION:	This is a new document for this release.
OPERATING SYSTEM AND VERSION:	VAX/VMS V01
SOFTWARE VERSION:	VAX/VMS V01

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, August 1978

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	

CONTENTS

CONTENTS (Cont.)

CONTENTS (Cont.)

FIGURES

TABLES

PREFACE

MANUAL OBJECTIVES

This manual describes the facilities supplied with the VAX-11 Symbolic Debugger. It is primarily an aid in debugging programs written in VAX-11 MACRO assembly language. For information on debugging programs written in other languages, such as VAX-11 FORTRAN IV-PLUS, refer to the appropriate language user's guide.

INTENDED AUDIENCE

This manual is intended for programmers using VAX-11 MACRO. To get the most out of this manual, you should have a working knowledge of VAX-11 architecture and be familiar with the VAX/VMS operating system. However, while not a tutorial, the manual can be used by relatively inexperienced programmers.

STRUCTURE OF THIS DOCUMENT

This manual comprises 15 characters and 1 appendix. Chapter 1 provides a functional overview of the VAX-11 Symbolic Debugger's concepts and facilities, while each subsequent chapter discusses each concept and facility individually. Finally, there is a summary of debugging commands in Appendix A.

ASSOCIATED DOCUMENTS

To obtain supplemental information you may need, the following documents are recommended:

- VAX-11/780 Architecture Handbook
- VAX/VMS Primer
- VAX-11 Linker Reference Manual

For a complete list of VAX-11 documents, see the VAX-11 Information Directory.

CONVENTIONS USED IN THIS DOCUMENT

The following syntactic conventions are used in this manual:

- Uppercase words and letters used in command examples indicate that you should type the word or letter as shown
- Lowercase words and letters used in format examples indicate that you are to substitute a word or value of your choice
- Brackets ([]) indicate optional elements
- Braces ({}) are used to enclose lists from which one element is to be chosen
- Ellipses (...) indicate that the preceding item(s) can be repeated one or more times

CHAPTER 1

INTRODUCTION TO DEBUGGING

One of the most difficult stages in the program development process involves locating and correcting errors, commonly called "debugging." This stage is reached after you've written the source program and compiled or assembled it successfully, but have received erroneous output when you tried to run the executable program. This means you've followed all the rules of the source language and have not violated any constraints of the compiler or assembler, but you've probably made at least one programming error that is producing incorrect results.

To help you find such errors, VAX-11 provides a special program: the Symbolic Debugger (or, simply, the debugger). The debugger lets you control the execution of your program so you can monitor specific locations; change the contents of locations; check the sequence of program control; and otherwise locate and correct errors as they occur. After you've tracked down the mistakes, you can edit your source program, recompile or re-assemble, relink, and execute the corrected version.

1.1 VAX-11 SYMBOLIC DEBUGGER FACILITIES

The VAX-11 debugger includes many facilities to help you.

- It is interactive - You control your program and converse with the debugger from your terminal.
- It is symbolic - You can refer to locations by using the symbols you created in your source program. The debugger also displays locations as symbolic expressions.
- It supports different languages - The debugger lets you converse in the language of your source program (for example, FORTRAN). You can change from one language to another in the course of a debugging session by means of a command (see Table 1-1 for a summary of commands).
- It permits a variety of data forms - You can control the mode in which the debugger accepts and displays addresses and data. An address can be represented symbolically or as a virtual address, in decimal, octal, or hexadecimal. Data can be represented by symbols, symbolic expressions (X+3), VAX-11 MACRO instructions, ASCII character strings, or numeric strings in decimal, octal, or hexadecimal.

1.2 USING THE VAX-11 DEBUGGER

This section comprises brief descriptions of the functions of the debugger, and how to use them. The remaining chapters of the manual provide more detailed information on how these functions can be utilized.

1.2.1 Breakpoints

A breakpoint is a place in your program where execution is suspended so the debugger can get control and request a command. Program execution is suspended before the instruction at this breakpoint address is executed. Thus, by setting breakpoints, you are able to examine the status of your program at key moments of its execution. See Chapter 7.

1.2.2 Tracepoints and Opcode Tracing

Tracepoints help you follow the sequence of program execution. When you set a tracepoint in your program, the debugger will momentarily suspend execution at that point, display a message indicating that the tracepoint was reached, and continue execution from that point. Thus, you can determine whether the program is being executed in the proper sequence.

You can also trace the execution of branch and/or call-type instructions, by specifying which set of instruction opcodes you want traced. See Chapter 8.

1.2.3 Watchpoints

A watchpoint refers to a specific location, and causes the program to stop whenever the location is modified. Thus, you can monitor addresses to ensure that they are not being modified inadvertently, or in an unspecified manner. See Chapter 9.

1.2.4 Examining and Modifying Locations

When execution of your program is suspended, you can look at the contents of locations and modify them as you wish. For example, you might examine a location to verify that it contains the expected value. You might then change the value to determine the effect on subsequent execution. See Chapter 10.

1.2.5 Evaluating Expressions

You can use the debugger as a calculator, to compute the value of expressions, perform radix conversions, compute an address value, etc. See Chapter 11.

1.2.6 Program Control

You can initiate and suspend program execution in a number of ways. For example, you can set breakpoints (see Section 1.2.1), and specify the GO command to start or continue execution. You can also execute the program on a one by one basis by means of the STEP command. This is slower than the GO-breakpoint method, but allows a closer examination of the program, particularly in those areas that are especially complex and prone to obscure errors.

1.2.7 Starting and Ending Debugging Sessions

There are several methods of passing control to the VAX-11 debugger. Generally, you specify a qualifier when you compile or assemble the source program, to ensure that the symbols defined in the program are included in the debugger's symbol table (see Section 6.3). Then, when you link the object program, you include a qualifier to make the debugger available to the program. For example:

```
$ LINK/DEBUG file-spec
```

When you enter the RUN command to begin executing your program, the debugger gets control, displays its identifying message, and prompts for a command. The prompt has the form:

```
DBG>
```

You respond to the prompt with one of the commands recognized by the debugger (see Section 1.2.8). To terminate the debugging session, use the EXIT command.

1.2.8 Debugger Commands

You use a set of commands to tell the debugger what to do. The general form of a debugger command is:

```
cmd[/qualifier] [keyword [param ...]][DO command [;command...]]
```

cmd
: command name (SET, CANCEL, SHOW, etc.) indicating the general function to be performed.

/qualifier
: modifies the effect of the command.

keyword
: indicates the specific function to be performed by the command (MODULE, SCOPE, LANGUAGE, etc.).

param
: qualifies the function in some way, such as specifying a range of locations to be monitored.

DO command(s)
: list of debugger commands to be performed. (Used only with SET BREAK commands.) If more than one command is specified, you must put a semicolon between them.

Table 1-1 summarizes the debugger commands.

Table 1-1
Summary of VAX-11 Symbolic Debugger Commands[1]

Command	Keyword	Parameter	Function
CALL	routine name	argument list	Call a subroutine.
SET SHOW CANCEL	BREAK EXCEPTION BREAK LANGUAGE MODE MODULE SCOPE STEP TRACE WATCHPOINT ALL	address expression name list /ALL step type(s) opcode class(es)	Initialize (SET), display (SHOW), or delete (CANCEL) the specified elements. Not all combinations can be used. For example, SET ALL is not a valid command. See individual command descriptions.
DEFINE		symbol=value	Equate a symbol and a value.
DEPOSIT[/mode]		address=data	Put data in a location.
EVALUATE[/mode]		expression	Compute the value of expressions.
EXAMINE[/mode]		address[:address]	Display contents of an address, or range of addresses.
EXIT			Terminate debugger.
GO		[address]	Start or continue program execution.
STEP	SYSTEM/NOSYSTEM INTO/OVER LINE/INSTRUCTION	[decimal integer]	Execute a portion of the program, then stop.
Appendix A covers the debugger commands in greater detail.			

1 Underlines indicate the abbreviated form of a command or keyword.

1.3 SYMBOLIC REFERENCES

The debugger lets you refer to locations symbolically. Thus, if you've defined a symbol in your source program as MINIM, you can tell the debugger to examine or modify the contents of MINIM, without worrying about MINIM's location in the executable image. The debugger resolves symbolic references by using a symbol table and scope.

1.3.1 Debugger Symbol Table

The debugger maintains a table that describes the symbols that may be referenced during a debugging session. The debugger can resolve symbolic references only to symbols described in this table. When you initiate a debugging session (assuming you've met the conditions

needed to supply symbol information), this table describes permanent symbols (for example, general register definitions); global symbols; and local symbols in the first module input to the linker. Use the SHOW MODULE command to determine which modules' symbols are currently in the symbol table. You can add or delete symbols by means of the SET MODULE and CANCEL MODULE commands; or by using the DEFINE command. See Chapter 6 for more information on the symbol table and the commands used to control its contents.

1.3.2 Scope

If a symbol is unique, there can be no ambiguity when you refer to it. However, if there are two or more symbols with the same name, appearing in different modules, and these symbols are in the debugger's symbol table, you must indicate which of them you mean when you use the symbol's name in a debugger command. To do so, you specify the scope of the symbol, either by prefixing a module name, or by means of a SET SCOPE command (if the current scope or the default scope is not appropriate). The default scope is initialized to the first module input to the linker. See Chapter 6 for a discussion of scope and related commands.

1.3.3 Pathnames

A pathname comprises the complete, unambiguous identity of a location in your executable program. For unique symbols, the symbol name alone is a pathname. For symbols that are not unique, a pathname comprises the scope and the symbol name, in the form:

```
scope\symbol
```

1.3.4 Local Symbol Definition

To ensure that symbols local to your source program appear in the debugger's symbol table, you must indicate to the assembler or compiler that you want local symbol information to be available to the debugger. You do this by specifying the appropriate qualifier in the command line when you assemble or compile the program. For VAX-11 MACRO, the qualifier is /ENABLE=DBG. The qualifiers used for other languages are described in the user's guides for those languages.

CHAPTER 2

BEGINNING AND ENDING A DEBUGGING SESSION

This chapter tells you how to initiate and terminate the debugger.

2.1 INITIATING THE DEBUGGER

The usual method of initiating the debugger is by specifying:

```
$ RUN[/DEBUG] file-spec
```

You need to express the qualifier, /DEBUG, only if you did not specify /DEBUG at link time; "file-spec" is the file identification assigned to your program at link time. You cannot initiate the debugger by this method if you specified NOTRACE at link time (LINK/NOTRACE ...).

You can inhibit or defer the debugger by specifying:

```
$ RUN/NODEBUG   file-spec
```

You need not express the qualifier, /NODEBUG, unless you specified the /DEBUG qualifier when you linked your program. If you specify $ RUN/NODEBUG, but later decide you want the debugger, interrupt your program by typing CTRL/Y (echoed as ^Y) and respond to the command interpreter's prompt with the command DEBUG. For example:

```
^Y

$ DEBUG
```

The debugger indicates its readiness to accept commands by displaying its prompt, DBG>. To determine the location at which you interrupted your program, enter the command

```
DBG>EXAMINE/INSTRUCTION @PC
```

The debugger reports the current contents of the program counter (PC), plus the instruction to be executed if your program continues at the indicated location, as shown below.

```
location:  instruction
```

Note that typing CTRL/C has the same result, and echoes the same as CTRL/Y if your program does not include an exception handler for this condition.

Table 2-1 summarizes the command qualifiers that affect debugger initiation.

Table 2-1
Debugger Initiation Qualifiers

Command Sequence	Effect
LINK/NOTRACE RUN	Inhibits both debugging and traceback
LINK RUN	Inhibits debugging, allows traceback
LINK RUN/DEBUG	Allows debugging, but full symbolic debugging will not be possible
LINK/DEBUG RUN	Allows full symbolic debugging
LINK/DEBUG RUN/NODEBUG	Inhibits debugging but allows traceback

If you receive the RMS "file not found" message in response to a RUN command, instead of the debugger's identification message, it may mean that you mistyped the program's name, or that the corresponding logical name for the Symbolic Debugger is not assigned to the system directory that contains the debugger. In the latter case, you should log out and then log in again.

2.2 STARTUP CONDITIONS

The following sections describe how to modify the conditions that exist when the debugger is initialized.

2.2.1 Startup Messages

When the debugger first gets control, it displays messages in the following form:

```
VAX/VMS DEBUG   version number   release date

%DEBUG-I-INITIAL, language is xxx, scope and module set to yyy

DBG>
```

The first message identifies the installed version of the debugger and the release date. The second message indicates that the debugger automatically has:

- Set its language to the source language of the first module in your program.
- Set the name of the first module as the scope (area prefix for symbolic pathnames; see Chapter 6).
- Read symbol information from the first module into its symbol table for use in creating symbolic pathnames.

If this message does not appear, the debugger has not performed these initialization procedures. You must use the SET LANGUAGE, SET SCOPE, and SET MODULE commands to initialize the appropriate settings.

The debugger's prompt, DBG>, indicates that it is now ready to process your commands.

2.2.2 Language Setting

The debugger can interpret input or display output in the syntax of supported native-mode languages. The language is initially set to the source language of the first module linked in your program. Thereafter, you can change to any supported language by the command

```
DBG>SET LANGUAGE language-name
```

where "language-name" is MACRO (for VAX-11 MACRO) or FORTRAN (for VAX-11 FORTRAN IV-PLUS).

2.2.3 Scope Setting

When the debugger is invoked, the scope is set to the first module in your program. To refer to symbols in this module, you need only specify the symbol names. To refer to duplicate symbol names in other modules, specifying only the symbol name, you must either give the whole pathname, or change the scope. You must also ensure that the symbol information is in the symbol table (see Section 2.2.4). To change the scope, use the SET SCOPE command. For example:

```
DBG>SET SCOPE AJAX
```

Whatever previous scope existed is superseded by AJAX. See Section 6.4 for more information on SCOPE commands.

2.2.4 Setting Symbols

The debugger initially reads into its symbol table the symbol information associated with the first module in your program. If you intend to make use of other modules' symbols in pathnames, you must use the SET MODULE command to read in the symbol information from specified modules. The commands, SET MODULE, SHOW MODULE, and CANCEL MODULE let you read information into the table, display its status, and purge its contents, respectively. See Section 6.3 for more information.

2.2.5 Entry and Display Modes

The debugger's entry/display modes determine how it interprets your command entries and displays output. The initial condition of these modes is: SYMBOLIC, NOINSTRUCTION, NOASCII, NOGLOBAL, HEXADECIMAL, LONG, and SCOPE.

Chapter 5 describes these modes, and use of the commands, SET MODE, SHOW MODE, and CANCEL MODE.

2.3 TERMINATING A DEBUGGING SESSION

You indicate that you are through by responding to the DBG> prompt with the command:

```
EXIT
```

You can also terminate the debugger by typing CTRL/Z.

The VAX/VMS command interpreter gains control, and displays its prompt character ($). After exiting from the debugger, you can not use the DEBUG command to reinvoke the debugger.

CHAPTER 3

CONTROLLING PROGRAM EXECUTION

This chapter describes how you start your program with GO and continue your program with STEP or GO. The chapter also describes how to interrupt your program, for example, to return control to the debugger when your program is looping.

3.1 INITIATING AND CONTINUING EXECUTION WITH GO

The GO command tells the debugger to let your program run, beginning either at the transfer address, at a starting address you specify, or from a location at which the debugger stopped it. Program execution continues until an exception condition (such as a breakpoint) causes the debugger to stop execution, or the program runs to completion (refer to Chapter 12 for information about exception conditions).

The command format is:

```
DBG>GO [address-expression]
```

The first GO command without an address starts the program at its transfer address. Note that the debugger responds with the message

```
[routine] start PC is mod\rtn.
```

If "routine" is included in the message, "mod\rtn" is 2 less than the actual PC value. The PC was at the beginning of routine "rtn" in module "mod".

If you enter a GO command subsequent to program suspension (such as following a breakpoint) and do not specify an address, execution resumes from the point at which it was suspended (for example, at the instruction at the breakpoint's address).

If you specify an address with GO, that address replaces the current contents of the program counter (PC) and execution starts at or continues from the new location. Your program's behavior can be unpredictable if you initiate execution at any address other than its transfer address, or if you attempt to restart your program at its transfer address or any other address.

3.2 STEPPING THROUGH YOUR PROGRAM

The STEP command lets you specify the number of instructions (VAX-11 MACRO) or statements (FORTRAN) that your program can execute before the debugger regains control. The basic command format is:

```
DBG>STEP [decimal-integer]
```

If you do not include a decimal integer (2 through 32767), or you specify a value of 1, the debugger executes the next instruction (or statement) and stops the program. (A step value of zero will be accepted, but no step will be performed.) Although you can specify large step counts, the recommended practice is to set a breakpoint (see Chapter 7) at the desired location and use GO to run to the specified location.

If an exception condition stops your program before the specified number of instructions or statements are executed, the debugger resets the step counter to zero, as though the specified number of steps had been completed.

STEP also has modes that determine how the debugger interprets the step increment. The following sections describe the functions of these modes and how you can express them at command level or set them as default conditions for stepping.

3.2.1 Step Types

The STEP types are:

```
LINE or INSTRUCTION
INTO or OVER
SYSTEM or NOSYSTEM
```

You can express these types at command level as follows:

```
DBG>STEP [/type[...]] [decimal-integer]
```

where a slash (/) must precede each step type. A step type expressed at command level overrides its counterpart at the default level (see SET STEP, below).

The STEP types exert the following control over program stepping.

INSTRUCTION	Step in increments of instructions (the only valid increment for VAX-11 MACRO).
LINE	Step in increments of lines for line-oriented (statement) languages, such as FORTRAN (ignored for VAX-11 MACRO).
INTO	Step into a routine called by a call-type instruction (CALLS, CALLG, JSB, BSBB, BSBW).
OVER	Step over the next routine called by a call-type instruction; that is, the instruction, all routine instructions (or lines), and the corresponding RET instruction are treated as one step.

NOSYSTEM	Decrement the step count only for steps executed in nonsystem space; the debugger ignores instruction/line steps executed in system space.
SYSTEM	Decrement the step count for instructions (or lines) that are executed in system space as well as process space. (For a definition of system space, see the VAX-11 Software Handbook.)

For VAX-11 MACRO, the initial STEP modes are:

INSTRUCTION, OVER, and NOSYSTEM.

3.2.2 Setting Step Types

You can change the default types for STEP at any time with the SET STEP command.

```
DBG>SET STEP type[,type...]
```

Multiple type entries must be separated by commas.

3.2.3 Showing Step Types

The SHOW STEP command reports the current STEP types. For example:

```
DBG>SHOW STEP
step type:  nosystem, by line, over routine calls
```

3.3 INTERRUPTING EXECUTION

You can interrupt execution of your program or the debugger by typing CTRL/Y (echoed at the terminal as ^Y). VAX/VMS stops your program and displays the command interpreter prompt. To return control to the debugger, you must type the command DEBUG.

```
$ DEBUG
```

The debugger in turn displays its prompt, DBG>. You can also continue execution of your program (or the debugger) from the location at which you interrupted it by responding with the command CONTINUE rather than DEBUG.

```
$ CONTINUE
```

Typing any VAX/VMS command other than DEBUG or CONTINUE will generally cause your program to exit immediately.

CHAPTER 4

SPECIAL CHARACTERS

This chapter describes how the debugger interprets special characters in arithmetic expressions, in address expressions, and as delimiters with VAX-11 MACRO as the current language. Tables 4-1, 4-2, and 4-3 summarize the arithmetic, address, and delimiting functions, respectively. Some characters (such as @) appear in more than one table because of multiple uses, based on context.

4.1 EVALUATING ARITHMETIC EXPRESSIONS

The debugger performs integer arithmetic. All operations are performed according to the length mode currently in effect (that is, BYTE, WORD, or LONG) with arguments and results limited to the corresponding value ranges. The debugger truncates values that exceed the current length mode by discarding the most significant bit positions. Note, however, that truncation does not occur on data that is "typed," for example, FORTRAN double precision values.

Table 4-1 lists special characters used in arithmetic expressions.

Table 4-1
Arithmetic Special Characters

Character	Interpretation
+	Arithmetic addition (binary) operator, or unary plus sign.
-	Arithmetic subtraction (binary) operator, or unary minus sign.
*	Arithmetic multiplication operator.
/	Arithmetic division operator.
@	Arithmetic shift operator.
<...>	Precedence operators; do <enclosed> first.
^D	Decimal radix operator.
^O	Octal radix operator.
^X	Hexadecimal radix operator.

An arithmetic expression is evaluated in the context of the current language. For VAX-11 MACRO, the debugger evaluates an expression from left to right under the following rules of precedence:

1. Terms or expressions enclosed by angle brackets, <...>, are evaluated first. You can nest expressions to many levels. For example:

   ```
   <BEGIN+<INDEX*100>>
   ```

   The debugger evaluates nested expressions in the order of innermost to outermost.

2. Unary operators and radix operators have priority over arithmetic (binary) operators; thus values are evaluated according to their signs and radices, and indirect "contents of" operations (see Section 4.2.4) are performed before the remaining arguments and terms are evaluated. For example, in the expression

   ```
   A+-@B
   ```

   the value addressed by the contents of B is first negated and then added to the value represented by A. Thus, A+-@B is equivalent to A+<-<@B>>.

3. The arithmetic operations (add, subtract, multiply, divide, and shift) have equal precedence.

Thus, the following expression

```
^D1000 + ^D1000 / 2 * ^D10
```

results in the decimal value 10000

However,

```
^D1000 + << ^D1000 / 2 > * ^D10 >
```

results in the decimal value 6000

4.1.1 Plus Sign (+)

A plus sign, as a binary operator, adds the following argument to the preceding argument (or interim result). As a unary operator, a plus sign means take the following argument as having an unchanged value. The debugger interprets an unsigned argument as having a positive value by default.

Examples:

```
DBG>SET BREAK BEGIN + ^X10
DBG>EVALUATE ^D2000 = ^X1000 + ^O777
```

4.1.2 Minus Sign (-)

A minus sign, as a binary operator, subtracts the following argument from the preceding argument. As a unary operator, a minus sign means negate the following argument.

Examples:

```
DBG>CANCEL WATCH NAME - OFFSET
DBG>EXAMINE INQUEUE - 1000 - INDEX
```

4.1.3 Multiplication Operator (*)

An asterisk multiplies the preceding argument by the following argument.

Examples:

```
DBG>EVALUATE ^X50 * ^D512
DBG>DEFINE PAGE = PAGE - 256 * 4
```

4.1.4 Division Operator (/)

A slash divides the preceding argument by the following argument. Any remainder is discarded. The debugger rejects an attempt to divide by zero.

Examples:

```
DBG>DEFINE MODULO = < INDEX + POINTER >/ QUEUE_SIZE
DBG>SET WATCH < PAGE / 2 > * GO_TO_ZEBRA
```

4.1.5 Shift Operator (@)

The shift operator is a unary "at" sign. It means shift the preceding argument (or interim result) the number of bit positions specified by the following argument. A positive value means shift left; a negative value means shift right. The shift is arithmetic; that is, no wraparound occurs as in a logical shift. Shifts to the left cause loss of the contents of the sign bit. Shifts to the right cause the contents of the sign bit to fill the vacated bit positions.

Examples:

```
DBG>EVALUATE 0F000FFF0 @ 4
000FFF00
DBG>EVALUATE ^XF000FF00 @ - 4
0FF000FF0
```

4.1.6 Precedence Operators (<...>)

The debugger first evaluates terms or expressions enclosed by angle brackets. An expression can contain up to 20 levels of nesting, with the debugger evaluating them in the order of innermost to outermost. The left and right angle brackets must match.

4.1.7 Radix Operators

The debugger interprets numeric arguments in the current radix mode (see Entry and Display Modes, Chapter 5), unless you precede each argument with an explicit radix operator. A radix operator affects only the entry that it accompanies; it has no control over the radix in which the debugger displays a value.

The radix operators for VAX-11 MACRO are:

^D - Decimal radix.

^X - Hexadecimal radix.

^O - Octal radix.

No spaces or tabs are permitted between the radix operator and its operand.

Examples:

```
DBG>EV ^D10+^D10
00000014 (assumes hexadecimal display mode)
DBG>EV ^O77+^XFF
0000013E
DBG>EV 77+^XFF
00000176
```

4.2 SPECIAL CHARACTERS IN ADDRESS EXPRESSIONS

This section describes the significance of special characters that can be used to represent locations in address expressions. Table 4-2 lists the address representation characters.

Table 4-2
Address Representation Characters

Character	Interpretation
.	Represents the location last addressed by an EXAMINE, DEPOSIT, SET BREAK, SET TRACE, or SET WATCH command. This is called the "current" location.
^	Represents the location previous to the last location addressed (as represented by .); (equal to last location less the current length mode; that is, .-1, -2, or -4).
\	Represents the value last displayed by EXAMINE or EVALUATE in NOINSTRUCTION mode; in INSTRUCTION mode for branch instructions only, this character represents the effective destination address of the branch. (The backslash is also used in forming pathnames. See Section 6.1.)
@	"Contents" operator.
:	Range operator (low address:high address) for the EXAMINE command; bit field operator for EVALUATE command (DBG>EVALUATE value<high bit:low bit>).

4.2.1 Current Location Symbol (.)

A dot represents the location last addressed by an EXAMINE, DEPOSIT, SET BREAK, SET TRACE, or SET WATCH command. This value remains unchanged until you use one of these commands to refer to a different location.

Example:

```
DBG>EXAMINE /ASCII  MSG_1
ERR_MESSAGE\MSG_1:    NEZT
DBG>DEPOSIT/ASCII/BYTE  . + 2 = 'X'
DBG>EXAMINE/ASCII   ERR_MESSAGE\MSG_1
ERR_MESSAGE\MSG_1:    NEXT
```

The EXAMINE command assigns a dot to the value of the examined address. You can then use this symbol in the DEPOSIT command's address expression to represent that location.

4.2.2 Previous Location Symbol (^)

A circumflex represents the last location addressed (by EXAMINE, DEPOSIT, SET BREAK, SET TRACE, or SET WATCH) less the current length mode; that is .-1, -2, or -4. The use of this character in INSTRUCTION mode is not recommended, because VAX-11 MACRO instructions vary in length.

Examples:

```
DBG>EXAMINE/ASCII  TEXT1:TEXT1+4
TEXT1:  AH T
TEXT1+4:  WEET
DBG>DEPOSIT /ASCII/BYTE ^ = "S"
DBG>EXAMINE/ASCII  TEXT1:TEXT1+4
TEXT1:  AH S
TEXT1+4:  WEET
```

4.2.3 Last Value Displayed Symbol (\)

A backslash can be used to represent the value last displayed in NOINSTRUCTION mode. In INSTRUCTION mode, a backslash represents the effective operand of the last branch instruction displayed. The value in either mode remains unchanged until the debugger displays a new value or a new branch instruction.

Example:

```
DBG>EV/ADDR PI
1028
DBG>EXAMINE\
CIRCLE\PI:  3.141593
```

The EVALUATE command produces an address value for the location symbolized by PI. The EXAMINE\ command produces the contents of that location.

4.2.4 Contents Operator (@)

The unary "contents" operator (@) requests that the debugger evaluate the expression following it and then extract the contents of the location addressed by the expression value rather than use the expression value itself.

Examples:

```
DBG>EXAMINE PC
PC:  00000448
DBG>EXAMINE/INSTRUCTION @PC
00000448:     MOVB    #OFF,W^0400(R7)
```

The first EXAMINE reports the PC's current contents; the second EXAMINE reports the current contents (in INSTRUCTION mode) of the location (00000448) addressed by the PC's contents.

The command

```
DBG>DEPOSIT MASK = @MASK @ 4
```

shifts the current contents of the location MASK four bit positions to the left. (Note that this example shows how the @ character is used as both a shift and a "contents of" operator.)

The command

```
DBG>EXAMINE @R7 :  @R7 +20
```

displays the current contents of the 21 bytes beginning with the location addressed by the current contents of general register R7.

4.2.5 Range Operator (:)

A colon is used in specifying an address range for an EXAMINE command. The colon is also used as a range operator in bit field specifications for an EVALUATE command (see Chapter 11).

Examples:

```
DBG>EXAMINE INBUFFER:INBUFFER + 6
DBG>EXAMINE .:. + ^X200
DBG>EXAMINE/INSTRUCTION  @PC : @PC+10
```

4.3 SPECIAL DELIMITING CHARACTERS

This section describes the significance of special characters that can be used to delimit various debugger expressions. Table 4-3 lists the delimiting characters.

Table 4-3
Delimiting Characters

Character	Interpretation
/	Precedes mode keywords after commands that can be used to override current modes.
=	Separates an address expression from data entries in a DEPOSIT command; separates a symbol name from its definition in a DEFINE command.
\	Separates elements of a symbolic pathname.
( )	Enclose DO command specifications in a SET BREAK command, or argument list in a CALL command. Note that the debugger does not use parentheses to control the order of evaluation of arithmetic expressions (see Table 4-1).
;	Separates individual commands in a multiple command line, or in a DO command sequence associated with a SET BREAK command.
, (comma)	Separates multiple arguments for input.
Apostrophes or Quote marks	Enclose ASCII string input or VAX-11 MACRO instruction input.
< >	Enclose bit field specification for EVALUATE command.
-	Hyphen as last printing character on line signifies line continuation. The debugger prompts with an underline as the first character of each continued line, and defers command execution until you enter a line that does not end with a hyphen.

4.3.1 Mode Keyword Delimiter (/)

A slash must precede each mode keyword entered after a command.

Example:

```
DBG>EXAMINE/ASCII/BYTE INBUF:INBUF+8
```

This command specifies that the contents of 9 bytes, beginning at INBUF, are to be displayed as ASCII characters.

4.3.2 DEPOSIT and DEFINE Command Delimiter (=)

In a DEPOSIT command, an equal sign separates the address expression from the data entries. In a DEFINE command, an equal sign separates a symbol name from the definition.

Examples:

```
DBG>DEPOSIT X_RAY=0F15,0FFFF5C
```

This command causes the values 0F15 and 0FFFF5C to be deposited in successive longwords, starting at location X_RAY.

```
DBG>DEFINE OFFSET=^X200
```

This command specifies that the symbol OFFSET is to be defined as the hexadecimal value 200.

4.3.3 Symbolic Pathname Element Separator (\)

A backslash separates individual elements of a symbolic pathname.

Examples:

```
DBG>SET BREAK MAIN_CODE\BEGIN
```

In module MAIN_CODE, set a breakpoint at the location identified by the local symbol BEGIN.

```
break at pc = CODE2\LOOP3+10
```

The debugger reports the occurrence of a breakpoint in module CODE2, at the location 10 bytes after the location identified by local symbol LOOP3.

A pathname identifies the program elements needed to completely and unambiguously identify a location. In VAX-11 MACRO, a pathname can be:

- A symbol (a global symbol, or one that you created with the DEFINE command, or any symbol that is unique in the symbol table).
- A symbol in the module to which scope is currently set.
- A local symbol that is unique among the modules current set (see SET MODULE, Section 6.3.1).
- A local symbol preceded by its module name (module name\symbol). Program section names in VAX-11 MACRO are classified as local symbols.

A pathname can be used in any expression; its value is the address value for the location it represents. This feature is useful when the available symbolic information is not sufficient to identify a required location.

4.3.4 DO Command Sequence Delimiters

A SET BREAK command can include a list of commands, separated by semicolons, that the debugger executes whenever your program stops at the breakpoint or watchpoint. This command list, known as a DO command sequence, must be enclosed by parentheses.

Example:

```
DBG>SET BREAK ALPHA DO(EXAMINE COUNT_1:COUNT_7;GO)
```

After the debugger stops the program at location ALPHA, it displays the current contents of the locations, COUNT_1 through COUNT_7, and then resumes execution of the program.

4.3.5 CALL Command Argument Delimiters ((...))

A CALL command can include a list of arguments, separated by commas. Parentheses must enclose any supplied argument or arguments.

Examples:

```
DBG>CALL COMP(A)

DBG>CALL SORT(BASE,ITEMS)

DBG>CALL CALC
```

4.3.6 Command Separator (;)

A semicolon separates individual commands in a multiple command line, or individual commands in a DO command sequence.

Examples:

```
DBG>SET WATCH RAIN_BOW;SET BREAK LOOP3;GO
DBG>SET BREAK CLOSEUP DO(EXAMINE WHERE;GO)
```

If GO, STEP, or CALL is used in a DO command sequence, it must be the last command specified. If not, the debugger prints a message, and the GO, STEP, or CALL and any subsequent commands in the DO sequence are ignored.

4.3.7 Argument Separator (,)

A comma separates individual arguments in an argument list.

Examples:

```
DBG>SET MODULE X_RAY,CLOSE_UP,BAKE
DBG>DEFINE INDEX = ^X200,OPEN=^D512
```

4.3.8 Input String Delimiters

The debugger requires that input strings in ASCII or INSTRUCTION modes be enclosed by matching apostrophes or quotation marks. If you wish to enter a literal apostrophe or quotation mark in a string, use the other type to delimit the string. Otherwise, use either type. Refer to ASCII mode and INSTRUCTION mode (Chapter 5) for mode use and input restrictions.

Examples:

```
DBG>DEPOSIT/ASCII 2500="IT'S"
DBG>DEPOSIT/INSTRUCTION SHUT='MOVL #30,R0'
```

4.3.9 Bit Field Delimiters

A colon within angle brackets signifies a bit field specification that the EVALUATE command is to report on. The syntax is:

```
DBG>EVALUATE value<high bit:low bit>
```

The bit positions are numbered 0 (lowest bit) through 7 (for a byte), 0 through 15 (for a word), and 0 through 31 (for a longword).

The following procedure is recommended when you want to evaluate a bit field when you know the corresponding longword value:

```
DBG>EV 2468A<9:7>

00000005
```

The following command sequence is recommended when you want to evaluate a bit field for which you know only the address.

```
DBG>EXAMINE address-expression
address:        contents

DBG>EVALUATE \<high bit:low bit>
bit-field value
```

The EXAMINE command establishes the location's contents as the value represented by the backslash, the "last value displayed" symbol. This sequence is useful when you want to extract a bit field from the contents of a location.

Examples:

```
DBG>EXAMINE LOOP3
WATCH\LOOP3:  0FFFF8FD0
DBG>EVALUATE \<6:4>
00000005
```

Use the EXAMINE command to display the contents of the location, then use the backslash ("last value displayed" symbol) with the EVALUATE command, indicating the bit positions to be evaluated.

To display other bit patterns of the same location, you can specify the following:

```
DBG>EXAMINE .
WATCH\LOOP3:  0FFFF8FD0
DBG>EVALUATE \ <8:6>
00000007
```

4.3.10 Line Continuation Operator (-)

A hyphen as the last printing character on a line requests continuation of the command line. The debugger echoes an underline as the prompt instead of DBG> for each continued line. You may continue a command line up to approximately 500 characters, exclusive of space and horizontal tab characters.

Example:

```
DBG>EXAMINE/ASCII/BYTE -
_BUFFER:BUFFER+20
```

5.L

CHAPTER 5

ENTRY AND DISPLAY MODES

The entry and display modes determine how the debugger interprets your entries and displays solicited or unsolicited output. This chapter describes the four classes of modes: context, length, radix, and pathname search. It tells you how to use the SET MODE and SHOW MODE commands to establish and report current modes, and how to use the CANCEL MODE command (or the CANCEL ALL command) to restore the debugger's initial modes. The chapter also describes how you can override current modes at the command level for the EXAMINE, DEPOSIT, and EVALUATE commands. The EXAMINE and DEPOSIT commands are described in Chapter 10; the EVALUATE command is described in Chapter 11.

5.1 KEYWORD SUMMARY FOR ENTRY AND DISPLAY MODES

Table 5-1 summarizes the mode keywords. In the table, the following letters are used to indicate mode class:

C - Context
L - Length
R - Radix
P - Pathname search

Table 5-1
Keyword Summary for Entry and Display Modes

Mode Class	Keyword	Function
C	ASCII	Interpret/display data as ASCII characters.
L	BYTE	Interpret/display data in byte lengths.
R	DECIMAL	Interpret/display data in decimal radix.
P	GLOBAL	Use symbolic entry as first pathname in search.
R	HEXADECIMAL	Interpret/display data in hexadecimal radix.
C	INSTRUCTION	Interpret/display data as VAX-11 MACRO instructions.

(continued on next page)

Table 5-1 (Cont.)
Keyword Summary for Entry and Display Modes

Mode Class	Keyword	Function
L	LONG	Interpret/display data in longword lengths.
C	NOASCII	Inhibit entry/display of ASCII characters.
P	NOGLOBAL	Use symbolic entry as last pathname in search.
C	NOINSTRUCTION	Inhibit entry/display of VAX-11 MACRO instructions.
P	NOSCOPE	Inhibit SCOPE's contribution to pathname.
C	NOSYMBOLIC	Inhibit display of symbolic addresses.
R	OCTAL	Interpret/display data in octal radix.
P	SCOPE	Prefix entry with SCOPE's contents to form pathname.
C	SYMBOLIC	Display symbolic addresses.
L	WORD	Interpret/display data in word lengths.

5.2 INITIALIZED MODES

For VAX-11 MACRO, the modes are initialized as follows: SYMBOLIC, NOINSTRUCTION, NOASCII, NOGLOBAL, HEXADECIMAL, LONG, and SCOPE.

NOTE: In high-level languages, such as FORTRAN, these defaults are overridden by the data typing of variables.

5.3 CONTROL OF DEBUGGING MODES

The SET MODE, SHOW MODE, and CANCEL MODE commands let you change modes, report the current modes, or restore the initial modes, respectively.

5.3.1 Changing Modes

You can change one or more modes with the SET MODE command.

```
DBG>SET MODE mode-keyword[,mode-keyword, ...]
```

The following mode choices are available:

Context modes:

SYMBOLIC or NOSYMBOLIC
INSTRUCTION or NOINSTRUCTION
ASCII or NOASCII

NOTE: If both INSTRUCTION and ASCII modes are active at the same time (or if you enter them both at command level), the debugger defaults to INSTRUCTION mode.

Radix modes:

DECIMAL
HEXADECIMAL
OCTAL

Length modes:

LONG
WORD
BYTE

Pathname search modes:

GLOBAL or NOGLOBAL
SCOPE or NOSCOPE

5.3.2 Reporting Current Modes

You can determine the state of the entry and display modes by using the SHOW MODE command.

```
DBG>SHOW MODE
```

The debugger reports the mode states by keyword (symbolic, ascii, etc.). For example:

```
symbolic, instruction, noascii, scope, noglobal, decimal, long
```

Debugger messages are usually in lower case.

5.3.3 Restoring the Debugger's Initial Modes

To restore the debugger's initial entry and display modes, type

```
DBG>CANCEL MODE
```

Whatever mode changes you have made are canceled and the debugger re-initializes the mode state to:

symbolic, noinstruction, noascii, noglobal, hexadecimal, long, scope

You can also restore the debugger's initial modes by typing

```
DBG>CANCEL ALL
```

This command also cancels all breakpoints, tracepoints, and watchpoints.

5.3.4 Overriding Current Modes at Command Level

The EXAMINE, DEPOSIT, and EVALUATE commands let you temporarily override current modes by specifying mode keywords after the command verb. For example, the command

```
DBG>EXAMINE/BYTE/ASCII BUFFER:BUFFER+^D10
```

causes the debugger to report the current contents of eleven bytes beginning with BUFFER as ASCII characters regardless of the modes currently active.

This mode override feature lets you specify an EXAMINE, EVALUATE, or DEPOSIT command without having to remember (or check) what modes are current. Each mode keyword entered after the command verb must be preceded by a slash.

With the exception of the INSTRUCTION and ASCII modes, mode keywords entered at the command level simply override their counterpart modes. The following summarizes the relationships between command level modes and current modes.

- ASCII/INSTRUCTION modes: these modes are mutually exclusive. The debugger defaults to INSTRUCTION mode if it finds both ASCII and INSTRUCTION active or requested. You can avoid getting unexpected results by leaving both modes in their initialized NO ... states and requesting the particular mode only at command level.

- Radix mode: a radix mode specified at command level controls the debugger's interpretation and display of all numeric information for the command.

- Length mode: a length mode specified at command level overrides the current length mode.

- Symbolic mode: you can set (SYMBOLIC) or inhibit (NOSYMBOLIC) the symbolic mode as you require.

- Pathname search modes: as with symbolic mode, you can set or inhibit the GLOBAL and SCOPE mode conditions at command level as you require.

5.4 CONTEXT MODES

The context modes allow the entry and display of addresses and data in various forms. An address can be represented symbolically or as a virtual address. Data can be represented by symbols, VAX-11 MACRO instructions, or ASCII character strings. The context mode keywords are:

```
SYMBOLIC and NOSYMBOLIC
INSTRUCTION and NOINSTRUCTION
ASCII and NOASCII
```

The above keyword pairs function as on-off switches to allow or inhibit a condition.

The debugger initializes the context modes as: SYMBOLIC, NOINSTRUCTION, and NOASCII.

5.4.1 Effects of Context Modes

The following summarizes the effect of the context modes on the entry and display of addresses and data.

Address Entry:

The debugger is insensitive to the [NO]SYMBOLIC mode for address entries. You can express an address either as a symbolic pathname or as a virtual address.

Address Display:

In SYMBOLIC mode, the debugger displays all locations by pathnames when possible. Offsets are expressed in the current radix mode. When the pertinent symbolic information is unavailable, SYMBOLIC mode is ignored.

In NOSYMBOLIC mode, the debugger displays locations as virtual addresses in the current radix mode.

Data Entry:

In INSTRUCTION mode, the debugger interprets a quoted string entry as a VAX-11 MACRO instruction, interprets numeric values in the current radix mode, and ignores the current length mode. The debugger rejects instructions not enclosed in quotes.

In ASCII mode, the debugger interprets a quoted string entry as ASCII characters. The string is deposited as entered (that is, the current length mode is overridden if necessary).

Data Display:

In INSTRUCTION mode, the debugger displays the current contents of specified locations as VAX-11 MACRO instructions. Most numeric values are displayed in the current radix. The current length mode is ignored and the debugger increments sequential instruction locations on the basis of each instruction's allocated storage. The debugger tries to display instruction operands in symbolic form if the addressing mode is PC-relative or absolute.

In ASCII mode, the debugger displays the current contents of specified locations as ASCII characters. The character count associated with each requested location is limited by the current length mode, to four characters (LONG), two characters (WORD), or one character (BYTE). The current radix mode is ignored.

When both NOASCII and NOINSTRUCTION are in effect the debugger displays the current contents of the specified locations in the current radix mode. The debugger increments sequential locations on the basis of the current length mode, if no data typing information is available (such as in FORTRAN programs).

5.4.2 SYMBOLIC/NOSYMBOLIC Modes

The SYMBOLIC/NOSYMBOLIC modes allow or inhibit the reporting of locations symbolically, that is, by pathname. In VAX-11 MACRO, a pathname can be:

- A symbol (a global symbol, or one that you defined with the DEFINE command - see Section 6.2.2).
- A local symbol preceded by its module name (module name\symbol).

In SYMBOLIC mode, the debugger reports all locations by pathnames. In NOSYMBOLIC mode, the debugger reports all locations as virtual addresses in the current radix mode.

You can enter locations symbolically regardless of which mode is set.

Refer to Chapter 6 for more information on how the debugger builds pathnames, and translates pathnames to values and values to symbolic expressions.

5.4.3 INSTRUCTION/NOINSTRUCTION Modes

The INSTRUCTION/NOINSTRUCTION modes allow or inhibit the entry and display of data as VAX-11 MACRO instructions. In INSTRUCTION mode, the debugger interprets quoted data entries and displays current data only as VAX-11 MACRO instructions. If the debugger cannot interpret your entry as an instruction, it reports that it cannot encode the instruction. If it cannot translate the current contents of a location as an instruction, the debugger reports that it cannot decode the instruction.

NOINSTRUCTION inhibits the entry or display of data as instructions.

The storage requirements of VAX-11 MACRO instructions vary according to the instruction type and number of operands. The debugger ignores the current length mode when it enters or displays instructions; the debugger instead increments the current address according to the number of bytes required or occupied by an instruction.

An instruction string entry must be delimited by apostrophes or quotation marks.

```
DBG>DEPOSIT /INSTRUCTION PLUNK = 'ADDL3 #5,R3,R4'
```

When entering an instruction, you must verify that the length of the data string can be accommodated by the number of bytes you intend to overwrite. The debugger neither guards against spillover into subsequent bytes, nor pads memory left vacant when you replace an instruction with another instruction that requires less storage. While you cannot deposit more than can be accommodated, you can use the NOP instruction to fill bytes that are unoccupied after you complete the deposit of an instruction or instructions.

You should examine the location to be changed and those following it, before and after the deposit to verify that the contents are correct

before you attempt to execute the new instruction. The following example illustrates the change of the instruction in location SORT\BEGIN+12 from an ADDL3 #10,R2,R4 to an ADDL2 #10,R2, which occupies one less byte.

```
DBG>EXAMINE /INSTRUCTION BEGIN+12
SORT\BEGIN+12:  ADDL3   #10,R2,R4
DBG>EXAMINE /INSTRUCTION
SORT\TEST_SEQ: CMPB    (R0)[R2],(R0)[R4]
```

In INSTRUCTION mode, the debugger interprets an EXAMINE command with a null address expression (carriage return typed directly after the verb and mode keywords, if any) to mean display the instruction that follows the location last displayed.

Make the change as follows.

```
DBG>DEPOSIT/INSTRUCTION BEGIN+12='ADDL2      #10,R2'
DBG>EXAMINE/INSTRUCTION
SORT\BEGIN+14:  EMODF @(R1)+,(R0)[R2],(R0)[R4],#0400C7FF,@W^D157(R6)
```

The debugger typically translates a leftover byte and subsequent bytes as parts of some meaningless instruction. If you continue examining locations as instructions, the debugger eventually reports that it cannot decode the instruction, because it determines that the data in the given bytes does not translate into a VAX-11/780 instruction. To suppress the effect of the leftover byte or bytes, you must enter one NOP instruction per byte.

```
DBG>EXAMINE/INSTRUCTION BEGIN+12
SORT\BEGIN+12: ADDL2   #10, R2
DBG>EXAMINE/INSTRUCTION
SORT\BEGIN+14:  EMODF @(R1)+,(R0)[R2],(R0)[R4],#0400C7FF,@W^D157(R6)
DBG>DEPOSIT/INSTRUCTION .='NOP'
```

Examination of the locations reveals that the desired instruction sequence is intact:

```
DBG>EXAMINE/INSTRUCTION    BEGIN+12:TEST_SEQ
SORT\BEGIN+12: ADDL2 #10,R2
SORT\BEGIN+14: NOP
SORT\TEST_SEQ: CMPB (R0)[R2],(R0)[R4]
```

The DEPOSIT command can accept an instruction sequence for entry, but, as for any other command, you usually must reenter the entire command if you make an error. The following command sequence is a suggested method that allows you to enter a series of instructions by independent DEPOSITs without having to compute the actual address in each case.

```
DBG>SET MODE INSTRUCTION

DBG>DEPOSIT address-expression='instruction n'

DBG>EXAMINE

DBG>DEPOSIT . = 'instruction n+1'

DBG>EXAMINE

DBG>DEPOSIT . = 'instruction n+2'
```

Each EXAMINE command increments the previous address by the number of bytes required for the entered instruction and thus sets up the current address symbol (.) with the correct address for the next DEPOSIT command. The deposit of the NOP instruction in the previous example illustrates this method.

5.4.4 Evaluating VAX-11 MACRO Literals

When the debugger displays data in symbolic mode, it does not translate literal values into their symbolic equivalents. Thus, a displayed instruction may not appear exactly as you entered it in the source code. For example, the instruction

```
ADDL3 #literal-value, R0, R1
```

is displayed as

```
ADDL3 #3F4, R0, R1
```

if literal-value was previously assigned the value 3F4.

The EVALUATE command can help you quickly verify that the instructions are the same. If you type

```
DBG>EVALUATE/LITERAL expression
```

The debugger displays all pathnames it finds that have the value of the expression as their literal assignment. It is then a simple matter to scan the pathname list for the literal symbol name you wish to verify.

5.4.5 ASCII/NOASCII Modes

The ASCII/NOASCII modes allow or inhibit the entry and display of data as ASCII characters. ASCII means interpret or display the data as ASCII characters. NOASCII means do not interpret the input as ASCII or do not display the current data as ASCII. The debugger is initialized in NOASCII mode.

ASCII character input is usually by quoted string. You must enclose each string with either apostrophes or quotation marks. This provision lets you include literal apostrophes or quotation marks within a string. For example,

```
DBG>DEPOSIT /ASCII  WINK  = 'ZZZZ'
DBG>DEPOSIT /ASCII THINK  = "IT'S"
DBG>DEPOSIT /ASCII PLINK  = '"1"'
```

The delimiter at the string's end must match the beginning delimiter, and must not appear within the string.

The current length mode (LONG, WORD, or BYTE) is overridden by the length of the string.

Nonprinting ASCII characters (carriage return, line feed, horizontal tab, etc.) must be entered as numeric equivalents. For example, you can enter a carriage-return, line-feed combination between strings as follows.

```
DBG>DEPOSIT/ASCII/HEXADECIMAL/WORD  TEXT="IT'S",0D0A, "NEXT LINE"
```

The debugger enters the value 0D0A into memory following the string, IT'S. By specifying /WORD, you ensure that the value 0D0A is not deposited as a longword value.

You can verify the presence of the nonprinting characters by displaying the contents of the specific locations. Note that the debugger displays numeric data in the order that it resides in memory (that is, the contents of lower addresses appear in the least significant digits) whereas it displays ASCII characters in a left-to-right reversal of the actual character storage.

5.5 RADIX MODES

The radix mode keywords are:

```
DECIMAL
HEXADECIMAL
OCTAL
```

The base used in performing arithmetic operations depends on the radix mode specified. The radix mode also determines how numeric values are entered and displayed. You can use the SET MODE command to specify the radix mode. For example:

```
DBG>SET MODE DECIMAL
```

Numeric values specified in subsequent commands will be interpreted as decimal values, and numeric displays will also be in decimal, unless you override the current radix mode by including a radix mode keyword with the command. For example:

```
DBG>EVALUATE/HEX 15+15

0000002A
```

You can also specify that data be entered in a specific radix, by using a radix operator. For example:

```
DBG>EV  ^X15 + ^X15

42
```

Note that the resulting value is displayed in the current radix mode (in this example, decimal). See Section 4.1.7 for information on radix operators.

5.5.1 DECIMAL Mode

In DECIMAL mode, the debugger interprets entries and displays information in the decimal radix. A decimal entry can include the characters 0 through 9. With the debugger set for VAX-11 MACRO, you can use the radix operator ^D to identify individual entry arguments as decimal when the current radix mode is set to another radix.

5.5.2 HEXADECIMAL Mode

In HEXADECIMAL mode, the debugger interprets entries and displays information in the hexadecimal radix. A hexadecimal entry can include the characters 0 through 9 and A through F. You can use the radix operator ^X to identify individual entry arguments as hexadecimal when the current radix mode is set to another radix.

If an entry has an alphabetic as the leftmost character, you must include a leading zero or use the hexadecimal radix operator to differentiate hexadecimal constants from symbols.

5.5.3 OCTAL Mode

In OCTAL mode, the debugger interprets entries and displays information in the octal radix. An octal entry can include the characters 0 through 7. With the debugger set for VAX-11 MACRO, you can use the radix operator ^O to identify individual entry arguments as octal when the current radix mode is set to another radix.

5.6 LENGTH MODES

The length mode keywords are:

LONG (for longword)
WORD
BYTE

The current length mode specifies the value (4, 2, or 1) by which the debugger increments memory addresses for the entry or display of data forms other than VAX-11 MACRO instructions. In INSTRUCTION mode, the debugger ignores the current length mode and increments memory as a function of each instruction's storage requirements.

In NOINSTRUCTION mode, the results of all arithmetic operations are limited to value ranges corresponding to the current length mode. The debugger truncates values that exceed the current length mode by discarding most significant bit positions.

5.7 PATHNAME SEARCH MODES

The pathname search mode keywords are:

GLOBAL and NOGLOBAL
SCOPE and NOSCOPE

The following sections summarize the use of these mode keywords. For a complete description of their use and how such use relates to the debugger's search rules for translating pathnames to values, refer to Chapter 6.

5.7.1 GLOBAL/NOGLOBAL Modes

In GLOBAL mode, the debugger searches its symbol table for a symbolic match to the pathname as you entered it (that is, it does not prefix a scope to the entry). In NOGLOBAL mode, the debugger prefixes the

entry with the contents of scope, or a PC-implied scope, and searches the symbol table. If it does not find a match, it then attempts to find a match for the entry exactly as you entered it.

5.7.2 SCOPE/NOSCOPE Modes

In SCOPE mode, the debugger prefixes a pathname entry with the current contents of SCOPE and searches the symbol table for a match. If the SCOPE prefix fails or is not applied (NOSCOPE), the debugger prefixes a pathname entry with the name of the module that the program counter (PC) currently points to and searches for a match.

The debugger initializes SCOPE with the name of the first module read into the symbol table. You can then use the SET SCOPE, SHOW SCOPE, and CANCEL SCOPE commands to set, display, or delete the contents of SCOPE.

See Figure 6-1 for a diagram of the algorithm followed by the debugger in resolving references to symbols.

CHAPTER 6

SYMBOLS AND PATHNAMES

Pathnames symbolically represent address values that refer to locations in your program. This chapter describes how symbol information is entered in the debugger's symbol table for your use in specifying pathnames. The chapter also describes how you control both the contents of the symbol table and the manner in which the debugger translates a pathname into a value or a value into a symbolic expression.

6.1 PATHNAMES

The debugger uses pathnames to symbolically identify locations to avoid the possibility of ambiguous local symbols in a multimodule program. A pathname has two components: scope and symbol. The form of a pathname is:

```
scope\symbol
```

Scope identifies the module in which the symbol is unique (the backslash separates scope and symbol in a pathname).

If a symbol reference is unambiguous, you can ignore the scope of a pathname and simply specify the symbol in the debugger command. The search rules that govern the debugger's translation of a symbol into a value guarantee that the specified location or data element will be accessed. If, however, you specify ambiguous symbols, the debugger must know (or default) the scope of a symbol so that it can access the correct location or data.

The debugger's symbol-to-value algorithm defines how and when scope is expressed explicitly or implicitly. See Section 6.4.

In VAX-11 MACRO, the module name performs the scope function for local symbols. Global symbols, symbols that you define at debugger run time, and the debugger's permanent symbols do not require a regional identity since they are, by definition, known throughout the program (and to the debugger). They have a global scope and their pathnames are simply their symbol names.

Since a pathname is equivalent to the address for the location it represents, you can specify a pathname in any expression. For example, appending a numeric offset to a pathname creates an address expression that identifies an unlabeled location.

6.2 SYMBOL TYPES

The symbol types you can use are:

- Permanent symbols
- Symbols you create with the DEFINE command
- Your program's local symbols
- Your program's global symbols

A symbol is usually regarded as a way to specify a value. For example, symbolic labels point to locations in your program. However, most data symbols represent a sequence of bytes, that is, a range of values. A PSECT name, for example, refers to an entire program section within a given module.

While the concept of a symbol representing an address pair is not always pertinent (after all, you can't deposit a single value into an entire program section with one command), it is a useful concept for many cases, particularly when dealing with a language such as FORTRAN: Both EXAMINE and DEPOSIT work with a range of bytes for all standard FORTRAN data types. In VAX-11 MACRO, SET WATCH and translation from a value to a symbol also depend on this view of symbols.

6.2.1 Permanent Symbols

The debugger has the following VAX-11 abbreviations as permanent symbols. They cannot be redefined.

- R0 - R11 General registers 0 - 11
- AP Argument pointer
- FP Frame pointer
- SP Stack pointer
- PC Program counter
- PSL Processor Status Longword

Refer to Chapter 14 for information on the Processor Status Longword.

6.2.2 Defining Symbols During a Debugging Session

The DEFINE command lets you define global-type symbols at any time during a debugging session to supplement or override existing symbols in your program. The command format is:

```
DBG>DEFINE symbol=expression[,symbol=expression ...]
```

Symbol is a name, and expression is any valid expression. Symbols appearing in an expression must be resident in the debugger's symbol table. You must separate multiple symbol-expression pairs with commas.

You can, for example, create explicit symbols for unlabeled locations and/or assign various data values to symbols for ease of reference

during the session. The debugger always searches these symbol definitions first when it translates a symbolic entry into a value or when it translates an address location into a symbolic equivalent in SYMBOLIC mode. Because the debugger treats these defined symbols as first priority global symbols, your defined symbols have precedence over all other symbols in your program having identical names and/or definitions. This priority lets you create a symbolic shorthand.

For example, rather than request a breakpoint as

```
DBG>SET BREAK LOOP3\SILO_3
```

you could define the location to be BP7 with the command

```
DBG>DEFINE BP7=LOOP3\SILO_3
```

BP7 assumes the value of LOOP3\SILO_3. You can now request the breakpoint as follows:

```
DBG>SET BREAK BP7
```

When your program stops at the breakpoint, the debugger reports the location by:

```
break at pc = BP7
```

You can disable the shorthand notation, or determine the symbolic value of BP7, by redefining it, and then causing the breakpoint to be displayed. For example:

```
DBG>SET B BP7 DO(DEF BP7=BP7-BP7)
DBG>GO
start pc is BP7
break at pc = BP7
DBG>GO
start pc is LOOP3\SILO_3
```

The debugger requires that the definition of a user-defined symbol exactly match the address value if the debugger is to report that symbol as the symbolic equivalent rather than a global or local symbol. The section on translating output values into pathnames describes the complete rules on translation priority. See Section 6.5.

Symbolic names follow the VAX/VMS conventions:

- No more than 15 characters.
- Include only characters from the character set: A - Z, 0 - 9, dot (.), underline (_), and dollar sign ($). The debugger interprets lowercase alphabetics to be uppercase. Thus, "LOOP" and "loop" are the same to the debugger.
- Begin with an alphabetic character, underline, or dollar sign.

The debugger truncates a symbol that exceeds 15 characters to the 15 leftmost characters and issues a message.

The values that you assign to symbols must observe the following limits.

An unsigned value must be within the following ranges:

- Decimal: 0 - 4294967295
- Hexadecimal: 0 - FFFFFFFF
- Octal: 0 - 37777777777

A signed value must be within the following ranges:

- Decimal: - 2147483648 <= value <= 2147483647
- Hexadecimal: -80000000 <= value <= 7FFFFFFF
- Octal: -20000000000 <= value <= 17777777777

Additional restrictions for values are:

- Precede a hexadecimal value with either 0 or the radix operator ^X if the first character is alphabetic, A - F (otherwise, the debugger tries to interpret the character string as a symbol).
- Do not include commas within the value (2024; not 2,024).

A symbol cannot be canceled once you have defined it, but you can redefine it by specifying a different value with the DEFINE command. For example, the previous definition of BP7 to represent the location LOOP3\SILO_3 could be changed as follows.

```
DBG>DEFINE BP7=SORT\TEST_END
```

You can also redefine a symbol created by a DEFINE command, in terms of its current definition. For example, you can redefine the symbol LOOP to represent a value of 1000 plus its current definition.

```
DBG>DEFINE loop=loop+1000
```

NOTE

The DEFINE command can not, under any circumstances, be used to redefine or create a multi-element pathname.

With the EVALUATE command, you can determine the current definition of any symbol and express it in any radix. For example:

```
DBG>EVALUATE [/radix-mode-keyword] symbol
```

You cannot, however, translate a numeric value to learn its symbolic equivalent(s).

Refer to the description of the EVALUATE command (Chapter 11) for more information on its use. Refer also to Chapter 4 for information on the use of the debugger's special characters in expressions to create values for your defined symbols.

6.2.3 Local Symbols

Local symbol information includes:

- Module names assigned by the VAX-11 MACRO directive, .TITLE (they can be used only as the scope of pathnames, because they have no values)
- Program section names assigned by the VAX-11 MACRO directive, .PSECT (program section names assigned by default are not normally accessible)
- All symbols and associated definitions not identified as being global, but not "n$" type symbols

For the debugger to have access to local symbol information, you must:

- Request that the assembler produce debugging records when it assembles each module (/ENABLE=DEBUG)
- Use the /DEBUG qualifier at link time
- At run time, use SET MODULE to ensure that symbol information for a particular module is present in the symbol table when you intend to specify any local symbols from that module in pathnames or you want the debugger to represent any of its locations by local symbol pathnames

If you prefer to enter or have the debugger display locations as program section names plus offsets (some programmers find this convenient when referring to assembly listings), you can request that the image contain only traceback records. This request limits the debugger's access to only module and program section names.

6.2.4 Global Symbols

Global symbols include:

- Those symbols identified as labeling external definitions in a module by the VAX-11 MACRO directive, .GLOBL
- Those symbols delimited by the double colon (::) operator
- Those symbols that label global literals (that is, those delimited by the double equal (==) operator)

The debugger references global literals only when translating a pathname into a value. It ignores them in the translation of a value into a pathname. You can, however, determine the correspondence of a value to a global literal name by use of the EVALUATE command (see Chapter 11).

You can access your program's global symbols only if you specified /DEBUG when you linked your program. Refer to the VAX-11 Linker Reference Manual and the VAX-11 MACRO User's Guide for information on how to make global symbols available to the debugger.

6.3 THE DEBUGGER'S SYMBOL TABLE

The debugger translates pathnames into values and values into symbolic expressions on the basis of information in its symbol table. The debugger has no knowledge of symbol information not present in this table. After the debugger is initialized, this information consists of permanent symbols, any symbols created by DEFINE commands, all global symbols, and local symbol information for the first module in your image, if you specified the appropriate qualifier when you compiled or assembled the source program. For example, for a VAX-11 MACRO program, /ENABLE=DBG.

When you initiate the debugger, it establishes a data base for the modules in your program and reads symbol information from the first module into the symbol table.

The following sections describe how you control the symbol table contents with the SET MODULE, SHOW MODULE, and CANCEL MODULE commands.

6.3.1 Symbol Table Input (SET MODULE)

The command

```
DBG>SET MODULE module-name[,module-name,...]
```

tells the debugger to add local symbol information for the specified module(s) to the symbol table. Rather than specify individual modules, you can request that all symbol information for all modules be entered in the symbol table by specifying:

```
DBG>SET MODULE/ALL
```

If the debugger is not able to include some of the modules, it prints a message indicating those modules that were not included.

6.3.2 Symbol Table Status Report (SHOW MODULE)

The command

```
DBG>SHOW MODULE
```

produces a status report on the symbol table. The report lists all modules in the program and indicates by yes or no whether their associated local symbol information is currently present in the symbol table. The report also includes the following information:

- the approximate number of bytes (in decimal) required to accommodate the entry of symbol information from the respective modules into the table
- the total number of modules in your program
- the amount of free bytes (in decimal) available in the table
- the name of the language in which the modules were written. If the same language was used for all modules, the language name appears only in the line that indicates the total number of modules.

6.3.3 Symbol Table Purging (CANCEL MODULE)

The command

```
DBG>CANCEL MODULE module-name[,module-name, ...]
```

purges symbol information associated with the specified module(s) from the symbol table. Typically, it is used to make space available for symbol information associated with other modules. The CANCEL MODULE command does not affect global symbols or symbols that you defined during this debugging session.

You can delete all local symbol information currently in the symbol table by the command

```
DBG>CANCEL MODULE/ALL
```

6.4 TRANSLATING SYMBOLS INTO VALUES

The debugger's translation of symbolic entries into values is governed by the GLOBAL/NOGLOBAL and SCOPE/NOSCOPE modes, which give you control of the debugger's search rules. Figure 6-1 illustrates the search algorithm.

The debugger evaluates an expression in which a symbolic entry appears only if a definition was located for the entry under the search rules. If it fails to locate a match for a pathname, the debugger reports the search failure and the symbol name. The expression becomes undefined.

If you specify GLOBAL, the debugger first assumes the symbolic entry represents the entire pathname, and tries to find a match for that pathname. If the search is unsuccessful, or if NOGLOBAL is in effect, the debugger then tries all the other search possibilities.

If you specify NOGLOBAL, the debugger assumes that the symbolic entry represents the entire pathname only after first trying all other search possibilities.

You might set the mode to GLOBAL if your program contained a global symbol and a local symbol with the same names, and you wanted to set a breakpoint (or execute any debugger command) at the global symbol location.

Another use would be when you are working in one of your program's modules and want to execute the debugger command in another module without having to change the current contents of SCOPE. In this case, you would enter the complete pathname, module name and local symbol, in the command.

For example, this sequence

```
DBG>EVALUATE/GLOBAL  SORT\SEQ_CHECK
00003AF2
DBG>SET BREAK   \
```

makes the debugger determine the value of the given pathname and display it. The display in turn assigns the value to the last value displayed symbol (see Section 4.2.1) for use as the operand in the SET BREAK command.

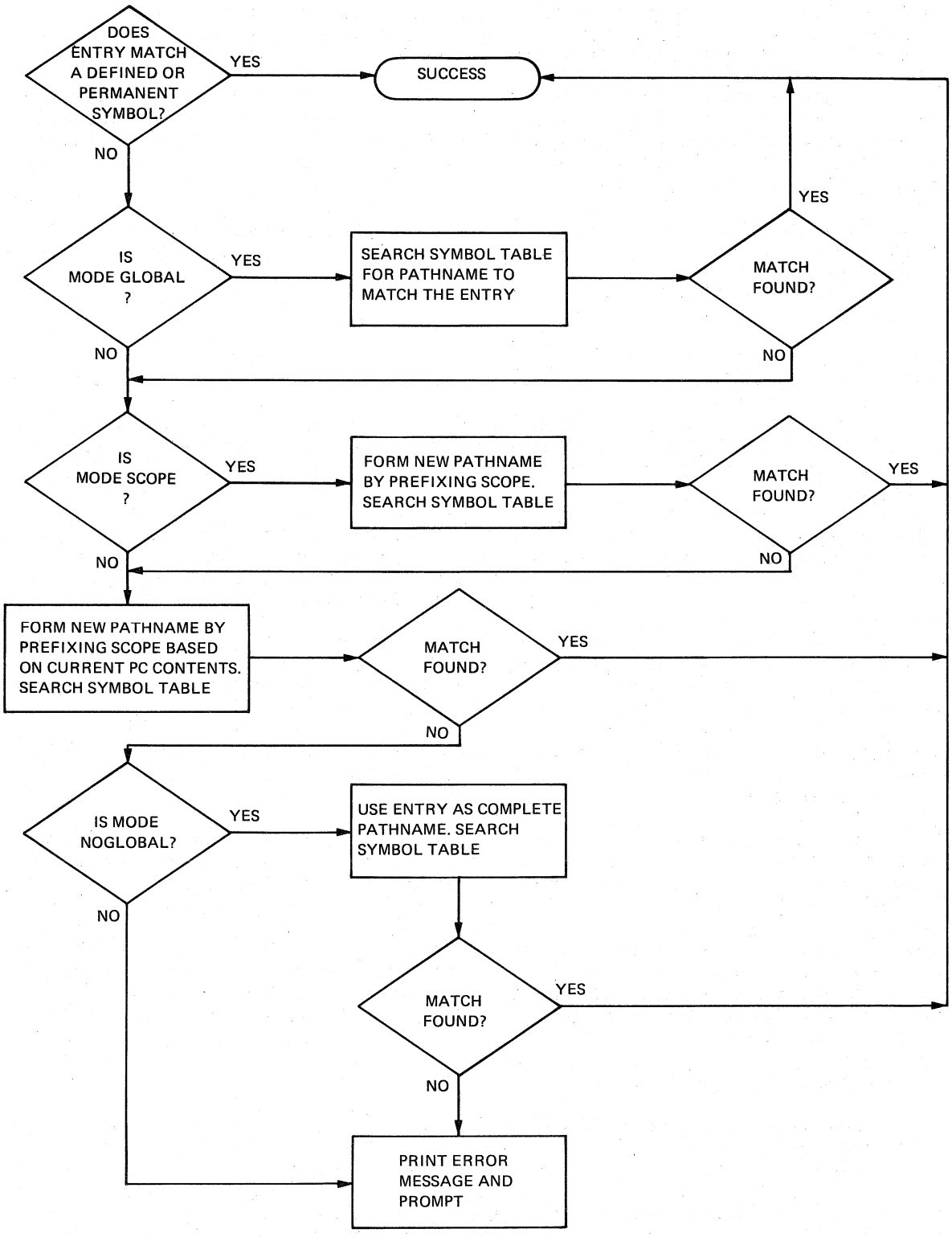

Figure 6-1 Debugger Symbol-to-Value Search Algorithm

NOSCOPE tells the debugger that you do not want the current contents of SCOPE to be prefixed to a pathname entry. The debugger then uses the name of the module that the program counter (PC) is currently pointing to as the scope, and searches the symbol table for a local symbol from that module that matches your entry.

The sequence

```
DBG>EVALUATE/NOSCOPE  SEQ_CHECK
00003AF2
DBG>SET BREAK \
```

directs the debugger to construct a pathname by prefixing the name of the currently executing module to SEQ_CHECK, determine the value of this pathname, and display it. The display, as before, assigns the value to the last value displayed symbol for use as the address in the SET BREAK command.

The command

```
DBG>SET SCOPE module-name
```

establishes the specified module name as the explicit scope to be used under the search rules for the translation of local symbol and program section name pathnames. The debugger also reads symbol information associated with the specified module into the symbol table if the necessary number of bytes are free in the table. If the table lacks the number of bytes required to accommodate the module's symbol information, the debugger aborts the SET SCOPE command and prints a message.

The command

```
DBG>SHOW SCOPE
```

requests that the debugger report the current contents of SCOPE. A <null> report indicates that the SCOPE rule has no effect in looking up symbols.

The command

```
DBG>CANCEL SCOPE
```

enters a null string in SCOPE. If you subsequently specify SET MODE SCOPE, the previous contents of SCOPE are restored.

6.5 TRANSLATING VALUES INTO PATHNAMES

In SYMBOLIC mode, the debugger translates an address value into a pathname as follows.

1. The debugger first compares the value with its permanent symbol definitions, then with the symbol definitions, if any, that you created with the DEFINE command. If it locates an exact match (no offset permitted), the debugger reports the found symbol as the pathname.

2. If step 1 fails, the debugger compares the value with the global and local symbol definitions. A global symbol definition is sought only if no local definition is found. If an exact match is found, the debugger reports the symbol as the pathname.

3. If no exact match can be found, the debugger searches all symbol definitions for the one that is nearest to, yet less in value than, the value to be translated, and expresses the initial value as that pathname plus the necessary offset. The debugger rejects a global symbol definition as being the nearest to the value unless the difference between the symbol and the value is less than 100 (hexadecimal).

4. If the debugger does not find a suitable definition by means of steps 1, 2, and 3, it reports the address value as a virtual address in the current radix mode. The probable cause of the virtual address display rather than a pathname is that the respective module's symbol information is not present in the debugger's symbol table.

CHAPTER 7

BREAKPOINTS

Breakpoints stop your program at selected locations to let you observe and change the context of your program while it is suspended. This chapter describes breakpoints, their options (command sequences to be executed at the breakpoint, deferred breakpoints, and temporary breakpoints), and how you use the commands, SET BREAK, SHOW BREAK, and CANCEL BREAK, to establish, report the status of, and delete breakpoints.

7.1 USE OF BREAKPOINTS

Without breakpoints, your program might run to completion, exit prematurely, or enter an infinite loop, depending on the type of errors it contains. Your observations during testing would be limited to an analysis of data produced, if any, and possibly a general register dump if your program exited prematurely because it violated system restrictions.

A breakpoint can be specified with several options. They include:

- The option to specify a sequence of commands that the debugger executes automatically each time your program stops at the associated breakpoint.
- The option to ignore a breakpoint until it has been encountered a specified number of times.
- The option to specify a temporary (or one-time) breakpoint. The debugger automatically cancels the breakpoint after your program stops at the breakpoint location.

7.1.1 Breakpoint Reporting at Program Stop

When your program is suspended at a breakpoint, the debugger usually reports the location by

```
break at pc = LOCATION
```

where the location is given symbolically (SYMBOLIC mode) or as a virtual address in the current radix mode (NOSYMBOLIC mode). For example, a breakpoint occurrence could be reported as

```
break at pc = SORT\INSEQ
```

where SORT\INSEQ is the pathname that uniquely identifies the location

labeled by local symbol INSEQ in the object module named SORT. In NOSYMBOLIC mode, the location would be reported by

```
break at pc = 00000846
```

The debugger sometimes displays the report as

```
routine break at pc = LOCATION
```

Note that in this case the value shown is 2 less than the actual PC contents. This is the case whenever symbolic information is available indicating that a location or symbol is an entry point or the beginning of a routine.

7.1.2 Continuing From a Breakpoint

To continue your program from a breakpoint, you can enter a GO command or a STEP command. After GO, program execution continues until either a breakpoint or another condition causes the program to stop. After STEP, program execution continues either through the number of steps you specified (the default is one) or until some condition causes the program to stop.

The debugger usually reports the resumption of program execution by

```
start pc is location
```

where "location" is again given as a pathname, or as a virtual address. If the report is displayed as "routine start pc is location", the value of "location" is actually 2 less than the contents of the PC, and "location" is an entry point.

7.2 SETTING BREAKPOINTS

Breakpoints are set at and identified by address. Once set, a breakpoint remains active until you cancel it or terminate the debugging session. No breakpoints are set when you begin the session.

The debugger's breakpoint table stores the information relating to each breakpoint. This table can accommodate many breakpoints. If the debugger reports a full table, simply cancel one or more breakpoints to clear sufficient table space for the new entry.

The debugger does not protect current breakpoints against overwriting by a new request. The debugger simply replaces the previous specification with the new command entry without warning. This condition works to your advantage when you want to modify a breakpoint specification. Instead of having to cancel a breakpoint and then specify the new conditions for the breakpoint, you can just enter the new specification for the same location.

7.2.1 General Breakpoint Specification

You set a breakpoint at the address of the first byte of an instruction (your program stops at the breakpoint before executing the instruction). The debugger accepts the address specification without verifying that it represents the first byte of the instruction's storage.

Warning: Run-time errors usually result if a breakpoint is set in the middle of an instruction.

The general command format for specifying a breakpoint is:

```
DBG>SET BREAK address-expression [DO (command-list)]
```

To verify the breakpoint, you can use the "current location" symbol (see Section 4.2.1) as follows:

```
DBG>EXAMINE/INSTRUCTION
```

The debugger displays the instruction on which the breakpoint is set.

7.2.2 DO Command Sequence at Breakpoint

When specifying a breakpoint, you can include a sequence of commands that the debugger executes whenever your program stops at the breakpoint. The command format is

```
DBG>SET BREAK address-expression DO(command[;command...])
```

The command list can include any complete debugger command. If a GO, STEP, or CALL command is included, it must be the last command in the sequence. The parentheses are required regardless of the number of commands specified. The semicolon is not necessary if you include one command. For DO sequences that comprise more than one command, you may want to use line continuation (a hyphen as the last character before carriage return) and/or abbreviated keywords.

The debugger does not evaluate a DO command sequence for proper syntax or context until your program stops at the breakpoint.

Note that a symbol that appears in a DO command sequence needn't be defined at the time you enter the SET BREAK command, because the debugger defers binding symbols and values until the breakpoint is encountered. You can define the symbol at any point prior to that time.

You can nest SET BREAK commands within DO command sequences. For example:

```
DBG>SET B LOOP DO (E/BYTE BUF:BUF+^X10;SET B LOOP2 -
_DO (E/WORD BUFX+4))
```

The sequence above shows one level of SET BREAK DO nesting. You can extend this nesting to any level, as long as you ensure that the initiating and terminating parentheses match.

All command sequences are executed in the context in effect when the breakpoint occurs.

To cancel or alter the DO command sequence, enter a new SET BREAK command with the desired content. If you cancel a breakpoint, any associated DO command sequence is also canceled.

7.2.3 Breakpoint "After" Option

If your program is to stop only after the nth pass through a breakpoint location, as in an iteration or conditional program loop, specify the breakpoint as follows:

```
DBG>SET BREAK/AFTER:n address-expression
```

where n is a decimal integer in the range 1 through 32767.

Once an "after" breakpoint has stopped your program, it will continue to stop your program each time it is encountered until you cancel it (that is, the breakpoint functions as if the count is 1). You can include the "after" option in any breakpoint specification.

The SHOW BREAK command (see below) displays an "after" count for a breakpoint only if it is other than 1; that is, the debugger must see the location n more times before the breakpoint takes effect.

7.2.4 Temporary Breakpoints

A temporary (or one time) breakpoint stops your program once and then is canceled automatically. You specify such a breakpoint by

```
DBG>SET BREAK/AFTER:0 address-expression [DO (command)]
```

The breakpoint status report produced by SHOW BREAK (see below) lists a temporary breakpoint (by displaying /AFTER:0) until the debugger executes it.

7.3 SHOWING BREAKPOINTS

You can determine where current breakpoints are set, along with a description of any breakpoint actions that were specified, by typing:

```
DBG>SHOW BREAK
```

The debugger responds with:

```
breakpoint/after:n at location etc.
breakpoint at location do-command-sequence, etc.
   .   .   .
```

The debugger identifies the breakpoint locations by pathnames (SYMBOLIC mode) or by virtual address (NOSYMBOLIC mode) in the current radix mode (decimal, hexadecimal, or octal).

If the debugger does not find any breakpoints, it displays the appropriate message.

7.4 CANCELING BREAKPOINTS

You cancel a breakpoint when you no longer want it to stop your program. All breakpoints are automatically canceled when you end the current debugging session.

To cancel a specific breakpoint, type:

```
DBG>CANCEL BREAK address-expression
```

When canceling a breakpoint, you can not identify DO command sequences or options that were previously established for the breakpoint. An address expression of the correct value is sufficient information.

If the debugger cannot find a specified breakpoint, it prints a message.

To cancel all breakpoints, type

```
DBG>CANCEL BREAK/ALL
```

7.5 BREAKPOINT EXAMPLES

The following examples illustrate use of the SET, SHOW, and CANCEL BREAK commands.

7.5.1 Examples of Setting Breakpoints

```
DBG>SET BREAK TERMINAL_IO\BEGIN+30
```

Sets a breakpoint at the location 30 bytes after the location identified by the pathname TERMINAL_IO\BEGIN (the debugger interprets the value 30 in the current radix mode).

```
DBG>SET BREAK/AFTER:6 SORT\SEQCHK
```

Sets a breakpoint at the location identified by the pathname SORT\SEQCHK. The debugger does not stop your program until the sixth pass through this location.

```
DBG>SET BREAK SORT\INSEQ DO (EXAMINE/ASCII/BYTE @R7:@R7+^D10)
```

Sets a breakpoint at the location identified by the pathname SORT\INSEQ. The debugger executes the DO command sequence after the program stops at this breakpoint. The sequence tells the debugger to report as ASCII characters the contents of the eleven bytes beginning with the location that is indirectly addressed by the contents of general register R7.

```
DBG>SET BREAK ^X7249
```

Sets a breakpoint at virtual address 7249 (hexadecimal).

7.5.2 Examples of Showing Breakpoints

1. In SYMBOLIC mode (the initialized condition):

```
DBG>SHOW BREAK
routine breakpoint at SORT\INSEQ do(set scope inseq)
breakpoint at SORT\SEQCHK do (examine BUF:BUF+6,R8,COUNT)
```

The debugger reports the current breakpoint locations and associated DO sequences.

2. In NOSYMBOLIC mode:

```
DBG>SHOW BREAK
breakpoint at 0000846
breakpoint at 000082A do (examine BUF:BUF+6,R8,COUNT)
```

The debugger reports the breakpoint locations as virtual addresses in the current radix mode (in this case hexadecimal).

7.5.3 Examples of Canceling Breakpoints

```
DBG>CANCEL BREAK TERMINAL_IO\BEGIN
DBG>CANCEL BREAK SORT\SEQCHK
DBG>CANCEL BREAK ^X7249
```

The debugger cancels the specified breakpoints.

```
DBG>CANCEL BREAK/ALL
```

The debugger cancels all breakpoints.

CHAPTER 8

TRACEPOINTS AND OPCODE TRACING

Tracing is the process of observing the sequence in which a program is executed. By using the SET TRACE command, you can monitor the order in which your program executes its instructions or statements. The debugger can let you know whether unanticipated control transfers are occurring as your program is running. There are two basic forms of tracing: tracepoints, and tracing on opcodes.

A tracepoint is similar to a breakpoint. When your program reaches a tracepoint, it momentarily suspends execution and reports the tracepoint. It then automatically resumes execution. Thus you can see if your program is reaching specified locations in the correct sequence.

Tracing on opcodes means requesting that the debugger report the occurrence of each instruction of a specified type, such as call-type instructions and branch-type instructions.

8.1 USING THE TRACE FACILITY

You can specify tracing as follows:

- At the first byte of specified instruction locations (that is, set tracepoints).
- At all call-type instructions in your program (includes all CALLG, CALLS, RET, JSB, BSBW, BSBB, and RSB instructions).
- At all branch-type instructions in your program (includes all branches and JMP; excludes subroutine-type instructions).
- At both call-type instructions and branch-type instructions.

Tracing degrades the performance of your program. If this concerns you, enter breakpoints with DO command sequences that include GO as the last (or only) command instead of using tracing (see SET BREAK examples, Chapter 7).

At a tracepoint, the debugger reports the location and then allows your program to proceed automatically. The report has the form:

```
trace at pc = location : instruction
```

where location is given symbolically or as a virtual address, and

instruction is the instruction at the location shown. For example, a tracepoint occurrence could be reported as:

```
trace at pc = SORT\INSEQ :  CMPB (R0)[R2],(R0)[R4]
```

where SORT\INSEQ is the pathname that represents the location addressed by the program counter and CMPB (R0)[R2],(R0)[R4] is the instruction at that location. In NOSYMBOLIC mode, the location would be reported by

```
trace at pc = 00000846 :  CMPB (R0)[R2],(R0)[R4]
```

If the message is displayed as

```
routine trace at pc = location : instruction
```

the value of location is actually 2 less than the current PC, and location is an entry point or the beginning of a routine.

8.2 SETTING TRACEPOINTS

Once set, a tracepoint remains until you either cancel it or terminate the debugging session. No tracepoints are set when you begin the debugging session.

The debugger's tracepoint table stores the information relating to each tracepoint. This table can accommodate a large number of tracepoints. If the debugger reports a full table, simply cancel one or more tracepoints to clear sufficient table space for the new entry.

8.2.1 Individual Tracepoints

You set a tracepoint by specifying a command in the form:

```
DBG>SET TRACE address-expression
```

You must be sure that address-expression is the first byte of an instruction. (The debugger does not verify the validity of address-expression.)

To verify a tracepoint you can use the "current location" symbol, as follows:

```
DBG>EXAMINE/INSTRUCTION
```

The debugger displays the instruction on which the tracepoint is set.

8.2.2 Tracing All Call-Type Instructions

To trace all call-type instructions, specify:

```
DBG>SET TRACE/CALL
```

8.2.3 Tracing All Branch-Type Instructions

To trace all branch-type instructions, specify:

```
DBG>SET TRACE/BRANCH
```

8.2.4 Tracing All Call-Type and Branch-Type Instructions

To trace both forms of control transfer instructions, simply enter both forms of SET TRACE commands in either order. For example:

```
SET TRACE/BRANCH

SET TRACE/CALL
```

8.3 SHOWING TRACING MODES

You can determine where tracepoints are set, and the form of tracing in effect by using the command

```
DBG>SHOW TRACE
```

The debugger responds with:

```
tracepoint at location
tracepoint at location
tracing /CALL instructions: list-of-opcodes
tracing /BRANCH instructions: list-of-opcodes
```

The debugger identifies the tracepoint locations by pathnames or by numeric virtual address in the current radix mode (decimal, hexadecimal, or octal).

If the debugger does not find tracepoints set, and no opcode tracing is in effect, it prints a message.

8.4 CANCELING TRACING

You can cancel a tracepoint when you no longer want to monitor a program location. You can also disable one or both forms of opcode tracing. All tracing is automatically canceled when you end the current debugging session.

To cancel a specific tracepoint, type:

```
DBG>CANCEL TRACE address-expression
```

To cancel call-type instruction tracing, type:

```
DBG>CANCEL TRACE/CALL
```

To cancel branch-type instruction tracing, type:

```
DBG>CANCEL TRACE/BRANCH
```

To cancel all tracepoints and opcode tracing, type:

```
CANCEL TRACE/ALL
```

8.5 TRACING EXAMPLES

The following examples illustrate the SET, SHOW, and CANCEL TRACE commands.

8.5.1 Examples of Setting Tracepoints

```
DBG>SET TRACE TERMINAL_IO\BEGIN+30
```

Sets a tracepoint at the location 30 bytes after the location identified by the pathname TERMINAL_IO\BEGIN (the debugger interprets the value 30 in the current radix mode).

```
DBG>SET TRACE ^X7249
```

Sets a tracepoint at virtual address 7249 (hexadecimal).

8.5.2 Examples of Showing Tracepoints

1. In SYMBOLIC mode (the initialized condition) the debugger reports the current tracepoint locations. For example:

```
DBG>SHOW TRACE
tracepoint at SORT\INSEQ
tracepoint at SORT\SEQCHK
```

2. In NOSYMBOLIC mode the debugger reports the tracepoint locations as virtual addresses in the current radix mode (in this case hexadecimal). For example:

```
DBG>SHOW TRACE
tracepoint at 0000846
tracepoint at 000082A
```

8.5.3 Examples of Canceling Tracepoints

```
DBG>CANCEL TRACE TERMINAL_IO\BEGIN

DBG>CANCEL TRACE SORT\SEQCHK

DBG>CANCEL TRACE ^X7249
```

The debugger cancels the specified tracepoints.

```
DBG>CANCEL TRACE/ALL
```

The debugger cancels all tracepoints and opcode tracing.

CHAPTER 9

WATCHPOINTS

Watchpoints are selected program locations you monitor to identify instructions that modify these locations. This chapter describes watchpoints and the use of the commands, SET WATCH, SHOW WATCH, and CANCEL WATCH, to establish, report the status of, and delete watchpoints.

9.1 USE OF WATCHPOINTS

If an instruction modifies a watchpoint location, the debugger stops your program after the instruction completes execution. The debugger then reports the watchpoint location, the location of the instruction, and both the previous and the current contents of the location being monitored.

The number of bytes monitored at a watchpoint depends on whether the location has a data type. For example, if the location is a double precision FORTRAN variable, eight bytes are monitored. However, if no data type is associated with the location (as in VAX-11 MACRO), four bytes are monitored. The current LENGTH mode is ignored.

9.1.1 Watchpoint Reporting

When your program writes into a watchpoint location, the debugger stops the program and reports the following:

```
write to location at pc = location
      old value = value
      new value = value
```

The "write to location" indicates the location that was modified. The "at pc = location" indicates the location of the instruction that did the writing.

The debugger reports the locations either symbolically or as virtual addresses; it reports the old (previous) value and the new (current) value in hexadecimal.

For example, a watchpoint modification could be reported as

```
write to TERMINAL_IO\OUTLENGTH at pc = TERMINAL_IO\MAIN_CODE+51
      old value = 000008A2
      new value = 00000000
```

where TERMINAL_IO\OUTLENGTH is the pathname that identifies the location labeled OUTLENGTH in module TERMINAL_IO, and

TERMINAL_IO\MAIN_CODE+51 is the pathname plus offset that identifies the location of the trapped instruction.

In NOSYMBOLIC mode, the locations are displayed as virtual addresses. For example:

```
write to 00000432 at pc = 000006A2
      old value = 000008A2
      new value = 00000000
```

Note that values are displayed in hexadecimal.

9.1.2 Continuing From a Watchpoint

To continue your program from a watchpoint, enter a GO command or a STEP command. After GO, program execution continues until a watchpoint or another condition causes the program to stop. After STEP, program execution continues either through the number of instructions you specified (the default is one instruction) or until some condition causes the program to stop.

The debugger reports the resumption of program execution by

```
start pc is location
```

where location is given either as a pathname or as a virtual address.

9.2 SETTING WATCHPOINTS

You specify a watchpoint request by,

```
DBG>SET WATCH address-expression
```

Once set, a watchpoint remains active until you either cancel it or terminate the debugging session. No watchpoints are set when you initialize the debugging session.

The debugger's watchpoint table stores the information relating to each watchpoint. The space allocation can accommodate a large number of watchpoints.

9.3 SHOWING WATCHPOINTS

You can determine where current watchpoints are set by typing

```
DBG>SHOW WATCH
```

The debugger responds with:

```
watchpoint at location for nnn bytes
watchpoint at location for nnn bytes
   .   .   .
```

The debugger identifies the watchpoint locations by pathnames (SYMBOLIC mode on) or by numeric virtual address (NOSYMBOLIC mode on) in the current radix mode (decimal, hexadecimal, or octal). The value nnn, in decimal, indicates how many bytes are monitored by the associated watchpoint.

9.4 CANCELING WATCHPOINTS

You can cancel a watchpoint when you no longer want to monitor the specified location(s). All watchpoints are automatically canceled when you end the current debugging session.

To cancel a specific watchpoint, type:

```
DBG>CANCEL WATCH address-expression
```

If you specify CANCEL WATCH/ALL, all watchpoints are canceled.

If the debugger cannot find the specified watchpoint, it displays a message.

9.5 WATCHPOINT EXAMPLES

The following examples illustrate the SET, SHOW, and CANCEL WATCH commands.

9.5.1 Examples of Setting Watchpoints

```
DBG>SET WATCH TERMINAL_IO\BEGIN
```

The debugger watches the location identified by the pathname, TERMINAL_IO\BEGIN.

```
DBG>SET WATCH ^X7249
```

The debugger watches virtual address 7249 (hexadecimal).

9.5.2 Examples of Showing Watchpoints

1. With SYMBOLIC MODE on (the initialized condition):

```
DBG>SHOW WATCH
watchpoint at SORT\INSEQ for 4.  bytes
watchpoint at SORT\SEQCHK for 2.  bytes
```

   The debugger reports the current watchpoint locations by pathnames.

2. With NOSYMBOLIC mode on:

```
DBG>SHOW WATCH
watchpoint at 0000846 for 4.  bytes
watchpoint at 000082A for 2.  bytes
```

   The debugger reports the watchpoint locations as numeric virtual addresses. The addresses are displayed according to the current radix mode.

9.5.3 Examples of Canceling Watchpoints

```
DBG>CANCEL WATCH TERMINAL_IO\BEGIN

DBG>CANCEL WATCH SORT\SEQCHK

DBG>CANCEL WATCH ^X7249
```

The debugger cancels the specified watchpoints.

9.6 WATCHPOINT RESTRICTIONS

When you set a watchpoint, the entire page containing the watchpoint location is protected. When an instruction attempts to write to any location on that page, at user mode level, the modification is made and execution continues unless the modification was to the watchpoint location. In this case, the debugger suspends execution and reports the old and new contents, and the location of the instruction that caused the change.

If a system service needs to write to a location on a protected page, it will return failure status. Therefore, you should not set watchpoints on pages that contain locations that may be modified by system software; for example, I/O status blocks subject to modification by Record Management Services.

CHAPTER 10

EXAMINE AND DEPOSIT COMMANDS

This chapter describes how to use the EXAMINE and DEPOSIT commands to display and change the contents of selected memory locations.

10.1 EXAMINING MEMORY LOCATIONS AND REGISTERS

The EXAMINE command displays the contents of selected memory locations and registers.

The command format is:

```
DBG>EXAMINE[/mode]address[:address][,address[:address]]
```

You can specify a value for /mode, to override the current modes, as described in Section 5.3.4.

You can use EXAMINE to display any combination of the following:

- A single location
- Multiple locations
- A range of contiguous locations
- Multiple ranges of locations

If you specify more than one address, and separate them with commas, the contents of the locations specified are displayed. However, if you use a colon to separate a pair of addresses, then all addresses within that range are displayed. For example

```
DBG>EXAMINE/WORD 1028,1040

00001028:  046b
00001040:  0EF40

DBG>EXAMINE/WORD 1028:1040

00001028:  046B
0000102A:  0000
0000102C:  08C2
0000102E:  0D7EF
00001032:  0FFF3
00001034:  0AEFF
00001036:  0D004
00001038:  04AE
0000103A:  9850
0000103C:  22A0
0000103E:  0D450
00001040:  0EF40
```

To specify multiple ranges, use a command such as:

```
DBG>EXAMINE/WORD 1028:102E,103A:1040
```

The results are:

```
00001028:  046B
0000102A:  0000
0000102C:  08C2
0000102E:  0D05E
0000103A:  9850
0000103C:  22A0
0000103E:  0D450
00001040:  0EF40
```

When you specify a range, you must specify the low address first. When you specify more than one individual location, you can use any order.

If you wish to display the next location, you needn't specify an address. Thus, after you've examined a location by specifying an address, you don't have to specify the next contiguous location. For example:

```
DBG>SET MODE WORD

DBG>EXAMINE 1028

1028:  046B

DBG>EXAMINE

102A:  0000
```

10.1.1 Examining Numeric Data

The following examples illustrate the use of EXAMINE to display the contents of a range of locations as hexadecimal data in the length modes, LONG, WORD, and BYTE, respectively.

```
DBG>SET MODE  HEXADECIMAL , LONG , NOINSTRUCTION , NOSYMBOLIC
DBG>EXAMINE    4000:4004

00004000:  0D0500AD0
00004004:  01D05000

DBG>EXAMINE/WORD 4000:4006

00004000:  0AD0
00004002:  0D050
00004004:  5000
00004006:  01D0

DBG>EXAMINE/BYTE 4000:4007

00004000:     0D0
00004001:     0A
00004002:     50
00004003:     0D0
00004004:     00
00004005:     50
00004006:     0D0
00004007:     01
```

The current contents of these locations could be displayed as instructions by

```
DBG>EXAMINE/INSTRUCTION 4000:4004

4000:   MOVL   #0A, R0
4003:   MOVL   #00, R0
```

The example above illustrates that the current length mode does not affect how the debugger increments memory to display the instructions.

10.1.2 Examining Instructions

The following example illustrates how EXAMINE displays the contents of several locations as VAX-11 MACRO instructions. For complete information on examining data as instructions, refer to Section 5.4.3.

```
DBG>EXAMINE/INSTRUCTION SORT\BEGIN+12 :  TEST_SEQ
SORT\BEGIN+12:          ADDL3   #10,R2,R4
SORT\TEST_SEQ:          CMPB    (R0)[R2],(R0)[R4]
```

In INSTRUCTION mode, the debugger ignores the current length mode and displays whatever storage the instruction occupies. With the exception of PC relative displacements, literals and displacements in instructions are displayed in the current radix mode. PC relative displacements are evaluated and displayed symbolically (SYMBOLIC mode) or as virtual addresses (NOSYMBOLIC mode).

10.1.3 Displaying Locations As ASCII Characters

The following example illustrates how EXAMINE displays the contents of a range of locations as ASCII characters. For complete information on examining data as ASCII characters, refer to Section 5.4.5.

```
DBG>EXAMINE/ASCII/LONG  CHARS:CHARS+^X13
CHARS:  IT'S
CHARS+4:  A "
CHARS+8:  SMAL
CHARS+0C:  L" W
CHARS+10:  ORLD
```

10.2 MODIFYING MEMORY LOCATIONS AND REGISTERS

The DEPOSIT command lets you alter the contents of memory locations and registers. The command format is:

```
DBG>DEPOSIT[/mode,...] address-expression=data[,data,...]
```

With DEPOSIT, you can enter data in one location or in several sequential locations beginning with a specified location.

10.2.1 Depositing Numeric Data

The following examples illustrate the entry of a hexadecimal value in a byte, a word, and a longword, respectively.

The suggested method is to first display the current contents of the location. For example:

```
DBG>EXAMINE    4000
00004000:  0D0500AD0
```

The byte of data is deposited and verified by,

```
DBG>DEPOSIT/BYTE   4000 = ^XFF
DBG>EXAMINE  .
00004000:  0D0500AFF
```

The word of data is deposited and verified by,

```
DBG>DEPOSIT/WORD   4000 = ^XFFFF
DBG>EXAMINE .
00004000:  0D050FFFF
```

The longword of data is deposited and verified by,

```
DBG>DEPOSIT     4000 = ^XFFFFFFFF
DBG>EXAMINE .
00004000:  0FFFFFFFF
```

The following example illustrates the entry and verification of data in an intermediate byte of a longword that initially contains 77777777.

```
DBG>SET MODE LONG , HEXADECIMAL
DBG>EXAMINE  4000
00004000:  77777777
DBG>DEPOSIT/BYTE   4002 = 0FF
DBG>EXAMINE 4000
00004000:  77FF7777
```

Note that a 0 must be used to prefix a hexadecimal number that starts with an alphabetic character.

10.2.2 Depositing Instructions

This section describes how to use DEPOSIT to enter data as instructions. For complete information on depositing instructions, refer to Section 5.4.3.

The storage requirements of VAX-11 MACRO instructions vary according to the instruction type, and number and complexity (addressing mode) of operands. The debugger ignores the current length mode when it enters instructions; instead the current address is incremented according to the number of bytes required by the instruction.

An instruction string entry must be enclosed with quotation marks or apostrophes.

```
DBG>DEPOSIT/INSTRUCTION INCRS = 'ADDL3 #5,R3,R4'
```

The debugger interprets numeric values in the current radix mode.

When entering an instruction, you must verify that the length of the data string can be accommodated by the number of bytes you intend to overwrite. While you cannot deposit more than there is space for, you can use the NOP instruction to fill bytes that are unoccupied after you complete the deposit of an instruction or instructions.

You must also enter a B^, W^, or L^ when you specify a value offset from a register. For example:

```
B^4(R5)
```

Leading zeros must be specified for hexadecimal constants that begin with alphabetic characters, to differentiate them from symbols. For example:

```
B^0F(R5)
```

Symbols can be included in instructions being deposited. However, symbolic expressions must not contain the backslash character.

10.2.3 Depositing ASCII Data

ASCII character input is by quoted string. You must enclose each string with quotation marks or apostrophes. This provision lets you include literal quotation marks within a string. For example,

```
DBG>DEPOSIT/ASCII  WINK  = 'ZZZZ'
DBG>DEPOSIT/ASCII THINK  = "IT'S"
DBG>DEPOSIT/ASCII PLINK  = '"1"'
```

The ending delimiter must match the beginning delimiter.

The current length mode has no effect on the string being deposited. The string is deposited as specified, with no truncation or padding.

CHAPTER 11

USING THE EVALUATE COMMAND

The EVALUATE command lets you use the debugger as a calculator, expression analyzer, radix converter, bit field examiner, and literal verifier.

11.1 USING EVALUATE

EVALUATE interprets an input expression in terms of the current modes, reduces the expression to a value, and displays the value in the current modes. The command format is:

```
DBG>EVALUATE[/mode][...] expression[,...]
```

The evaluations of multiple input expressions are displayed in a list, which is ordered to match the input order.

11.2 EXPRESSION EVALUATION

EVALUATE performs integer arithmetic with all operations performed according to the current length mode (that is, BYTE, WORD, or LONG) with arguments and results limited to the corresponding value ranges. The debugger truncates values that exceed the current length mode by discarding most-significant-bit positions, and prints a message.

EVALUATE analyzes an expression in the context of the current language. The rules of precedence applicable to VAX-11 MACRO are described in Section 4.1.

11.3 EVALUATING BIT FIELDS

You can use EVALUATE to display the current contents of specified bits in a location. The syntax is:

```
DBG>EVALUATE value <high bit:low bit>
```

You specify the bounds of a bit field by decimal integers, regardless of the current radix mode. Bit positions are from 0 (least significant) through 31 (most significant). The debugger extracts the contents of the bit positions, right justifies them in a longword, and reports the contents in the current radix mode. The current length mode is ignored.

The following method is recommended for evaluating bit fields of a location.

```
DBG>EXAMINE address-expression
address:  contents
DBG>EVALUATE \ <high bit:low bit>
bit-field value
```

The EXAMINE command establishes the location's contents as the value represented by the backslash (\), which is the "last value displayed" symbol. This sequence is necessary because EVALUATE simply reduces an input expression to a value, but EXAMINE reduces an expression to an address and displays the contents of that address.

Examples:

```
DBG>EXAMINE LOOP3
WATCH\LOOP3:  0FFFF8FD0
DBG>EVALUATE \ <6:4>
00000005
```

To display other bit patterns of the same location, you can make use of the fact that the "current location" symbol retains the address that you last examined. For example:

```
DBG>EXAMINE .
WATCH\LOOP3:  0FFFF8FD0
DBG>EVALUATE \ <8:6>
00000007
```

11.4 EVALUATING VAX-11 MACRO LITERALS

When SYMBOLIC mode is in effect, the debugger does not translate literal values into their symbolic equivalents for purposes of displaying these values. Thus, a displayed instruction may not appear exactly as you entered it in the source code. For example, the instruction

```
MOVL  #6,OFFSET(FP)
```

would be displayed as

```
MOVL  #6,W^0FFDC(FP)
```

where OFFSET represents the literal -24.

The EVALUATE command can help you verify that instructions are the same. If you type

```
DBG>EVALUATE/LITERAL expression
```

The debugger displays every literal pathname that has the value of the expression as its literal assignment. It is then a simple matter to scan the pathname list for the literal symbol name you wish to verify.

CHAPTER 12

EXCEPTION CONDITIONS

Exception conditions are conditions that interrupt execution of your program. In the context of the debugger, an exception condition is either forced by the debugger, or external to the debugger. Forced exception conditions include: the occurrence of a breakpoint, tracepoint, or watchpoint; or the completion of a requested program step or debugger command.

This chapter describes the debugger's response to both forced exception conditions and external exceptions. It does not describe the cause and effect of external exception conditions, nor does it describe how to write handler routines for them. Refer to the VAX/VMS System Services Reference Manual, the VAX-11 MACRO User's Guide, and the VAX-11/780 Architecture Handbook for appropriate information.

12.1 PROCESSING EXCEPTION CONDITIONS

Exception conditions are processed in the following manner. An exception condition interrupts your program and causes VAX/VMS to pass control to the debugger. The debugger must first determine if the exception was forced. If it was, the debugger reports the condition by printing the appropriate message. For example:

```
Breakpoint exception: [routine] break at pc = LOCATION

Tracepoint exception: [routine] trace at pc = LOCATION:INSTRUCTION

WATCHPOINT EXCEPTION: WRITE TO LOCATION at pc = LOCATION
                            old value = value
                            new value = value

Step exception:       [routine] stepped to pc = LOCATION
```

If the debugger determines that the exception condition is external, it returns control to VAX/VMS unless you previously specified SET EXCEPTION BREAK (described in Section 12.2). This causes the debugger to react as if you had specified a breakpoint at the exception location. Generally, you will have to exit from the debugger when an exception break occurs.

If you did not specify this option, VAX/VMS gets control. What happens next depends on whether you provided a condition handler for the exception condition. If VAX/VMS finds such a handler, it allows the handler to decide the future of your program. If a handler is not found, or if all handlers resignaled the condition, the debugger again acquires control, reports the type of exception condition, and waits for your command.

12.2 BREAK ON EXTERNAL EXCEPTION CONDITION

Rather than have the debugger return control to VAX/VMS for an external exception condition, you can request that the debugger treat all such exceptions as breakpoints. The command is

```
DBG>SET EXCEPTION BREAK
```

The debugger reports the occurrence of exception conditions by printing the error message for the exception, and then printing the following:

```
exception break at pc = LOCATION
```

Where LOCATION indicates where the error occurred in your program. The debugger then prints its prompt message.

To cancel this option, enter the command

```
DBG>CANCEL EXCEPTION BREAK
```

CHAPTER 13

CALLING ROUTINES AND SHOWING CALLS

The debugger's CALL command lets you call procedures or subroutines in your program directly from command level. This chapter tells you how to use the CALL command, and how to use the SHOW CALLS command to report all currently active call frames for your program.

13.1 CALLING ROUTINES

The debugger's CALL command executes a call directly to any routine in your program's address space, whether or not your program actually includes a call to that routine.

The command format is:

```
DBG>CALL name [(argument-list)]
```

where name is the routine's symbolic name or its virtual address. Arguments in the optional argument list must be separated by commas; these arguments are actual arguments to be passed to the called routine. The debugger assumes that the called routine conforms to the VAX-11 procedure calling standard (refer to the VAX-11/780 Architecture Handbook for details).

You can thus easily access any routine in your program for debugging purposes. You can also debug unrelated routines by linking them with a dummy main module. The dummy module need only provide a transfer address for the image. You need not be concerned with coding call statements and argument lists. You can express them with the CALL command.

The debugger creates a complete set of pseudo-register locations for interim use by the called routine. When control returns from the called routine to the point at which it was called, the debugger discards the interim registers, restores the previous register context, and displays the value returned by the called routine.

13.2 SHOWING ACTIVE CALLS

The SHOW CALLS command reports various information concerning the current level of nested procedure calls. The command format is:

```
DBG>SHOW CALLS [decimal-integer]
```

where you have the option of requesting that all call levels be reported (the default) or requesting that the debugger report on a specified number of call levels. The call count can be any decimal

integer in the range 0 through 32767. If the call count exceeds the number of calls currently active, it is ignored. If you specify 0, the command is accepted, but no output results.

Normally, the debugger responds with the following report:

```
MODULE NAME     ROUTINE NAME     LINE     ABSOLUTE PC     RELATIVE PC
```

The first line in the report refers to the current call level. The remaining lines report all (or the requested number) of call levels in the order of most recent call through first call. For VAX-11 MACRO, the report presents the following information.

MODULE NAME — Reports the module in which the call occurred. If the debugger's symbol table does not include symbol information for the module in which the call occurred, the module name remains blank and the debugger reports the routine name by the appropriate global symbol.

ROUTINE NAME — Reports the routine or program section name in which the call occurred.

LINE — Left blank for VAX-11 MACRO (that is, it has no meaning). Used only for line-oriented (statement) languages (such as FORTRAN) to identify the line number of the call.

RELATIVE PC — Reports the address of the call relative to the symbol expressed under ROUTINE NAME. The debugger displays the relative address in hexadecimal, regardless of the current radix mode.

ABSOLUTE PC — Reports the virtual address of the call in hexadecimal, regardless of the current radix mode.

If there are no active call frames, the debugger responds to SHOW CALLS with an error message. This indicates that the stack has been corrupted, or that the user program has terminated.

CHAPTER 14

PROCESSOR STATUS LONGWORD (PSL)

This chapter describes how to display the current contents of the Processor Status Longword (PSL), and how to alter its contents to support your program debugging. For a detailed description of the PSL, see the VAX-11/780 Architecture Handbook.

14.1 DISPLAYING THE PROCESSOR STATUS LONGWORD

To display the current contents of the Processor Status Longword (PSL), type:

```
DBG>EXAMINE/SYMBOLIC PSL
```

The debugger responds with:

```
PSL:  CMP TP FPD IS CURMOD PREMOD IPL DV FU IV T N Z V C
        n  n   n  n  mode   mode   lv  n  n  n n n n n n
```

where "n" is 0 or 1. The interrupt priority level, lv, is displayed as a hexadecimal value, 0 through 1F. Mode is expressed as: KERN, EXEC, SUPR, or USER.

You can display the current contents of the PSL as a hexadecimal value by specifying:

```
DBG>EXAMINE/NOSYMBOLIC/HEXADECIMAL PSL
```

14.2 ALTERING THE PROCESSOR STATUS LONGWORD

You can alter the PSL's low-order word, which is the processor status word (PSW), regardless of the privileges allocated to your account. However, you cannot alter the following conditions, regardless of the privileges allocated to your account.

- CMP - compatibility mode
- IS - interrupt stack
- CURMOD - current mode
- PREMOD - previous mode

You can compute the value to be entered in the PSL by

```
DBG>EVALUATE/HEXADECIMAL expression
```

where "expression" is the sum of key numbers selected from Table 14-1 according to the conditions that must be maintained (that is, reentered as they were displayed) and the conditions that you wish to change.

To replace the current PSL contents, type:

```
DBG>DEPOSIT/HEXADECIMAL PSL = value
```

Table 14-1
PSL Alteration Values

Bit	Key	Key Number (Hex)	Description
31	CMP	80000000	Compatibility mode
30	TP	40000000	Trace Pending
29 28	MBZ	0	(Must Be Zero)
27	FPD		First Part Done
26	IS		Interrupt Stack
25 24	CUR- MOD	2000000 1000000	Current mode: 3000000=user, 2000000=supr, 1000000=exec, 0=kern
23 22	PRE- MOD	800000 400000	Previous mode: 0C00000=user 800000=supr, 400000=exec, 0=kern
21	MBZ	0	(Must Be Zero)
20	IPL	100000	Interrupt priority level: 0 - 1F. Enter the displayed or desired priority level in hexadecimal and append 0000 to the value.
19		80000	
18		40000	Remember to precede the leftmost character, if an alphabetic, with a zero.
17		20000	
16		10000-+	EXAMPLE: 0C0000 for level 12.
15		0	(must be zero)
14		0	(must be zero)
13		0	(must be zero)
12		0	(must be zero)
11		0	(must be zero)
10		0	(must be zero)
9		0	(must be zero)
8		0	(must be zero)
7		80	Decimal overflow trap enable

(continued on next page)

Table 14-1 (Cont.)
PSL Alteration Values

Bit	Key	Key Number (Hex)	Description
6		40	Floating underflow trap enable
5		20	Integer overflow trap enable
4		10	Trace trap enable
3		8	Negative condition code
2		4	Zero condition code
1		2	Overflow condition code
0		1	Carry condition code

CHAPTER 15

DEBUGGER MESSAGES

The debugger provides four classes of messages:

- Informational - informational messages are provided to let you know the status of the debugger or your program. While not, strictly speaking, error messages, they may indicate erroneous command input. For example, the message

  %DEBUG-I-NOSUCHBPT, no such breakpoint

  means you have specified a breakpoint incorrectly. Informational messages are prefixed:

  %DEBUG-I-

- Warning - warning messages are displayed for the least severe level of errors detected. Your debugging session continues unaffected after a warning message, providing you a chance to respecify the erroneous command or operand. Warning messages are prefixed:

  %DEBUG-W-

- Error - error messages indicate that the debugger has detected a condition that prevents it from continuing the session. You should submit a Software Performance Report (SPR) if you receive a severe error message from the debugger. Severe errors are prefixed:

  %DEBUG-E-

- Fatal - fatal errors are errors that directly affect the debugger. Following a fatal error, the debugger prints the appropriate message and returns control to the VAX/VMS Command Interpreter.

  You should submit an SPR if you receive a fatal error message from the debugger. Fatal error messages are prefixed:

  %DEBUG-F-

Each message is listed in the following subsections, alphabetically within subsection.

15.1 INFORMATIONAL MESSAGES (PREFIX:%DEBUG-I-)

DBGBUG, DEBUG coding error, please report no. 'number'

If you receive this message, please submit an SPR stating the conditions that existed when the message appeared, including the number specified in the message.

EXITSTATUS, is 'xxx'

This message is displayed to indicate that the user image has exited, with the status specified by xxx. The string xxx is produced by the system's error message facility.

INITIAL, language is 'aaa', scope and module set to 'name'

You usually receive this message when you initiate the debugger, to inform you of the debugger's settings for language, scope, and initial symbol table contents (for local symbols).

LONDST, too many modules - some ignored

This message indicates that when the debugger was initialized, it found more modules in the image than it could accommodate; to determine which modules were included, use the SHOW MODULE command. If crucial modules were omitted, relink the image, specifying those modules before modules not needed for debugging purposes.

MODNOTADD, no space to add module 'name'

A SET MODULE command has failed because of insufficient symbol table space. To make room, use CANCEL MODULE to remove modules with symbols that are no longer needed, then retry the SET MODULE command. You can use SHOW MODULE to see how much space is needed and available.

NOBREAKS, no breakpoints are set

This is the response to SHOW BREAK when no breakpoints are set.

NOGLOBALS, some or all global symbols not accessible

This message indicates an error or overflow in the global symbol table of the image. Reduce the number of global symbols.

NOLOCALS, image does not contain local symbols

This message indicates that when the debugger was initiated there were no local symbols to be put into its symbol table. Recompile or reassemble, specifying DEBUG or TRACEBACK, and then relink the image.

NOSUCHBPT, no such breakpoint

This is the response to CANCEL BREAK address-expression when no breakpoint is set at the specified address.

NOSUCHTPT, no such tracepoint

This is the response to CANCEL TRACE address-expression when no tracepoint is set at the specified address.

NOSUCHWPT, no such watchpoint

This is the response to CANCEL WATCH address-expression when no watchpoint is set at the specified address.

NOTALLSYM, cannot initialize symbols for default modules

The debugger could not put symbol information into the symbol table for the first module in the image (the default module). Use SET MODULE to initialize the symbol table.

NOTRACES, no tracepoints are set, no opcode tracing

This is the response to SHOW TRACE when no tracepoints are set, and no opcode tracing is in effect.

NOWATCHES, no watchpoints are set

This is the response to SHOW WATCH when no watchpoints are set.

NUMTRUNC, number truncated

The debugger truncated a numeric entry that exceeded the current length mode, or could not be accommodated in the specified context.

STEPINTO, cannot step over PC = 'xxx'

The debugger was forced to ignore the OVER mode when it reached the location indicated. An INTO step will be performed.

STGTRUNC, string truncated

The debugger truncated an ASCII string entry that exceeded the current length mode, or was otherwise inappropriate for the context in which it was specified.

15.2 WARNING MESSAGES (PREFIX:%DEBUG-W-)

BADOPCODE, opcode 'xxx' is unknown

You specified an unrecognized opcode to the debugger.

BADSCP, scope must end with module or routine

You have specified the SET SCOPE command, but the symbol type of the pathname is not module or routine. A SCOPE entry must end with a module name (VAX-11 MACRO) or routine name (VAX-11 FORTRAN IV-PLUS).

BADSTARTPC, cannot access start PC = 'xxx'

The PC indicated is not a readable address, therefore it can not be executed. Specify a valid start PC location.

BADSTEP, cannot decode instruction at address 'xxx'

A STEP command reached an instruction that is not recognized by the debugger. Check to be sure that your image has not been overwritten.

BADWATCH, cannot watch protected address 'xxx'

You have requested a watchpoint for a protected location.

BITRANGE, bit range out of limits

You have specified a bit range in an EVALUATE command that exceeds the range of bits that can be evaluated. The valid range is <31:0>, decimal. Only numeric characters are valid in a bit range.

BRTOOFAR, destination 'xxx' is too far for branch operand

You deposited a branch instruction that contained an unreachable target location.

DIVBYZERO, attempted to divide by zero

An expression can not be evaluated because it contains an attempt to use a divisor equal to zero.

ENDWITHGO, cannot imbed GO, STEP, or CALL in command sequence

A command sequence contained a GO, STEP, or CALL that was not the last command in the sequence. Commands up to, but not including, the GO, STEP, or CALL are executed. The rest of the sequence is ignored.

EXARANGE, invalid range of addresses

You specified the address range in the wrong order. The correct order is: low bound:high bound.

EXPSTKOVR, expression exceeds maximum nesting level

An expression containing more than 20 nesting levels was encountered.

INTEGER, this operation only valid on integers

You attempted to perform a computation that accepts only integer values.

INVARRDSC, invalid array descriptor

The debugger detected an invalid array descriptor. If this message occurs for any reason other than an incorrect specification in a command you entered to the debugger, please submit an SPR.

INVCHAR, invalid character

A character you entered in a command is not acceptable in the current context.

INVDIM, subscript error, was declared DIMENSION 'xx'

A subscript was specified incorrectly, according to the DIMENSION statement in the FORTRAN program.

INVNUMBER, invalid numeric string 'nn'

The number specified as 'nn' is invalid in the current context.

INVOPR, unrecognized operator in expression

An expression contained a character that the debugger did not recognize, in place of a valid operator.

INVPATH, improperly terminated pathname beginning with 'xxx'

An improperly-formatted pathname has been encountered: "symbol" followed by \ must begin a pathname. The characters following \ do not constitute a valid symbol.

LASTCHANCE, stack exception handlers lost, re-initializing stack

A user-program error has caused the VAX/VMS condition handling mechanism to fail. The probable cause is an overwritten stack.

MAXDIMSN, maximum number of subscripts is 'nn'

This is a FORTRAN-only message produced when an array reference is used in a debugger expression. The array was declared to have the indicated number of dimensions (nn), but the reference was made with either too few or too many subscripts.

MULTOPR, multiple successive operators in expression

You entered an expression that contains two or more operator characters in succession.

NEEDMORE, unexpected end of command line

The command line was terminated before it contained a complete command. It was valid to the point of termination.

NOACCESSR, no read access to virtual address 'loc'

The debugger does not have read access privileges to the address specified as 'loc'. The value of 'loc' is always hexadecimal.

NOACCESSW, no write access to virtual address 'loc'

The debugger does not have write access privileges to the address specified as 'loc'.

NOANGLE, unmatched angle brackets in expression

An expression you entered contains a left angle bracket that has no matching right angle bracket.

NOBRANCH, instruction requires branch-type operand

A branch-type instruction was given which did not contain a valid operand for the destination field. For example, you cannot DEPosit/Instruction addr='BNEQ R0'.

NOCALLS, no active call frames

Response to a SHOW CALLS command when the debugger locates no active call frames. Your image may have exited, or the stack may have been corrupted.

NODECODE, cannot decode instruction

This message is produced when you specify an EXAMINE command in instruction mode, and the indicated byte sequence is not a valid VAX-11 instruction.

NODELIMTR, missing or invalid instruction operand delimiter

An instruction has been given that contains a syntax error at the point where one operand has been terminated and another is supposed to begin. For example, MOVL R0 R1 (you forgot the ',' as in R0,R1).

NOEND, string beginning with 'xxx' is missing end delimiter x

An ASCII or INSTRUCTION string must begin and end with either apostrophes or quotes. If the ending delimiter is not encountered before the string ends, this message is produced. The message gives the ending delimiter the debugger expected to find (shown as x, above), and the first 10 characters of the string you entered.

NOINSTRAN, cannot translate opcode at location 'loc'

The contents of the location indicated as 'loc' are not a recognizable opcode. The value of 'loc' is always hexadecimal.

NOLABEL, routine 'name' has no %label 'label'

You attempted to refer to a label that does not exist in the indicated routine.

NOLINE, routine 'name' has no %line 'line'

The indicated line number does not exist in the subroutine specified as 'name'. Consult the compiler listing. This message is also produced when the indicated line number exists, but does not correspond to executable code. An example of this is the line number of a FORMAT statement.

NOLITERAL, no literal translation exists for 'xxx'

The value indicated as 'xxx' has not been assigned to a symbolic equivalent of type literal (absolute).

NOOPRND, missing operand in expression

One or more operands have been omitted from an expression.

NOSUCHLAB, no scope exists to look up %label 'label'

You referenced the indicated label without an implicit or explicit associated pathname. If you specifically indicated that the scope was to be ignored, or scope was <null>, and a PC-implied scope cannot be derived, this message is produced.

NOSUCHLAN, language 'name' is unknown

The debugger does not recognize the language specified.

NOSUCHLIN, no scope exists to look up %line 'line'

You referenced the indicated line without an associated pathname. If you specifically indicated that the scope was to be ignored, or scope was <null>, and a PC-implied scope cannot be derived, this message is produced.

NOSUCHMODU, module 'name' does not exist

The specified module is not part of the image.

NOSYMBOL, symbol 'name' does not exist

The specified symbol cannot be located in the debugger's symbol table.

NOTDONE, 'xxx' not yet a supported feature

You attempted to use a debugger feature that is not yet implemented. The message indicates which feature was requested.

NOTIMPLAN, 'xxx' is not implemented at command level

You tried to SET the indicated LANGUAGE, which the debugger knows about, but does not yet implement as a fully-supported language.

OPSYNTAX, instruction operand syntax error

You have specified invalid syntax in an operand within an instruction. For example, MOVL (R0],Rl.

PARSEERR, internal parsing error

If you receive this message, please submit an SPR.

PARSTKOVR, parse stack overflow, simplify expression

The expression you entered contains too many levels of angle brackets <...>. Reenter the expression, reducing the number of angle bracket levels. If this message recurs frequently, submit an SPR.

PATHTLONG, too many qualifiers on name

You entered a pathname that comprised more than 15 elements.

REDEFREG, register name already defined

You attempted to use a register name as a symbol to be defined in the DEFINE command.

RESOPCODE, opcode 'xxx' is reserved

The operand you specified is reserved for DIGITAL's use only.

SUBSTRING, invalid substring (a:b), was declared CHARACTER* NN

You specified a substring that is not entirely within a character string declared in a FORTRAN CHARACTER declaration.

SYNTAX, command syntax error at or near 'xxx'

Your command contains incorrect syntax at a point in the line indicated by 'xxx'.

15.3 ERROR MESSAGES (PREFIX:%DEBUG-E-)

The following error messages indicate that the debugger is unable to continue execution of your program. The image exits, and control returns to the VAX/VMS Command Interpreter.

If you receive any of these messages, please submit an SPR.

DBGERR, internal DEBUG coding error

DEBUGBUG, internal DEBUG coding error; please report no. 'number'

FRERANGE, storage package range error

FRESIZE, storage package size error

INVDSTREC, invalid DST record

NOFREE, no free storage available

NORSTBLD, cannot build symbol table

RSTERR, error in symbol table

15.4 FATAL ERROR MESSAGES (PREFIX:%DEBUG-F-)

The following messages indicate errors fatal to execution of the debugger. Control returns to the VAX/VMS Command Interpreter. If you receive any of these messages, please submit an SPR.

NOWBPT, cannot insert breakpoint

NOWOPCO, cannot replace breakpoint with opcode

NOWPROT, cannot set protection

APPENDIX A

COMMAND SUMMARY

This appendix summarizes the commands that can be used in debugging VAX-11 MACRO programs. Refer to the appropriate language user's guide for information regarding the use of the debugger for programs written in other languages.

The summary presents the commands in alphabetical order.

Brackets ([...]), where shown, enclose optional command elements; they are not part of the syntax.

See SET MODE for entry/display mode keywords.

With the exception of ASCII character input, the debugger automatically converts lowercase input to uppercase (that is, the debugger is not sensitive to the case of an input character).

"Address-expression" in the command syntax representations can be the pathname (see SET SCOPE) of a local or global symbol in your program, a numeric value, a symbol that you defined during this debugging session, a debugger special character, or an expression that combines any of these elements.

The term "program" means an executable image (refer to the VAX-11 Linker Reference Manual for additional information).

In VAX-11 MACRO, the radix indicators for numeric address or data entries are: ^X (for hexadecimal), ^D (for decimal), and ^O (for octal).

The debugger supports command line continuation. A command line can contain up to approximately 500 characters, including nonprinting characters. You indicate continuation with the hyphen (-) as the last character prior to the carriage return. The debugger indicates a continued line by displaying an underline character as the first character on the line rather than the DBG> prompt.

CTRL/"x" refers to the simultaneous typing of the CTRL key and the respective character key, that is, C, Y, or Z (refer to the VAX/VMS Command Language User's Guide for information on the complete list of CTRL functions). CTRL/"x" echoes at the terminal as ^x.

With the exception of the CTRL functions, you must end all command lines with a carriage return.

>CALL name [(argument,...)]

Call routine by its symbolic name or by its virtual address (address expression is not valid) with optional argument list. An argument list must be enclosed by parentheses.

>CANCEL ALL

Cancel all breakpoints, tracepoints, watchpoints, and user-set entry/display modes. Restore initial entry/display modes. This command does not change the current contents of the debugger's symbol table (that is, those symbols acquired from program modules at debugger initialization or through use of the SET MODULE command, or any symbols that you defined during this debugging session). The current language is not changed.

>CANCEL BREAK address-expression

>CANCEL BREAK/ALL

Cancel breakpoint set at specified address, or cancel all breakpoints.

>CANCEL EXCEPTION BREAK

Cancel the request that your program stop, as at a breakpoint, for any exception condition.

>CANCEL MODE

Restore initial entry/display modes. Command does not change SCOPE or current language.

>CANCEL MODULE module-name-list

>CANCEL MODULE/ALL

Purge symbolic information associated with the named modules from the debugger's symbol table, or purge all module related information from the symbol table. The typical use is to make space available for local symbols associated with another module or modules (see SET MODULE). Global symbols and any symbols defined during this debugging session are not affected.

>CANCEL SCOPE

Enter null contents in SCOPE (that is, delete the previously set scope).

>CANCEL TRACE address-expression

>CANCEL TRACE/CALL

>CANCEL TRACE/BRANCH

>CANCEL TRACE/ALL

Cancel tracepoint set at specified address, cancel all opcode tracing at call-type instructions, cancel all opcode tracing at branch-type instructions, or cancel all tracepoints and opcode tracing.

>CANCEL WATCH address-expression

>CANCEL WATCH/ALL

Cancel watchpoint set at specified address, or cancel all watchpoints.

CTRL/C

Has same effect, and echoes at terminal, as CTRL/Y (see below) if your program does not include an exception condition handler for CTRL/C.

CTRL/Y

Interrupt the debugger or executing program and transfer control to the VAX/VMS command interpreter (signaled by the system prompt $). Type

```
DEBUG
```

after the system prompt to return control to the debugger. Type

```
CONTINUE
```

after the system prompt to return control to the interrupted program. Typing any VAX/VMS command other than DEBUG or CONTINUE will probably force the premature exit of your program. You can use CTRL/Y to interrupt a looping program. To determine the point at which you interrupted your program, type

```
DBG>EXAMINE/INSTRUCTION @PC
```

CTRL/Z

Same result as EXIT; that is, terminate the debugging session and transfer control to the VAX/VMS command interpreter.

>DEFINE symbol-name=value[,symbol-name=value...]

Equate name(s) of name=value list with associated value(s) for use during this debugging session. The debugger searches these symbols first whenever it requires a definition for a symbolic entry, and whenever it requires a symbolic name to report a location.

>DEPOSIT[/mode[...]] address-expression=data[,data,...]

Enter data specified in data list in sequence of locations beginning with the specified address.

>EVALUATE[/mode[...]] expression[,...]

transform input (which can be arithmetic expression, ASCII string, VAX-11 MACRO instruction, symbol, or numeric value) to associated value(s) and display result(s). Can be used as desk calculator, radix converter, symbol verifier, etc. The debugger displays result(s) in the order in which you specified the input.

>EXAMINE[/mode[...]] address [:address][,address[:address]...]

Display current contents of specified address(es). The colon signifies range; that is, display contents of addresses from low address through high address.

>EXIT

Terminate debugging session and transfer control to the VAX/VMS command interpreter.

>GO[address-expression]

Start or continue program execution. First GO command without an address starts the program at its transfer address. GO commands thereafter continue execution from a stopped point (as at a breakpoint or watchpoint, or because of an exception condition).

An address entry replaces the current program counter (PC) contents; execution starts or continues from the new location.

Once you have started a program, you should not to attempt a restart at the transfer address or any other address. Program behavior is unpredictable when restarted.

>SET BREAK address-expression [DO (command list)]

Establish breakpoint at specified address (the breakpoint stops your program before the instruction beginning with "address-expression" is executed).

The debugger executes commands in DO sequence command format whenever your program stops because of the specified breakpoint. Parentheses are required as command list delimiters. Multiple commands must be separated by semicolons. Any complete debugger command can be used in this context, including GO, STEP, or CALL. If GO, STEP, or CALL is specified, it must be the last command in the sequence.

You can specify the "after" option to defer a breakpoint.

SET BREAK/AFTER:decimal-integer address-expression

Your program does not stop because of the breakpoint (that is, the breakpoint is ignored) until the "n"th pass through the specified location, as in an interation, where "n" is within the range 1 through 32767. Thereafter, the breakpoint takes effect each time the debugger encounters it.

You can specify a temporary (or one time) breakpoint by:

SET BREAK/AFTER:0 address-expression

The debugger automatically cancels the breakpoint after it stops your program the first time the breakpoint is encountered.

>SET EXCEPTION BREAK

Stop the program and report the current program counter contents if an exception condition occurs that was not initiated by a debugger command.

>SET LANGUAGE language-name

Let the debugger interpret input and display output in the syntax of the specified language. The debugger rejects commands that are not valid in the specified syntax. The debugger initially recognizes the language of the first module in your program that contains symbol information.

>SET MODE mode-keyword[,mode-keyword...]

Allow or inhibit the entry and display of data in specified formats.

The following list describes the function of each keyword (refer to SET SCOPE for additional information regarding the use of the symbol search control keywords, [NO]GLOBAL and [NO]SCOPE):

ASCII	Interpret/display data as ASCII characters.
BYTE	Interpret/display data in byte lengths.
DECIMAL	Interpret/display data in decimal radix.
GLOBAL	Use symbolic entry as first pathname in search.
HEXADECIMAL	Interpret/display data in hexadecimal radix.
INSTRUCTION	Interpret/display VAX-11 MACRO instructions.
LONG	Interpret/display data in longword lengths.
NOASCII	Inhibit entry/display of ASCII characters.
NOGLOBAL	Use symbolic entry as last pathname in search.
NOINSTRUCTION	Inhibit entry/display of VAX-11 MACRO instructions.
NOSCOPE	Inhibit SCOPE's contribution to pathname.
NOSYMBOLIC	Inhibit display of symbolic addresses.
OCTAL	Interpret/display data in octal radix.
SCOPE	Add SCOPE's contents to entry to form pathname.
SYMBOLIC	Display symbolic addresses.
WORD	Interpret/display data in word lengths.

The debugger's initial modes are: SYMBOLIC, NOINSTRUCTION, NOASCII, NOGLOBAL, HEXADECIMAL, and LONG. SCOPE is initialized to contain the name of the first module in your program.

You can also enter the mode keywords with the commands DEPOSIT, EVALUATE, and EXAMINE to override the current associated mode (radix, data length, symbol search, or type). A slash must precede each mode keyword entered after these command verbs.

```
command-verb/keyword/keyword ...
```

>SET MODULE module-name-list

>SET MODULE/ALL

Enter local symbols and program section names associated with the program-module list in the debugger's symbol table, or enter information from all modules in the symbol table. The debugger cannot interpret local symbols unless their associated module names appear in the status report produced by the SHOW MODULE command with a "yes" indication.

>SET SCOPE module-name

Retain module-name entry as SCOPE's contribution to creation of a pathname under control of the debugger's symbol search rules. A pathname completely and unambiguously identifies a symbol and points to that symbol's definition (that is, its translation value). For VAX-11 MACRO, a pathname is symbol-name or module-name\symbol-name.

The debugger evaluates an expression in which a symbolic entry appears only if a definition was located for the entry. If it fails to locate a match for a pathname, the debugger reports the search failure and the symbol name.

>SET STEP keyword [,keyword ...]

Establish default conditions for the STEP command. Valid keywords are:

INSTRUCTION - step increment in VAX-11 MACRO instruction.

LINE - step increment is line (for line-oriented languages).

INTO - allow stepping through called routine.

OVER - step over called routine (make call transparent).

SYSTEM - allow stepping into system space.

NOSYSTEM - inhibit stepping into system space.

The initialization conditions for VAX-11 MACRO are: INSTRUCTION, NOSYSTEM, and OVER.

>SET TRACE address-expression

>SET TRACE/CALL

>SET TRACE/BRANCH

Set tracepoint at specified address, or specify tracing of all call-type instructions, or all branch-type instructions. At a tracepoint, the debugger reports the current program counter contents and then continues program execution automatically.

>SET WATCH address-expression

Report if the contents of the specified location(s) are modified.

The locations watched can be individual addresses (including the number of bytes specified by the length mode in effect when the watchpoint was set), or an entire program section (specified by name).

The debugger stops the program (as at a breakpoint) and reports both the previous contents and the current contents of the location.

>SHOW BREAK

Report the locations of current breakpoints and any relevant information associated with them, such as DO command sequences and "after" options.

>SHOW CALLS [n]

Report current call level and the hierarchy of call levels that preceded it (that is, trace your program's call history). If "n" (a decimal integer) is expressed, the debugger reports "n" call levels back from the current level ("n" has the range 0 through 32767). If "n" is omitted, all preceding call levels are reported.

>SHOW MODE

Report the current entry/display modes (see SET MODE).

>SHOW MODULE

List program modules by name, indicate whether or not their associated local symbol data exists in the debugger's symbol table (by yes or no), and indicate the approximate space required for the entry of each module's symbol data. List also the amount of space currently unused. The debugger has no knowledge of any program module not reported in this status report.

>SHOW SCOPE

Report the current contents of SCOPE. A null string (<null>) indicates that SCOPE makes no effective contribution to the creation of a pathname.

>SHOW STEP

Report current default conditions for STEP (see SET STEP).

>SHOW TRACE

Report the locations of current tracepoints, or that opcode tracing is in effect.

>SHOW WATCH

Report the locations of current watchpoints and the number of bytes monitored by each watchpoint.

>STEP[/keyword] [decimal-integer]

Stop the program after executing the next instruction only (the default condition if you do not specify an instruction count), or after executing the next "n" instructions, where "n" is a decimal integer from 2 through 32767.

The following keywords can either be used after the STEP command verb (STEP/keyword) or be set with the SET STEP command to establish the default conditions for STEP. The SHOW STEP command displays the current defaults.

The keywords have the following relationships:

SYSTEM/NOSYSTEM - allow or inhibit steps into system space.

INTO/OVER - step into or over a called routine.

LINE/INSTRUCTION - step by lines or by instructions.

The initialized defaults for VAX-11 MACRO are:

INSTRUCTION, NOSYSTEM, OVER.

A

B

C

D

E

G

H

I

K

L

M

N

O

P

Q

R

S

T

V

W

VAX-11
Symbolic Debugger
Reference Manual
AA-D026A-TE

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please indicate the type of reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)_____

Name_____ Date_____

Organization_____

Street_____

City_____ State_____ Zip Code_____
or
Country

Please fold along this line.

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Documentation
146 Main Street ML5-5/E39
Maynard, Massachusetts 01754

digital

PRINTED IN USA