digital

VAX–11
Symbolic Debugger
Reference Manual

Order No. AA-D026B-TE

VAX11

March 1980

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-D026B-TE

SUPERSESSION/UPDATE INFORMATION: This revised document supersedes the VAX-11 Symbolic Debugger Reference Manual (Order No. AA-D026A-TE)

OPERATING SYSTEM AND VERSION: VAX/VMS V02

SOFTWARE VERSION: VAX/VMS V02

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
Revised, March 1980

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	PDT
DATATRIEVE	TRAX	

CONTENTS

CONTENTS

CONTENTS

CONTENTS

TABLES

PREFACE

MANUAL OBJECTIVES

This manual describes the facilities of the VAX-11 Symbolic Debugger. This manual is primarily an aid to debugging programs written in VAX-11 MACRO assembly language. For information on debugging programs written in other languages, refer to the appropriate language user's guide before reading this manual.

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. This manual is also intended for programmers using other languages who need more information than is available in the appropriate language user's guide.

STRUCTURE OF THIS DOCUMENT

This manual comprises 6 chapters and 3 appendixes. Chapter 1 provides a functional overview of the VAX-11 Symbolic Debugger's concepts and facilities. Chapters 2 through 5 describe how to use the debugger. Chapter 6 describes the debugger commands. Appendix A contains a list of error messages. Appendix B describes features that are included for compatibility with previous versions of the debugger. Appendix C provides a brief summary of commands and lists the minimum abbreviation of each.

ASSOCIATED DOCUMENTS

To obtain supplemental information, the following documents are recommended:

- VAX-11 Architecture Handbook
- VAX/VMS Primer
- VAX-11 MACRO Language Reference Manual

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

CONVENTIONS USED IN THIS DOCUMENT

This document uses the following conventions.

Convention	Meaning
Uppercase words and letters	Uppercase words and letters, used in examples, indicate that you should type the word or letter exactly as shown.
Lowercase words and letters	Lowercase words and letters, used in format examples, indicate that you are to substitute a word or value of your choice.
Quotation marks Apostrophes	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe (') is used to refer to a single quotation mark.
[ ]	Square brackets indicate that the enclosed item is optional (except when used in file specifications where square brackets delimit directory names).
{ }	Braces are used to enclose lists from which one element is to be chosen.
...	A horizontal ellipsis indicates that the preceding item(s) can be repeated one or more times.
. . .	A vertical ellipsis indicates that not all of the statements in an example or figure are shown.
DBG> EVALUATE X 00000002	In examples of commands you enter and system responses, all output lines and prompting characters that the system prints or displays are shown in black letters. All the lines you type are shown in red letters.
CTRL/x or <CTRL/x>	The phrase <CTRL/x> indicates that you must press the key labeled CTRL while you simultaneously press another key, for example <CTRL/C>, <CTRL/Y>, and <CTRL/Z>. In examples, this control key sequence is shown as ^x, for example ^Y, because this is how the system echoes control key sequences.

All numeric values in the text of this manual are represented in decimal notation unless otherwise noted. All numeric values in examples are represented in the notation that the debugger uses. For programs written in VAX-11 MACRO, most numbers are represented in hexadecimal notation by the debugger.

Unless otherwise specified, you terminate commands by pressing the RETURN key.

SUMMARY OF TECHNICAL CHANGES

This manual describes the VAX-11 Symbolic Debugger, Version 2. The following are the technical changes from the previous version.

The debugger now supports log files and command procedures. The debugger can produce log files that contain all the input commands that you type at the terminal and all the output displayed by the debugger. The debugger can accept input from command procedures instead of the terminal. Log files are controlled by the SET LOG, SET OUTPUT, SHOW OUTPUT, and SHOW LOG commands. Command procedures are executed by the @file-spec command. Echoing of commands in command procedures is controlled by the SET OUTPUT command.

The symbol table search rules and SET SCOPE command have been changed. If a symbol name is unique in the symbol table, specifying the symbol name is sufficient. By default, the debugger assumes that a symbol is in the scope that contains the current program counter (PC). If you have entered the SET SCOPE command, the debugger searches the scopes in the order specified. The SET SCOPE command accepts scope names and special scope names specifying the scope containing the current PC, the scope containing the PC of a preceding CALL, and the scope consisting of global symbol definitions.

The SET MODE command and the mode qualifiers have been changed. The SET MODE BYTE, WORD, LONG, ASCII, and INSTRUCTION commands have been replaced by the SET TYPE and the SET TYPE/OVERRIDE commands. (The SET MODE syntax for these data types has been retained for compatibility with the previous version.) The SET MODE GLOBAL, NOGLOBAL, SCOPE, NOSCOPE, NOASCII, and NOINSTRUCTION commands are not supported in this version.

The debugger now has a help facility. The HELP command displays a brief description of each debugger command.

The previous location specifier, the circumflex (^), is no longer dependent on the current data type. It now represents the address that is 4 less than the current address, as represented by the period (.).

This version of the debugger treats symbols that are typed by a language compiler differently from the previous version. In this version, SET TYPE/OVERRIDE and type qualifiers override the type of the symbols. Previously, typed symbols were always displayed as their compiler type; mode qualifiers had no effect.

The debugger supports the new H floating[1], G floating[1], and octaword[1] data types and instruction mnemonics.

---

1. Not all VAX-11 processors support these data types and instructions.

CHAPTER 1

INTRODUCTION TO THE VAX-11 SYMBOLIC DEBUGGER

One of the most difficult stages in program development is locating and correcting errors after you have successfully written and compiled or assembled a source program. This stage is commonly called "debugging". It occurs when you attempt to run the successfully compiled or assembled program and receive erroneous output. Since you have followed all the rules of the source language and have not violated any constraints of the compiler or assembler, it is likely that the incorrect results are caused by one or more programming errors.

To help you find such errors, VAX/VMS 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 track down the mistakes, you can edit your source program, recompile or reassemble, relink, and execute the corrected version.

1.1 VAX-11 SYMBOLIC DEBUGGER FACILITIES

The VAX-11 Symbolic Debugger includes many features to help you:

- It is interactive. You control your program and interact 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 various languages. The debugger lets you converse in the language of your source program. You can change from one language to another in the course of a debugging session.
- It permits a variety of data forms. You can control the way 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 notation. Data can be represented by symbols, expressions, VAX-11 MACRO instructions, ASCII character strings, or numeric strings in decimal, octal, or hexadecimal notation.

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 Beginning 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 accessible in a debugging session. When you link the object program, you include a qualifier to make the debugger available to the program.

Then, 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. To terminate the debugging session, use the EXIT command.

See Chapter 2 for more information on beginning and ending debugging sessions.

1.2.2 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.

1.2.3 Evaluating Expressions

You can use the debugger as a calculator to compute the value of expressions, perform radix conversions, compute an address value, and so on.

1.2.4 Breakpoints

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

1.2.5 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 trace the execution of branch and call instructions by specifying a set of instruction opcodes you want traced.

1.2.6 Watchpoints

A watchpoint refers to a specific block of memory locations and causes the program to stop whenever that block is modified. Thus, you can monitor addresses to ensure that they are not being modified incorrectly.

1.2.7 Initiating Program Execution

The GO command starts or continues program execution; the STEP command also starts or continues program execution but executes only part of your program. The STEP command can execute one or a specified number of instructions or lines.

When you initiate your program with GO, the program executes until a breakpoint is executed, a watchpoint is modified, or the program terminates.

1.2.8 Log Files and Command Procedures

The debugger can produce log files and can accept input from command procedures. Log files contain all the input commands that you type at the terminal and all the output displayed by the debugger. You can use log files as a record of your debugging sessions or as a way of creating command procedures. Command procedures can be created as log files or by using a text editor. You can use command procedures to reproduce a debugging session several times, to continue a previous session, or to check a series of items in a program being developed.

1.3 SYMBOLIC REFERENCES

The debugger lets you refer to locations symbolically. Thus, if you have 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.

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 have 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 module that you specify first in the LINK command. 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.

1.3.2 Local Symbol Definition

To ensure that symbols local to your source program appear in the debuggers 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 and link the program. For VAX-11 MACRO, the qualifier is /ENABLE=DEBUG. For language compilers and the linker, the qualifier is /DEBUG.

1.3.3 Scope

If a symbol name is unique in a program, then you only need to refer to the symbol name; but if two or more symbols with the same name appear in different parts of the program and are both in the debugger symbol table, you must differentiate between them. The debugger allows you to specify the scope, or part of the program, in which the symbol is defined. In VAX-11 MACRO, modules are used to specify the scope because, within each module, symbol names must be unique.

By default, the debugger assumes that a symbol is in the scope which contains the current program counter (PC), that is the module that is currently being executed. However, if you have entered the SET SCOPE command, the debugger searches the specified scopes for the symbol.

1.3.4 Pathnames

If you wish to specify a symbol that is not in the default (PC) scope, you can specify a pathname. Pathnames have the general form of:

```
scope\symbol-name
```

For VAX-11 MACRO a pathname has the form of:

```
module-name\symbol-name
```

1.4 DEBUGGING IN LANGUAGES OTHER THAN VAX-11 MACRO

If you are programming in a language other than VAX-11 MACRO, the debugger lets you enter symbols, expressions, and addresses in terms that are compatible with your source language. The debugger sets the default language to the language used in the first module specified in the LINK command. If you want to change the language, you can use the SET LANGUAGE command. The language affects how the debugger displays and interprets the following:

- Expressions -- the debugger accepts expressions that are valid in the source language.

- Addresses -- the debugger displays and accepts line numbers and labels that are valid in the source program. For example, in FORTRAN the debugger accepts %LINE 10 to specify the source line number 10 and %LABEL 20 to specify the program label 20, and in BASIC the debugger accepts %LINE 500.3 to specify the third statment on line 500.

- Radixes -- the debugger uses the default radix of the language.

- Step Types -- the debugger steps according to source program line when the language supports line numbers.

- Special Symbols -- the debugger defines certain special symbols that can be used in expressions. Some symbols are not available in all languages.

- Scopes and Pathnames -- the debugger uses scopes and pathnames that are compatible with the language. For example, in VAX-11 BLISS the scope is defined as a routine and a pathname consists of the module followed by any number of routines followed by the symbol name.

- Typed Symbols -- the debugger uses symbol typing information provided by the language compiler to correctly display floating point numbers, character strings, and other data types. You can override the symbol type if you want to.

The debugger uses the language-dependent expressions, address specifiers, radix, step type, and special symbols based on the default language or the language specified with SET LANGUAGE. The debugger uses the scope, pathnames, and typed symbols compatible with the language of the source module; setting the language does not change the way the debugger handles scope, pathnames, and typed symbols.

This manual only documents debugging in MACRO, but notes features that are different in other languages. See the appropriate language user's guide for more information on debugging in your language.

CHAPTER 2

BEGINNING AND ENDING A DEBUGGING SESSION

You begin a debugging session by initiating the debugger with the DCL RUN or DEBUG command. You end a debugging session with the debugger EXIT or <CTRL/Z> command.

2.1 INITIATING THE DEBUGGER

To initiate the debugger, you usually follow these steps:

1. Assemble your program with the /ENABLE=DEBUG qualifier or compile your program with the /DEBUG qualifier.

2. Link your program with the /DEBUG qualifier.

3. Run your program.

When you run your program, the debugger prints an informational message and the DBG> prompt. You can then enter any of the debugger commands described in Chapter 6.

Example

```
$ MACRO/ENABLE=DEBUG MAINPR,SUBPR1
$ LINK/DEBUG MAINPR,SUBPR1
$ RUN MAINPR

        VAX-11 DEBUG      V2.00

%DEBUG-I-INITAL, language is MACRO,  module set to 'MAINPR'
DBG>
```

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

If you receive the RMS "file not found" message in response to a RUN command, it means either that your program file could not be found or that the debugger could not be found.

If you assemble or compile your program without the /ENABLE=DEBUG or /DEBUG qualifier, you can still link your program with the debugger, but you will not have access to certain symbols.

Table 2-1
Command Qualifiers

Assembler or Compiler	LINK Command	RUN Command	Effect
/ENABLE=DEBUG /DEBUG	/DEBUG	[/DEBUG][1]	Allows full symbolic debugging and initiates the debugger
/ENABLE=DEBUG /DEBUG	/DEBUG	/NODEBUG	Allows full symbolic debugging but defers debugging
[/DISABLE=DEBUG] [/NODEBUG]	/DEBUG	[/DEBUG]	Allows limited symbolic debugging (global symbols are accessible) and initiates the debugger
[/DISABLE=DEBUG] [/NODEBUG]	[/NODEBUG]	[/DEBUG]	Allows limited symbolic debugging (few symbols are accessible) and initiates the debugger
[/DISABLE=DEBUG] [/NODEBUG]	[/NODEBUG]	[/NODEBUG]	Allows limited symbolic debugging (few symbols are accessible) and defers debugging
/DISABLE=TRACE /DEBUG=NOTRACE	/NOTRACE	--	Inhibits the debugger (RUN/DEBUG does not initiate the debugger)

1. Brackets indicate that the qualifier is the default qualifier.

If you link your program without the /DEBUG qualifier, you can still initiate the debugger by specifying the /DEBUG qualifier in the RUN command, but you will not have access to most symbols.

You can inhibit or defer the debugger by specifying the /NODEBUG qualifier in the RUN command. In this case, your program will execute without the debugger's getting control. You need not specify /NODEBUG in the RUN command unless you specified /DEBUG in the LINK command.

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 DEBUG command. For example:

```
^Y

$ DEBUG
```

The debugger prints an informational message and the DBG> prompt. You will not know what line of your program was executing when the program was interrupted; however, you can find out by using the EXAMINE PC command to determine the contents of the program counter or by using SHOW SCOPE (if the scope is set to the current PC scope).

If you specify /NOTRACE when you link your program, you cannot initiate the debugger without repeating the link. /NOTRACE also inhibits the symbolic traceback of error messages.

When you run a program linked with /DEBUG, the image activator transfers control to the system's symbolic debugger shareable image or to the shareable image specified by the logical name LIB$DEBUG.

2.2 DEBUGGER START-UP CONDITIONS

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

```
        VAX-11 DEBUG     version number

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

    DBG>
```

The first message identifies the installed version of the debugger. The second message indicates that the debugger automatically has set its language to the language of the first module specified in the LINK command and has read the symbol information from that module into its symbol table. The DBG> prompt indicates that the debugger is now ready to process your commands.

The debugger uses the default scope defined by the currrent PC. In VAX-11 MACRO, the default scope is the module which contains the current PC. Note that when the debugger first gets control, the default scope is the module that contains the transfer address.

The debugger sets default modes and step parameters based on the language. For VAX-11 MACRO, the debugger sets the following defaults:

- MODES: symbolic, hexadecimal
- TYPE: long integer
- TYPE/OVERRIDE: none
- STEP: no system, by instruction, over routine calls

2.3 GETTING HELP

Once you have initiated the debugger you can get online help for any debugger command. The HELP command displays information about debugger commands, command qualifiers, and command parameters. To get help, type HELP followed by the command. The debugger displays a brief description of the command, the format of the command, and any additional help information that is available for that command.

Example

```
DBG>HELP SET OUTPUT

  SET

    OUTPUT

       Controls whether the debugger displays output on the
       terminal or writes it to a log file and controls whether
       the debugger verifies commands in command procedure files.
       SET OUTPUT controls whether a log file is being created;
       SET LOG controls the file-specification of the log file.

      Format:

       SET OUTPUT option [,option ...]

   Additional information available:

   Parameters LOG        TERMINAL   VERIFY
```

2.4 ENDING A DEBUGGING SESSION

You end your debugging session by entering the EXIT command to the DBG> prompt. You can also end your session by typing CTRL/Z.

The DCL command interpreter gains control and displays its prompt character ($). After exiting from the debugger, you cannot use the DEBUG or the CONTINUE command to reinvoke the debugger.

CHAPTER 3

DEBUGGER COMMAND FORMAT AND COMPONENTS

All debugger commands have the same general format. The basic components of debugger commands are:

- Command verb, keyword, qualifiers, and parameters
- Symbols and pathnames
- Special characters and expressions
- Entry and display modes and types

The following sections describe these command components.

3.1 DEBUGGER COMMAND FORMAT

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

```
cmd [keyword] [/qualifier] [param ...][DO(command[;command...])]!comment
```

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

keyword
: Indicates, in conjuction with command verb, the specific function to be performed by the command (CANCEL MODULE, SET SCOPE, SHOW LANGUAGE, etc.).

/qualifier
: Modifies the effect of the command.

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

DO (command[;command...])
: A list of debugger commands to be performed. Used only with SET BREAK commands. If you specify more than one command, separate them with semicolons.

comment
: Any text message. The debugger ignores all text after the exclamation mark.

You can enter more than one command on a command line by separating the commands with semicolons (;).

You can continue a command on a new line by ending the line with a hyphen (-); the debugger will then prompt for the rest of the command with an underscore (_).

Table 3-1 summarizes the debugger commands. See Chapter 6 for a complete description of the commands.

Table 3-1
Summary of VAX-11 Symbolic Debugger Commands

Command	Keyword	Function
@file-spec		Executes specified indirect command file
CALL		Calls a subroutine.
SET SHOW CANCEL	BREAK EXCEPTION BREAK LANGUAGE LOG MODE MODULE OUTPUT SCOPE STEP TRACE TYPE WATCH ALL	Initializes (SET), displays (SHOW), or deletes (CANCEL) the specified elements. Not all combinations can be used. For example, SET ALL is not a valid command. See individual command descriptions in Chapter 6.
DEFINE		Assigns an address to a symbol.
DEPOSIT		Puts data in a location in memory.
EVALUATE		Computes the value of an expression.
EXAMINE		Displays contents of an address or of a range of addresses.
EXIT		Terminates the debugging session or returns control from an indirect command file.
GO		Starts or continues program execution.
HELP		Displays information about debugger commands.
STEP		Executes one or a specified number of instructions or lines of the program and then stops.

All debugger commands, keywords, and qualifiers can be abbreviated. The underlined portion of the command and keyword in Table 3-1 is the minimum abbreviation.

Examples

```
DBG>SET MODU/ALL
DBG>SET TYPE WORD; EXAMINE TABLE+12
CALC\TABLE+12:  22A0
DBG>SET BREAK EVALSTAR; SET WATCH SUB1\DATA
DBG>SET BREAK SUB1\REDO DO (EXAMINE/BYTE COUNT)
DBG>GO
routine start at MAIN\MAIN
    .
    .
    .
```

3.2 SYMBOLS AND PATHNAMES

You use symbols and pathnames to reference addresses and data values in your program. A pathname consists of a scope and a symbol name separated by a backslash. It allows you to distinguish among symbols with the same name in different parts of your program.

3.2.1 The Debugger Symbol Table

Before you reference any symbol, you must ensure that the symbol is in the debugger symbol table. You use the SET MODULE command to map symbols from the image to the debugger symbol table.

The symbols included in the image depend on the commands used to assemble or compile and link the image. The image always includes program sections names, entry point names, and module names. If you specify /DEBUG in the LINK command, the image also includes the names of all global symbols. If you specify /ENABLE=DEBUG to the assembler or /DEBUG to the compiler and /DEBUG to the linker, then the image includes all local and global symbols defined in the source code.

When the debugger is initiated, only the global symbols and the local symbols from the first module in the LINK command are copied into the debugger symbol table. You should add the symbols from the other modules that you need to the debugger symbol table with the SET MODULE command. The following command adds the symbols from the modules SUBPR, CALC, and IOMOD:

```
DBG>SET MODULE SUBPR,CALC,IOMOD
```

NOTE

> The name of a module is not the file name of the module, but is the name specified by the assembler or compiler. In VAX-11 MACRO, the name of a module is determined by the .TITLE directive.

If your image contains fewer than about 2,000 symbols, you can add all the symbols in the image by entering the SET MODULE/ALL command. For example:

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

If you want to add symbols to the debugger symbol table and they will not fit, you must first delete some of the symbols by using the CANCEL MODULE command. The SHOW MODULE command lists all the modules in the image and indicates whether their local symbols are currently present in the debugger symbol table.

NOTE

The debugger stores symbol information for each module in one contiguous area. Consequently, if you have added and deleted several modules, it may be necessary to delete modules to make room for a large module if there is not enough free contiguous space.

3.2.2 Scopes and Pathnames

When you refer to a symbol without a pathname the debugger assumes by default that the symbol is in the scope (part of your program) that is currently being executed. If the debugger cannot find the symbol in the default scope, it searches the entire symbol table. If it finds only one definition of the symbol in the program, it uses that definition. If it finds more than one definition, it reports that it cannot find a unique symbol. If it cannot find any definition of the symbol, the debugger prints a message to that effect. When you want to access a symbol that is not in the current scope and is not unique, you must then change the default scope or explicitly specify the symbol's pathname.

You can specify a scope search list with the SET SCOPE command. The debugger searches for symbols in the scopes in the order specified in the SET SCOPE command. The SET SCOPE command has the format:

```
SET SCOPE scope1[,scope2,scope3...]
```

You specify scopes by the following methods:

- Name of Scope -- The name of the scope in general consists of the module name and block or routine names separated by backslashes. In VAX-11 MACRO, the name of the scope consists of the module name only. When you specify the name of a scope, the debugger adds the symbols for the module specified to the symbol table if they are not already included.

- 0 -- The digit 0 specifies the scope currently being executed. In VAX-11 MACRO, this is the module containing the current PC.

- \ -- The special symbol backslash specifies the scope containing the global symbols defined in the image.

- Number of Scope -- The number of the scope specifies scope by the level of active calls. The number 0 represents the scope currently being executed. The number 1 represents the scope that contains the PC in the first call frame on the call frame stack. This is the scope that contains the last executed CALL instruction. THe number 2 represents the scope containing the previous active CALL instruction, and so on. This method of representing scope is not generally useful for VAX-11 MACRO.

Examples

```
DBG>SET SCOPE MAIN,0,SUBPR1,\
```

After you enter this command, the debugger will search first for a symbol definition in the scope MAIN. If it cannot find the symbol there, it will search the scope that contains the current PC. If it cannot find it there, it will search the scope SUBPR1. If it cannot find it there, it will search the global symbol scope. If it cannot find it in any of these scopes, it will search all scopes for a unique symbol.

You can use the SHOW SCOPE command to display the current scope search list. If you want to return to the initial default scope (scope containing the current PC), you can use the CANCEL SCOPE command.

You can also specify the pathname of a symbol to ensure that you get the symbol you want.

In VAX-11 MACRO, a pathname has the form:

```
module-name\symbol
```

In high-level languages, a pathname has the form:

```
module-name\block1\block2\...\blockn\symbol
```

Block1, block2,...and blockn are language-dependent block or routine names.

When you specify a pathname, the debugger ignores the current scope search list. For example, the following command will display the contents of the symbol COUNTER in the scope SUPR even if there are other symbols with the name COUNTER in other scopes:

```
DBG> EXAMINE SUPR\COUNTER
```

If you reference a dynamically allocated variable when the PC is not within the scope that defines symbol, the debugger displays a warning message. This is to notify you that the symbol does not have an address assigned exclusively to it and that its address may have another use in the current section of your program. VAX-11 MACRO does not have any dynamically allocated variables. All BASIC variables and arrays except those declared in COMMON or MAP statements are dynamically allocated. FORTRAN dummy arguments are dynamically allocated.

The pathname for a global symbol is a backslash followed by the symbol name.

Examples

```
DBG> EXAMINE \KEYVAR
```

NOTE

When you are using a language-dependent address specifier, such as %LINE, the address specifier must appear before the entire pathname. For example:

```
DBG> SET BREAK %LINE SUB1\7
DBG> SET BREAK %LABEL SUB1\20
```

3.2.3 Kinds of Symbols

You can reference these kinds of symbols:

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

Symbols can specify virtual addresses or can contain data values. All VAX-11 MACRO labels specify addresses. Either instructions or data may be stored at that address in the image.

Most of the symbols that you use in your program are local symbols. Local symbols can only be referenced in the source program in the scope in which they are defined. Global symbols can be referenced from other scopes as well. In VAX-11 MACRO, global symbols are defined by the .ENTRY directive, the .GLOBAL directive, the double colon (::) label definition, and the double equal sign (==) symbol definition.

The debugger has the following permanent symbols. They can be used in any language and they cannot be redefined.

- R0 - R11 General registers 0 through 11
- AP Argument pointer
- FP Frame pointer
- SP Stack pointer
- PC Program counter
- PSL Processor status longword

You should avoid defining symbols in your source program that are the same as the debugger's permanent symbols. If you do, you will be able to access them only by specifying a pathname.

See Section 5.5 for more information on the processor status longword.

The DEFINE command lets you define symbols at any time during a debugging session to supplement or override existing symbols in your program. It is useful to define a symbol equivalent to a long pathname that is needed to access a frequently needed address.

Examples

```
DBG>DEFINE ITEM4=DATA_AREA\ARRAY+<4*4>
```

3.3 SPECIAL CHARACTERS AND EXPRESSIONS

This section 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 3-2, 3-3, and 3-4 summarize the arithmetic, address, and delimiting special symbols, respectively. Some characters (such as @) appear in more than one table because then are used in more than one way, according to context.

3.3.1 Evaluating Arithmetic Expressions

The debugger can perform integer arithmetic. In languages that support floating point expressions, the debugger can perform floating point arithmetic as well.

Table 3-2 lists special characters used in arithmetic expressions.

Table 3-2
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

NOTE

If you are programming in a language other than VAX-11 MACRO, the special symbols and expression syntax will be different. For example, in VAX-11 MACRO, angle brackets (<>) are used to determine operator precedence, but in most other languages parentheses are used. In general, an expression allowed in the source program will be accepted by the debugger with the exception of function references and some operators, such as the exponentation 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. 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 radixes, and indirect "contents of" operations (see Section 3.3.2) 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.

3.3.1.1 **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.

Examples

```
DBG>SET BREAK BEGIN + ^X10
DBG>EVALUATE ^D2000 + ^X1000 + ^O777
000019CF
```

3.3.1.2 **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 'change the sign of the following argument'.

Examples

```
DBG>CANCEL WATCH NAME - OFFSET
DBG>EXAMINE INQUEUE - 1000 - INDEX
MAIN\INQUEUE+0EDB:     01000C53
```

3.3.1.3 **Multiplication Operator (*)** - An asterisk multiplies the preceding argument by the following argument.

Examples

```
DBG>EVALUATE ^X50 * ^D512
0000A000
DBG>DEFINE PAGE = PAGE - 256 * 4
```

3.3.1.4 **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
```

3.3.1.5 **Shift Operator (@)** - An "at" sign (@) is the binary shift operator. 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
```

3.3.1.6 **Precedence Operators (< >)** - The debugger first evaluates terms or expressions enclosed by angle brackets. An expression can contain levels of nesting, with the debugger evaluating them in the order of innermost to outermost. The maximum number of levels is dependent on the complexity of the expression.

3.3.1.7 **Radix Operators (^D, ^X, and ^O)** - The debugger interprets numeric arguments in the current radix mode (see Section 3.4.1), unless you precede an 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 a radix operator and its operand.

Examples

```
DBG>EV ^D10+^D10
00000014
DBG>EV ^O77+^XFF
0000013E
DBG>EV 77+^XFF
00000176
```

3.3.2 Special Characters In Address Expressions

This section describes the significance of special characters that can be used to represent locations in address expressions. Table 3-3 lists the address representation characters.

Table 3-3
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 previous location, one longword before the last location addressed (as represented by .). The previous location is equal to the last location addressed minus 4. This operator is not available in some languages.
\	Represents the value last displayed by EXAMINE or EVALUATE. The backslash is also used in forming pathnames (see Section 3.2).
@	"Contents" operator. This operator is not available in some languages. The at sign is also used in the shift operator (see Section 3.3.1.5) and in executing command procedures (see Section 5.2).
:	Range operator (low address:high address) for the EXAMINE command; bit field operator for EVALUATE command (EVALUATE value <high bit:low bit>). The colon is also used in specifying the length of the ASCII data type (see Section 3.4.2).

3.3.2.1 **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.

Examples

```
DBG>EXAMINE /ASCII  MSG_1
ERR_MESSAGE\MSG_1:    NEZT
DBG>DEPOSIT/ASCII:1  . + 2 = 'X'
DBG>EXAMINE/ASCII   MSG_1
ERR_MESSAGE\MSG_1:    NEXT
```

The EXAMINE command sets the current location symbol to the examined address. You can then use this symbol in the DEPOSIT command's address expression to represent that location.

3.3.2.2 **Previous Location Symbol (^)** - A circumflex represents the last location addressed (by EXAMINE, DEPOSIT, SET BREAK, SET TRACE, or SET WATCH) minus 4. This operator is only useful with a data type of long integer, ASCII:4 (See Section 3.4.2) or other data type that is 4 bytes in length. This operator is not available in some languages.

Examples

```
DBG>SET MODE HEX; SET TYPE LONG
DBG>EX MAIN\TABLE
MAIN\TABLE:    00000001
DBG>EX MAIN\TABLE+4
MAIN\TABLE+4: 00000002
DBG>DEP ^ = 0FF
DBG>EX MAIN\TABLE
MAIN\TABLE:  000000FF
```

3.3.2.3 **Last Value Displayed Symbol (\)** - A backslash can be used to represent the value last displayed by EXAMINE or EVALUATE. When the last EXAMINE or EVALUATE is an EXAMINE/INSTRUCTION command, the backslash has an undefined value unless the instruction is a branch instruction. In that case, it has the value of the address specified in the branch instruction.

Examples

```
DBG>EVAL @R3 + < 2* @R4 > - 10
0000087F
DBG>EX/NOSYM \
0000087F:  000004E9
DBG>E/I MAIN+2
MAIN\MAIN+12:  BRB     SUB1\SUB1
DBG>E \
SUB1\SUB1:  0EFDE4800
```

3.3.2.4 **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. This operator is not available in some languages.

Examples

```
DBG>EXAMINE PC
PC:  00000448
DBG>EXAMINE/INSTRUCTION @PC
00000448:     MOVB   R0,L^MAIN\COUNT
```

The first EXAMINE reports the PC's current contents; the second EXAMINE displays the instruction stored at the address contained in the PC.

The command

```
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

```
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.

3.3.2.5 **Range Operator (:)** - A colon is used to specify 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 Section 3.3.3.2).

Examples

```
DBG>EX LABEL:LABEL+10
MAIN\LABEL: 0FFF28F7D
MAIN\LABEL+04:  0FFFFFFFF
MAIN\LABEL+08: 0B2EFFFFF
MAIN\LABEL+0C:  11FFFFFD
MAIN\LABEL+10:  000004E9
DBG>EX . : .+4
MAIN\LABEL+10:  000004E9
MAIN\LABEL+14:  00000000
DBG>E/I @PC : @PC + 10
MAIN\MAIN+02:  MOVAL  L^MAIN\A,R11
MAIN\MAIN+09:  MOVC5  #0A,L^00000200,#20,#14,B^0D8(R11)
```

3.3.3 Special Delimiting Characters

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

Table 3-4
Delimiting Characters

Character	Interpretation
/	Precedes command qualifiers that can be used to override current modes and types.
=	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 (see Section 3.2).
( )	Enclose DO command specifications in a SET BREAK command, or argument list in a CALL command.

(continued on next page)

Table 3-4 (Cont.)
Delimiting Characters

Character	Interpretation
;	Separates individual commands in a multiple command line, or in a DO command sequence associated with a SET BREAK command.
, (comma)	Separates multiple arguments.
' (apostrophe) or " (quotation mark)	Enclose ASCII string input or VAX-11 MACRO instruction input.[1]
<:>	Enclose bit field specification for EVALUATE command.[1]
-	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.[1]
!	Indicates that a comment follows.[1]

1. Discussed in following sections.

3.3.3.1 **Input String Delimiters** (' and ") - The debugger requires that input strings in ASCII or INSTRUCTION modes be enclosed by apostrophes or quotation marks. If you wish to enter an apostrophe within a string, use quotation marks to delimit the string, and vice versa. Otherwise, use matching characters of either type. Refer to ASCII and INSTRUCTION types (Section 3.4.2) for input restrictions.

Examples

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

3.3.3.2 **Bit Field Delimiters (<:>)** - A colon within angle brackets signifies a bit field specification that the EVALUATE command is to report on. The syntax is:

```
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).

Examples

```
DBG>EV 2468A<9:7>
00000005
DBG>EVALUATE @LOOP3<6:4>
00000003
```

3.3.3.3 **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 continuing line. You may continue a command line up to approximately 500 characters, exclusive of space and horizontal tab characters. If you continue a line that contains a comment, the debugger considers the continuing line part of the comment and ignores it.

Examples

```
DBG>EXAMINE/BYTE -
_BUFFER:BUFFER+3
CALCMOD\BUFFER:  3C
CALCMOD\BUFFER+1:  48
CALCMOD\BUFFER:2 0DE
CALCMOD\BUFFER+3 0EF
```

3.3.3.4 **Comment Operator (!)** - An exclamation mark specifies the beginning of a comment. The debugger ignores all characters after the exclamation mark on the line. Comments are usful in debugger command procedures (see Section 5.2). A comment can be continued on another line by ending the line with a hyphen. The debugger ignores all characters on the continuation lines.

Examples

```
DBG>EX IVAR !IVAR contains the character count
MODULE2\IVAR:  00000007
```

3.4 ENTRY AND DISPLAY MODES AND TYPES

The entry and display modes determine how the debugger interprets your entries and displays output. The entry and display modes control the current radix and whether addresses are displayed symbolically or by virtual address. The entry and display types control the current default data type. The following data types are supported: byte integer, word integer, longword integer, ASCII string, or MACRO instruction.

The SET MODE and SET TYPE commands set the modes and types. The SHOW MODE command displays the current modes and types; the SHOW TYPE command displays the current types.

You can override a mode or type by specifying a mode or type qualifier on an EXAMINE, EVALUATE, or DEPOSIT command.

3.4.1 Entry and Display Modes

The entry and display modes are:

- Radix modes

  Decimal
  Hexadecimal
  Octal

- Address display modes

  Symbolic
  Nosymbolic

3.4.1.1 **Radix Modes** - The radix mode determines how numeric values are entered and displayed. You can use the SET MODE command to specify the radix mode.

Examples

```
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 qualifier with the command. A radix mode qualifier controls the radix for all numbers entered in the command and displayed by the command. You can override the current radix mode or a radix mode qualifier for a number entered in a command by using a radix operator.

Examples

```
DBG>EVALUATE/HEX 15+15
0000002A
DBG>SET MODE DECIMAL
DBG>EV ^X15 + ^X15
42
```

Note that the resultant value is displayed in the current radix mode (in this example, decimal). See Section 3.3.1.7 for information on radix operators.

In DECIMAL mode, the debugger interprets entries and displays information in the decimal radix. In 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. Decimal values are signed; in all other radix modes, negative numbers are in 2's complement form.

In HEXADECIMAL mode, the debugger interprets entries and displays information in the hexadecimal radix. In VAX-11 MACRO, 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 the leftmost character of an entry is alphabetic, you must include a leading zero or use the hexadecimal radix operator to differentiate hexadecimal constants from symbols.

In OCTAL mode, the debugger interprets entries and displays information in the octal radix. In 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.

3.4.1.2 **SYMBOLIC/NOSYMBOLIC Modes** - SYMBOLIC and NOSYMBOLIC modes control how the debugger displays addresses. If the current mode is SYMBOLIC, the debugger displays addresses symbolically when possible. If the current mode is NOSYMBOLIC, the debugger displays addresses as numbers in the current radix mode. You can enter locations symbolically or numerically regardless of which mode is set.

In SYMBOLIC modes, 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 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, or 3, it reports the address value as a virtual address in the current radix mode. Probable causes of the virtual address display rather than a pathname are either that the respective module's symbol information is not present in the debugger symbol table or the address is beyond the bounds of the program.

In NOSYMBOLIC mode, the debugger reports the address value as a virtual address in the current radix mode.

3.4.2 **Entry and Display Types**

The entry and display types control the format and length of data. The types are:

- Byte integer
- Word integer
- Long integer
- ASCII[:length]
- Instruction

When the data type is byte integer, word integer, or long integer, the debugger displays data as integers in the current or specified radix mode. The length of the integer is a byte for byte integer, a word (2 bytes) for word integer, and a longword (4 bytes) for long integer.

If the data type is ASCII, the debugger displays values as ASCII strings. You can specify the length of the ASCII data type by following ASCII with a colon and a number in current radix. The number specifies the number of characters in the ASCII data type. If you do not specify a length, the debugger assumes a length of 4. When the data type is ASCII, the debugger accepts ASCII strings enclosed in quotation marks or apostrophes.

If the data type is instruction, the debugger attempts to decode values into instruction opcodes and operands. The length of the instruction is variable and depends on the opcode and the addressing modes used in the instruction. When the data type is instruction, you can enter VAX-11 MACRO instructions enclosed in quotation marks or apostrophes.

Some language compilers specify in the symbol table the data type of a symbol. The debugger by default displays these symbols in the type specified by the language compiler. In addition to the integer and ASCII data types, some languages allow floating point, double precision, character, and decimal string data types. VAX-11 MACRO does not provide the debugger with any data type information.

There are three ways to specify a data type:

- SET TYPE command
- SET TYPE/OVERRIDE command
- Type command qualifiers

The SET TYPE command controls the type for all data displayed except for data with a compiler-specified type or when a type command qualifier is specified. The initial default for the SET TYPE command is long integer.

The SET TYPE/OVERRIDE command controls the type for all data displayed except when a type command qualifier is specified. This command overrides any compiler-specified type. The initial default for the SET TYPE/OVERRIDE command is none; that is, compiler-specified types are not overriden. Note that if you abbreviate the OVERRIDE qualifier, you must specify at least OVERR.

Type command qualifiers control the type of all data displayed by the command. Type qualifiers override types specified by the SET TYPE and SET TYPE/OVERRIDE commands, and by compilers.

The SET TYPE and SET TYPE/OVERRIDE commands have the form:

```
SET TYPE specifier
SET TYPE/OVERRIDE specifier
```

The specifier can be one of the following data types:

- BYTE
- WORD
- LONG
- ASCII:length
- INSTRUCTION

where length is the number of characters (the length of the ASCII data type).

Examples

```
DBG>EX MAIN
MAIN\MAIN:  0EFDE483C
DBG>SET TYPE WORD
DBG>EX MAIN
MAIN\MAIN:  483C
DBG>EX X
MAIN\X:   1.000000
DBG>SET TYPE/OVERR WORD
DBG>EX X
MAIN\X:  4080
DBG>E/BYT X
MAIN\X: 80
```

CHAPTER 4

USING THE DEBUGGER

During a debugging session, you can examine and change data stored in your program, execute your program, and set breakpoints, tracepoints, and watchpoints to interrupt the execution of your program. Thus, you can determine where and how the errors are occurring.

4.1 EXAMINING AND DEPOSITING DATA

You use the EXAMINE and DEPOSIT commands to examine and modify data stored in your program. The EXAMINE command displays the contents of selected memory locations and registers.

The command format is:

```
EXAMINE[/qualifier] address[:address][,address[:address]]...
```

You can specify a mode and a type qualifier to override the current defaults as described in Section 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 TABLE, 1040
CALC\TABLE:  046B
00001040:  0EF40

DBG>EXAMINE/WORD TABLE:TABLE +14
CALC\TABLE:  046B
CALC\TABLE+02:  0000
CALC\TABLE+04:  08C2
CALC\TABLE+06:  0D7EF
CALC\TABLE+08:  0FFF3
CALC\TABLE+0A:  0AEFF
CALC\TABLE+0C:  0D004
CALC\TABLE+0E:  04AE
CALC\TABLE+10:  9850
```

```
CALC\TABLE+12:  22A0
CALC\TABLE+14:  0D450
```

To specify multiple ranges, use a command such as:

```
DBG>EXAMINE/WORD 1028:102E,103A:1040
00001028:  046B
0000102A:  03A2
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 may do so in any order.

If you examine one location by specifying an address and then wish to examine the next contiguous location, you need not specify the next address. An EXAMINE command with no address specification displays the contents of the next address after last address examined or deposited.

```
DBG>SET TYPE WORD
DBG>EXAMINE STAR\CODE
STAR\CODE:  046B
DBG>EXAMINE
STAR\CODE+2:  0000
```

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

```
DEPOSIT[/qualifier,...] address-expression=data[,data]...
```

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

The following sections describe how to examine and deposit numeric data, ASCII string data, and instructions.

4.1.1 Examining and Depositing Numeric Data

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

```
DBG>SET MODE  HEXADECIMAL , NOSYMBOLIC
DBG>SET TYPE/OVERR LONG
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 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    STAR\DATA1
STAR\DATA1:  0D0500AD0
```

The byte of data is deposited and verified by:

```
DBG>DEPOSIT/BYTE   STAR\DATA1 = 0FF
DBG>EXAMINE  .
STAR\DATA1:  0D0500AFF
```

The word of data is deposited and verified by:

```
DBG>DEPOSIT/WORD   STAR\DATA1 = 0FFFF
DBG>E .
STAR\DATA1:  0D050FFFF
```

The longword of data is deposited and verified by:

```
DBG>D  STAR\DATA1 = ^XFFFFFFFF
DBG>E .
STAR\DATA1:  0FFFFFFFF
```

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

```
DBG>E       4000
00004000:  77777777
DBG>D/BYTE   4002 = 0FF
DBG>E      4000
00004000:  77FF7777
```

If a symbol has a compiler-specified type, you can examine or deposit data without specifying a type unless you want to use a type different from the compiler-specified type. In the following example, XVALUE has a compiler-specified type of floating point.

```
DBG>SET MODE HEXADECIMAL
DBG>SET TYPE BYTE
DBG>E XVALUE
MAIN\XVALUE: 14.50000
DBG>E/BYTE XVALUE
MAIN\XVALUE: 68
DBG>DEP XVALUE=-3.2
DBG>E .
MAIN\XVALUE: -3.200000
```

4.1.2 Examining and Depositing ASCII Strings

You can examine and deposit ASCII string data by setting the default type to ASCII or by using the ASCII type qualifier. You can specify the length of the string by specifying ASCII:length, where length specifies the length of the ASCII string. If you do not specify a length, the debugger assumes a length of 4. Note that the debugger evaluates the number in the current radix mode.

The following example shows how to examine ASCII data.

Example

```
DBG>SET TYPE ASCII:8
DBG>EXAMINE TXT_AREA
SUB1\TXT_AREA: This is
DBG>EXAMINE
SUB1\TXT_AREA+8: ASCII da
DBG>EXAMINE/ASCII:12 TXT_AREA
SUB1\TXT_AREA: This is ASCII data
```

When you enter ASCII data, 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 /ASC:8  WINK  = "HI THERE"
DBG>DEPOSIT /ASC THINK  = "IT'S"
DBG>DEPOSIT /ASC PLINK  = '"1"'
```

The delimiter at the string's end must match the delimiter at the beginning, and must not appear within 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:10 TXT="abcdefghijklmnop"
DBG>DEP/BYT TXT+10=0D,0A
DBG>DEPOSIT/ASCII:0F TXT+12="Start new line"
DBG>E/AS:21 TXT
MAIN\TXT: abcdefghijklmnop
Start new line
```

4.1.3 Examining and Depositing Instructions

When you set the type to INSTRUCTION or use the INSTRUCTION type qualifier, the debugger can decode and display the instructions in your program and can encode and deposit the instructions you enter. This allows you to examine the instructions in your program and modify them. You can also use the debugger to assemble and execute short sequences of instructions to learn about the VAX-11 instruction set.

The following example illustrates how the EXAMINE command displays the contents of several locations as VAX-11 MACRO instructions:

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

PC relative displacements are evaluated and displayed symbolically (SYMBOLIC mode) or as virtual addresses (NOSYMBOLIC mode). The

storage requirements of VAX-11 MACRO instructions vary according to the instruction type, and number and complexity (addressing mode) of operands.

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

```
DBG>DEP/INS INCRS = 'ADDL3 #5,R3,R4'
```

The debugger interprets numeric values in the current radix mode.

Symbols can be included in instructions being deposited. However, symbolic expressions must not contain the backslash last value displayed symbol (see Section 3.3.2.3).

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.

4.1.3.1 **Replacing Instructions with DEPOSIT** - When entering an instruction, you must verify that the size of the instruction you are entering is less than or equal to 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 should not 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. 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>E/I BEGIN+12
SORT\BEGIN+12:  ADDL3   #10,R2,R4
DBG>E/I
SORT\TEST_SEQ: CMPB    (R0)[R2],(R0)[R4]
DBG>D/I BEGIN+12='ADDL2     #10,R2'
DBG>E/I
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 instruction. To ensure that the instruction executes correctly, you must enter one NOP instruction per leftover 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 above 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]
```

4.1.3.2 **Depositing a Sequence of Instructions** - The DEPOSIT command can accept an instruction sequence for entry but, as for any other command, you 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 separate DEPOSIT commands without having to compute the actual address in each case.

```
SET TYPE INSTRUCTION

DEPOSIT address-expression='instruction n'

EXAMINE

DEPOSIT . = 'instruction n+1'

EXAMINE

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

The DEPOSIT command causes the EXAMINE command with no address specification to display the contents of the address after the instruction. The EXAMINE command sets the current location symbol (dot) to the location where the next instruction should start. It also attempts to decode the contents of that location. Because the location may not be the start of an instruction, the debugger may not be able to decode the instruction or may display a meaningless instruction.

4.1.3.3 **Specifying Displacements in DEPOSIT** - The debugger will not use a default displacement length for the displacement or displacement deferred addressing modes. You must specify a B^, W^, or L^ displacement specifier (specifying byte, word, and longword displacements, respectively). If you do not specify the displacement, the debugger displays a warning message and does not deposit the instruction. The debugger uses a default longword displacement for the relative and relative deferred addressing modes but does allow you to specify an alternate displacement. For example:

```
DBG>DEP/INS SUB1\ROUT2+20 = 'MOVL B^4(R5), R6'
DBG>DEP/I SUB2\FINDIT+0A = 'CLRL W^2FF(R4)'
```

4.1.3.4 **VAX-11 MACRO Instructions with the Same Opcodes** - In VAX-11 MACRO, different instructions can generate the same opcode. For example, MOVAL and MOVAF both generate the hexadecimal opcode DE. The debugger, however, accepts and displays only one instruction for any of the opcodes that can be produced by multiple instructions. For example, if you use MOVAF in your source program, the debugger displays MOVAL.

Table 4-1 lists the instructions that have the same opcodes, lists the instruction accepted and displayed by the debugger, and lists the hexadecimal opcodes.

Table 4-1
Instructions Accepted and Displayed by the
Debugger for VAX-11 MACRO Instructions with Equivalent Opcodes

Instructions with Equivalent Opcodes	Instructions Accepted and Displayed by the Debugger	Opcode
BCC BGEQU	BGEQU	1E
BCS BLSSU	BLSSU	1F
BEQL BEQLU	BEQL	13
BNEQ BNEQU	BNEQ	12
CLRD CLRG CLRQ	CLRQ	7C
CLRF CLRL	CLRL	D4
CLRH CLRO	CLRO	7CFD
MOVAD MOVAG MOVAQ	MOVAQ	7E
MOVAF MOVAL	MOVAL	DE
MOVAH MOVAO	MOVAO	7EFD
PUSHAD PUSHAG PUSHAQ	PUSHAQ	7F
PUSHAF PUSHAL	PUSHAL	DF
PUSHAH PUSHAO	PUSHAO	7FFD

4.2 EVALUATING EXPRESSIONS AND BIT FIELDS

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

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

```
EVALUATE[/qualifier][...] expression[,...]
```

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

The EVALUATE command analyzes an expression in the context of the current language. The rules of precedence applicable to VAX-11 MACRO are described in Section 3.3. If you set the language to one that supports floating-point numbers, the debugger evaluates floating-point numbers. You should separate each floating-point number with a space.

Examples

```
DBG>EVAL 0A*<4+<3*8>>
00000118
DBG>EVAL @SUB1\D
0000001F
DBG>EVAL @SUB1\D * 3
0000005D
DBG>EVAL/DEC 10-1
9
DBG>SET LANG BASIC
DBG>EV 3.02 * 2.4 / 0.02
   362.4000
DBG>EVAL X, X / 2.5
   1.000000
  0.4000000
```

You can use EVALUATE to display the current contents of specified bits in a location. Note that the debugger does not allow you to evaluate bit fields in some languages. The syntax is:

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

You specify the bounds of a bit field by decimal integers, regardless of the current radix mode (unless you use radix operators, for example <^X0F:^X0>). 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 when the language is set to VAX-11 MACRO.

```
EVALUATE @address-expression<high-bit:low-bit>
```

Examples:

```
DBG>EX/WORD MAIN
MAIN\MAIN:  483C
DBG>EVAL @MAIN<15:0>
0000483C
DBG>EVAL @MAIN<8:0>
0000003C
DBG>EVAL @MAIN<6:4>
00000003
```

4.3 CONTROLLING PROGRAM EXECUTION

This section describes how you start or continue your program with GO, STEP or CALL. Section 4.4 describes how to interrupt your program at predetermined points or under certain conditions by means of breakpoints, tracepoints, watchpoints, and the CTRL/Y or CTRL/C command.

4.3.1 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 gain control, or the program runs to completion (refer to Section 5.3 for information about exception conditions).

The command format is:

```
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

```
start at mod\rtn
```

or the message

```
routine start at mod\rtn
```

These messages display the names of the module, mod, and of the routine, rtn, where execution starts.

If "routine" is included in the message, execution started at the beginning of a routine and "mod\rtn" is 2 less than the actual PC value (because of the routine entry mask).

If you enter a GO command subsequent to program suspension (at a breakpoint, for instance) 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.

4.3.2 Stepping Through Your Program

The STEP command lets you specify the number of instructions (VAX-11 MACRO) or lines (line-oriented language compilers only) that your program can execute before the debugger regains control. The basic command format is:

```
STEP [decimal-integer]
```

If you do not include a decimal integer (0 through 32767) or if 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 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.

The STEP command 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.

The STEP modes are:

LINE or INSTRUCTION
INTO or OVER
SYSTEM or NOSYSTEM

You can express these modes at command level as follows:

STEP [/qualifier[...]] [decimal-integer]

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

The STEP modes 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 languages, such as FORTRAN and BASIC (ignored for VAX-11 MACRO).
INTO	Step into a routine called by a call-type instruction (CALLS, CALLG, JSB, BSBB, BSBW).
OVER	Step over routines called by call instructions; that is, the call 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. Note this mode allows you to step through code in system space that is executing in user mode. (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.

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

```
SET STEP mode[,mode...]
```

Multiple mode entries must be separated by commas.

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

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

NOTE

In VAX-11 BASIC STEP/LINE executes a BASIC statement which may be on a line by itself, may be continued onto several lines, or may be one of several statements on the same line.

4.3.3 **Calling Routines**

The 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:

```
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 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.

A difference between calling a procedure and executing it with the GO or STEP commands is the way the debugger treats the general registers (R0 through R11). The debugger saves the current values in the registers before it calls the procedure. When the procedure executes a RET instruction, the debugger displays the value in R0 and then restores the general registers to values they had before you entered the CALL command.

4.4 INTERRUPTING EXECUTION OF YOUR PROGRAM

You can interrupt the execution of your program in five ways:

- By setting breakpoints -- The debugger interrupts execution of your program before the instruction at the breakpoint and then prompts you for a debugger command. You can set breakpoints at various points in your program and then observe and change the context of your program while it is suspended at that point.

- By setting tracepoints -- The debugger temporarily interrupts execution of your program before the instruction at the tracepoint, displays a message on the terminal, and then continues execution of your program. You can set tracepoints at various points in your program and then observe the flow of control in your program.

- By setting opcode tracing -- The debugger temporarily interrupts execution of your program before the specified control transfer opcode is executed. You can set opcode tracing and then observe the flow of control in your program. Opcode tracing allows you to follow the flow of control without having to set specific tracepoints.

- By setting watchpoints -- The debugger interrupts execution of your program after an instruction writes data at a watchpoint, and then it prompts you for a debugger command. You can set watchpoints at various data areas in your program and then observe and change the context of your program after your program has written to the data area. Watchpoints allow you to determine the section of your program that is incorrectly modifying data.

- By typing CTRL/Y -- The system stops execution of your program whenever you type CTRL/Y on the terminal. If you enter the DCL DEBUG command, the debugger gets control and prompts you for a debugger command. The CTRL/Y key command allows you to observe and change the context of your program when it is executing an infinite loop and there are no breakpoints set within the loop.

Breakpoints, tracepoints, and watchpoints are set with the SET BREAKPOINT, SET TRACEPOINT, and SET WATCHPOINT commands, respectively. Once set, they can be canceled with the CANCEL BREAKPOINT, CANCEL TRACEPOINT, CANCEL WATCHPOINT, and CANCEL ALL commands. You can determine what breakpoints, tracepoints, and watchpoints have been set by the SHOW BREAKPOINT, SHOW TRACEPOINT, and SHOW WATCHPOINT commands.

4.4.1 **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.

4.4.1.1 **Breakpoint Reporting at Program Stop** - When your program is suspended at a breakpoint, the debugger usually reports the location by:

```
break at 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 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 00000846
```

The debugger sometimes displays the report as:

```
routine break at location
```

Note that in this case the value shown is 2 less than the actual PC contents (because of the entry mask). 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.

4.4.1.2 **Continuing From a Breakpoint** - To continue your program from a breakpoint, you enter either a GO command or a STEP command. 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 (because of the entry mask), and "location" is an entry point.

4.4.1.3 **Setting Breakpoints** - Breakpoints are set at an 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, tracepoints, or watchpoints to clear sufficient table space for the new entry.

The debugger does not protect current breakpoints, tracepoints, and watchpoints against overwriting by a new request. The debugger simply replaces the previous breakpoint, tracepoint, or watchpoint specification with the new breakpoint 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 just enter the new specification for the same location.

4.4.1.4 **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. 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:

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

To verify the breakpoint, you can use the SHOW BREAK command.

4.4.1.5 **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 DO command sequence format is

```
DO(command[;command...])
```

The command list can include any complete debugger command. 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), abbreviated keywords, or command procedures (see Section 5.2).

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 need not be defined at the time you enter the SET BREAK command, because the debugger defers evaluating symbols 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.

4.4.1.6 **Breakpoint "After" Option** - If your program is to stop only on or after the nth pass through a breakpoint location, as in an iteration or conditional program loop, specify the breakpoint as follows:

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

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

NOTE

If you set a breakpoint with /AFTER:n, the debugger interrupts execution before the instruction is executed for the nth time, that is after it has been executed n-1 times.

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 some more times before the breakpoint takes effect.

4.4.1.7 **Temporary Breakpoints** - A temporary (or one-time) breakpoint stops your program once and then is canceled automatically. You specify such a breakpoint by:

```
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.

4.4.1.8 **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:

```
CANCEL BREAK address-expression
```

When canceling a breakpoint, you cannot 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 displays the NOSUCHBPT, no such breakpoint message.

To cancel all breakpoints, type:

```
CANCEL BREAK/ALL
```

4.4.1.9 **Showing Breakpoints** - You can display the current breakpoints and a description of any breakpoint actions that were specified, by typing:

```
SHOW BREAK
```

The debugger responds with:

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

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 NOBREAKS, no breakpoints are set message.

4.4.1.10 **Breakpoint Examples** - The following examples illustrate use of the SET, CANCEL, and SHOW BREAK commands.

```
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:9 @R7)
```

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 nine bytes beginning with the location that is addressed by the contents of general register R7.

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

Sets a breakpoint at virtual address 7249 (hexadecimal).

```
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.

In SYMBOLIC mode (the initialized condition):

```
DBG>SHOW BREAK
routine breakpoint at SORT\INSEQ do(set scope inseq)
breakpoint /after:4 at SORT\SEQCHK
```

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

In NOSYMBOLIC mode:

```
DBG>SHOW BREAK
routine breakpoint at 0000846 do (set scope inseq)
breakpoint/after:4 at 000082A
```

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

4.4.2 **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 lets you know whether unanticipated control transfers are occurring as your program is running. There are two basic forms of tracing: tracepoints, and opcode tracing.

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. The debugger uses the breakpoint table to store information about tracepoints.

Opcode tracing is when the debugger reports the occurrence of each instruction of a specified type, such as call instructions and branch instructions.

You can specify tracing at the following locations:

- At the first byte of specified instruction locations (that is, set tracepoints). The debugger treats tracepoints in a similar way as it treats breakpoints with DO(GO) command sequences.

- At all instructions that call subroutines in your program (includes all CALLG, CALLS, RET, JSB, BSBW, BSBB, and RSB instructions).

- At all branch instructions in your program (includes all branches and JMP; excludes subroutine-type instructions).
- At both instructions that call subroutines and branch instructions.

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

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

If the message is displayed as:

```
routine trace at 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.

4.4.2.1 **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.

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

```
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.) If any tracepoint, breakpoint, or watchpoint was set previously at the specified address, the debugger replaces it with the new tracepoint.

To ensure that the tracepoint is at the start of an instruction, you can use the "current location" symbol, as follows:

```
EXAMINE/INSTRUCTION .
```

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

To trace all instructions that call subroutines, type:

```
SET TRACE/CALL
```

To trace all branch instructions, type:

```
SET TRACE/BRANCH
```

To trace both forms of control transfer instructions, simply enter both forms of SET TRACE commands in either order. You cannot specify both qualifiers in one SET TRACE command.

To trace both forms of control transfer instructions, type:

```
SET TRACE/BRANCH
SET TRACE/CALL
```

4.4.2.2 **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:

```
CANCEL TRACE address-expression
```

To cancel tracing of instructions that call subroutine, type:

```
CANCEL TRACE/CALL
```

To cancel tracing of branch instruction, type:

```
CANCEL TRACE/BRANCH
```

To cancel all tracepoints and opcode tracing, type:

```
CANCEL TRACE/ALL
```

4.4.2.3 **Showing Tracing Modes** - You can determine where tracepoints are set, and the form of tracing in effect by using the command:

```
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.

If the debugger does not find tracepoints set, and no opcode tracing is in effect, it displays the NOTRACES, no tracepoints are set, no opcode tracing message.

4.4.2.4 **Tracing Examples** - The following examples illustrate the SET, CANCEL, and SHOW TRACE commands.

```
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).

```
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.

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
```

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
```

4.4.3 **Watchpoints**

Watchpoints are selected program locations you monitor to determine when instructions have modified these locations. This section describes watchpoints and the use of the commands SET WATCH, SHOW WATCH, and CANCEL WATCH, to establish, report the status of, and delete 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 default data type is ignored.

4.4.3.1 **Watchpoint Reporting** - When your program writes into a watchpoint location, the debugger stops the program and reports the following:

```
write to location1 at PC location2
      old value = value1
      new value = value2
```

Where location1 indicates the location that was modified, location2 indicates the location of the instruction that did the writing, and value1 and value2 are the old and new values, respectively.

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

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 notation.

4.4.3.2 **Continuing from a Watchpoint** - To continue your program from a watchpoint, enter either a GO command or a STEP command. The debugger reports the resumption of program execution by:

```
start at location
```

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

4.4.3.3 **Setting Watchpoints** - You specify a watchpoint request by:

```
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. If any watchpoint, breakpoint, or tracepoint was set previously at the specified address, the debugger replaces it with the new watchpoint.

4.4.3.4 **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:

```
CANCEL WATCH address-expression
```

where address-expression specifies the lower bound of a watched range. If you specify CANCEL WATCH/ALL, all watchpoints are canceled.

If the debugger cannot find the specified watchpoint, it displays the NOSUCHWPT, no such watchpoint message.

4.4.3.5 **Showing Watchpoints** - You can determine where current watchpoints are set by typing:

```
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. The value nnn, in decimal, indicates how many bytes are monitored by the associated watchpoint.

4.4.3.6 **Watchpoint Examples** - The following examples illustrate the SET, CANCEL, and SHOW WATCH commands.

```
DBG>SET WATCH TERMINAL_IO\DATA1
```

The debugger watches the location identified by the pathname, TERMINAL_IO\DATA1.

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

The debugger watches virtual address 7249 (hexadecimal).

```
DBG>CANCEL WATCH TERMINAL_IO\DATA1
DBG>CANCEL WATCH SORT\TABLE
DBG>CANCEL WATCH ^X7249
```

The debugger cancels the specified watchpoints.

With SYMBOLIC MODE on (the initialized condition):

```
DBG>SHOW WATCH
watchpoint at SORT\TABLE for 4.  bytes
watchpoint at SORT\INTVAR for 2.  bytes
```

The debugger reports the current watchpoint locations by pathnames.

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.

4.4.3.7 **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, an exception is generated. The debugger handles the exception and restarts the instruction. If the instruction writes to the watched location(s), the debugger interrupts 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. This is because the system service checks to see if user mode code has write access to the page. It cannot tell that the debugger has set the page protection and will allow write access after generating the exception. 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.

You cannot set a watchpoint on a dynamically allocated variable. Most variables and arrays in VAX-11 BASIC are dynamically allocated. Dummy arguments in VAX-11 FORTRAN are also dynamically allocated.

4.4.4 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.

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.

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

CHAPTER 5

ADDITIONAL DEBUGGING FEATURES

You can save a record of your debugging session in a log file, execute commands from a file, interrupt your program when an exception is encountered, show the procedures that have been called, access the processor status longword (PSL), and debug exit handlers.

5.1 SAVING A RECORD OF A DEBUGGING SESSION

The debugger can create a log file that provides a record of your debugging session. When the debugger creates a log file, it writes to the log file the commands that you enter at the terminal and its responses to these commands. Log files can be used as debugger command procedures (see Section 5.2).

The SET OUTPUT command controls the debugger's output configuration. The SET LOG command is used to specify the name of the log file. The SHOW OUTPUT and SHOW LOG commands display the current status of the debugger output configuration and log files.

The SET OUTPUT command controls whether output is written to a log file, whether output is displayed on the terminal, and whether commands executed from a command procedure or DO command sequence are echoed.

If you want to create a log file and continue to have the debugger display information on the terminal, enter the command:

```
DBG>SET OUTPUT LOG
```

When output is set to LOG, the debugger creates a log file if one does not exist. By default the debugger creates a log file with the file specification DEBUG.LOG with a version number equal to the highest existing version number plus one. You use the SET LOG command to specify a different name for the log file. When the output is set to LOG, the debugger writes all commands that you enter on the terminal and its responses to these commands to the log file. The debugger writes the same thing to the log file that it would to the terminal except that it does not write the DBG> prompt and it precedes each of its responses with an exclamation mark (the debugger comment character). This allows the log file to be used without changes as a debugger command procedure.

When you create a log file, you may want to enter commands to preserve the output in the log file. If you are generating a large amount of output, it would be time consuming to display it on the terminal as well. To create a log file and suppress output on the terminal, enter the command:

```
DBG>SET OUTPUT LOG,NOTERMINAL
```

When the output configuration is set to NOTERMINAL, the debugger does not display its responses on the terminal except for informational, warning, and error messages. Note that if you set the output to NOLOG and NOTERMINAL, the debugger displays a warning message indicating that output is being lost.

If the output configuration is set to VERIFY, the debugger echoes commands executed from a command procedure or a DO command sequence. If the output is set to TERMINAL, these commands are displayed on the terminal. If the output is set to LOG, these commands are written into the log file (preceded by an exclamation mark).

If you do not want to use the default log file specification, DEBUG.LOG, you can enter the SET LOG command. If you specify a version number in the SET LOG command and that version already exists, the debugger appends the new log file to the existing log file. The SET LOG command has the format:

```
SET LOG file-spec
```

When you specify the file specification in SET LOG, you must enclose it in quotation marks if the first character is not an alphabetic or numeric character. If the output is set to NOLOG when you enter the SET LOG command, the debugger saves the file specification until the output is set to LOG. Then the debugger creates the specified log file. If the output is set to LOG when you enter the SET LOG command (the debugger is currently writing to a log file), the debugger closes the existing log file and creates a new one with the specified name.

Examples

```
DBG>SHOW OUTPUT
output: noverify, terminal, not logging to DEBUG.LOG
DBG>SET LOG COMPTEST
DBG>SET OUTPUT LOG
DBG>SHOW OUTPUT
output: noverify, terminal, logging to DB1:[PROJ.START]COMPTEST.LOG;1
DBG>SET MODULE/ALL
DBG>SET BREAK SUB1
DBG>GO
routine start at MAIN\MAIN
routine break at SUB1\SUB1
DBG>EX X
SUB1\X:  1
DBG>SET OUTPUT NOTERMINAL
DBG>EX/WORD MAIN
DBG>E/I
DBG>E/I
DBG>SET OUT TERM
DBG>GO
routine start at SUB1\SUB1
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completio
DBG>EXIT
```

This debugging session creates the file COMPTEST.LOG, which contains the following text.

```
SHOW OUTPUT
!output: noverify, terminal, logging to DB1:[PROJ.START]COMPTEST.LOG;1
SET MODULE/ALL
SET BREAK SUB1
GO
!routine start at MAIN\MAIN
!routine break at SUB1\SUB1
EX X
!SUB1\X:  1
SET OUTPUT NOTERMINAL
EX/WORD MAIN
!MAIN\MAIN:  18492
E/I
!MAIN\MAIN+02:  MOVAL  L^MAIN\A,R11
E/I
!MAIN\MAIN+09 :  MOVC5     #0A,L^00000200,#20,#14,B^0D8(R11)
SET OUT TERM
GO
!routine start at SUB1\SUB1
!%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
EXIT
```

5.2 USING COMMAND PROCEDURES

A command procedure is a file that contains a series of debugging commands; the @file-spec command instructs the debugger to accept commands from a command procedure. If you have a series of debugger commands that you want to enter in more than one debugging session, you can put them in a command procedure.

You can create a command procedure by using an editor or by creating a log file. The debugger executes the commands that are in the file. The debugger ignores any line that starts with an exclamation mark (!). When the debugger finds an EXIT command in the command procedure or when it gets to the end of the file, it stops accepting commands from the file. If the @file-spec command was entered at the terminal, the debugger accepts the next command from the terminal. If the @file-spec command was in another command procedure, the debugger executes the next command in that procedure. If the @file-spec command was in a DO command sequence, the debugger executes the next command in the sequence.

By default, the debugger does not display commands read from a command procedure on the terminal or write them in the log file. The SET OUTPUT VERIFY command causes the debugger to display this information on the terminal and write it to the log file (if the output is set to LOG.

Examples

The following is a command procedure, MAIN.COM, that has been created with an editor.

MAIN.COM

```
! Command Procedure to set up breakpoints in MAIN and SUB1
SET MODU/ALL                ! Make sure all the modules are in
SET BR SUB1                 ! Set up breakpoint at SUB1 routine
                            ! entry
GO                          ! Start the program
E/WORD SUB1                 ! Display entry mask
E/I                         ! Display first two instructions of
E/I                         ! SUB1
CALL SUB2(X)                ! Call SUB2
EXIT
```

The following is a debugging session that executes the command procedure.

```
DBG>SET OUT VER
DBG>@MAIN
%DEBUG-I-VERIFYICF, entering indirect command file "MAIN"
 ! Command Procedure to set up breakpoints in MAIN and SUB1
 SET MODU/ALL               ! Make sure all the modules are in
 SET BR SUB1                ! Set up breakpoint at SUB1 routine
                            ! entry
GO                          ! Start the program
routine start at MAIN\MAIN
routine break at SUB1\SUB1
 E/WORD SUB1                ! Display entry mask
SUB1\SUB1:  4800
 E/I                        ! Display first two instructions of
SUB1\SUB1+02:  MOVAL   L^SUB1\Y,R11
 E/I                        ! SUB1
SUB1\SUB1+09:  INCL          @B^4(AP)
 CALL SUB2(X)               ! Call SUB2
routine start at SUB2\SUB2
value returned is 7FFF0A48
 EXIT
%DEBUG-I-VERIFYICF, exiting indirect command file "MAIN"
DBG>
```

5.3 EXCEPTION CONDITIONS

Exception conditions are conditions that interrupt the execution of your program. In the context of the debugger, there are two kinds of exception conditions: exception conditions caused by the debugger and external exception conditions. The debugger uses exception conditions to interrupt your program's execution. External exception conditions are caused by your program; these conditions would occur even if you had not initiated the debugger.

This section describes the debugger's response to external exception conditions. It does not describe the cause and effect of external exception conditions, nor does it describe how to write handler routines for them. See the VAX/VMS System Services Reference Manual and the VAX-11 Run-Time Library Reference Manual for this information.

The SET EXCEPTION BREAK command causes the debugger to treat an external exception condition as a breakpoint. If you enter SET EXCEPTION BREAK, the debugger will not execute any exception handlers that you have defined in your program. When an exception condition is encountered, the debugger gets control and then prompts you for a command.

If you have not entered a SET EXCEPTION BREAK command or have entered a CANCEL EXCEPTION BREAK command, the following sequence occurs after an exception condition is encountered: (1) the debugger gets control and transfers control to the exception handlers; (2) if you have defined an exception handler in your program, it is executed; (3) if you have not specified an exception handler or if it resignals the exception, the VMS exception handler is executed and it displays a message and then returns control to the debugger (if possible) which prompts you for a command. Note that if your program's execution handler continues execution, the debugger does not get control until the program terminates or is interrupted by the debugger (for example, at a breakpoint).

5.4 SHOWING ACTIVE CALLS

The SHOW CALLS command reports various information concerning the current nested procedure calls. The SHOW CALLS command displays information in the same format as the traceback handler, which displays information on the nested procedure calls when the image has been run without the debugger and an error has been detected. The command format is:

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

You have the option of requesting that all call levels be reported (the default) or 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 to the SHOW CALLS command with the following report:

```
module name     routine name     line     relative PC     absolute PC
```

The first line in the report refers to the routine currently being executed. 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 — The module in which the call occurred. If the debugger symbol table does not include symbol information for the module in which the call occurred, the module name remains blank.

routine name — The routine or program section name in which the call occurred.

line — Used only for line-oriented languages to identify the line number of the call. Left blank for VAX-11 MACRO.

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

absolute PC — The virtual address of the call in hexadecimal radix mode, 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.

Examples

```
DBG>SHOW CALLS
module name     routine name      line    relative PC     absolute PC

SUB2            SUB2                      00000002        0000085A
                SUB1                 5    00000014        00000854
                MAIN                10    0000002C        0000082C
```

5.5 PROCESSOR STATUS LONGWORD (PSL)

This section describes how to display the current contents of the processor status longword (PSL), and how to alter its contents. For a detailed description of the PSL, see the VAX-11 Architecture Handbook.

5.5.1 Displaying the Processor Status Longword

To display the current contents of the processor status longword (PSL), type:

```
DBG>EXAMINE PSL
```

The debugger responds with:

```
PSL:  CMP TP FPD IS CURMOD PRVMOD 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
```

5.5.2 Altering the Processor Status Longword

You can alter the PSL's low-order word. However, you cannot alter the the PSL's high-order word, regardless of the privileges allocated to your account. If you attempt to modify the high-order word you will get a system reserved operand fault error message when your program is executed. The following conditions are in the PSL's high-order word:

- CMP - compatibility mode
- TP - trace pending

- FPD - first part done
- IS - interrupt stack
- CURMOD - current mode
- PRVMOD - previous mode
- IPL - interrupt priority level

You cannot alter the PSL symbolically, but you can alter it by depositing a value. To alter the PSL, enter the following command:

```
DEP/HEX/WORD PSL = expression
```

where "expression" is the sum of the key numbers selected from Table 5-1 according to the bits that you want set. If you want a bit set, include the key number in the expression. If you want a bit clear, do not include the key number in the expression. The /WORD qualifier ensures that only the low-order word of the PSL is altered and that the high-order word remains unchanged. Note that setting the trace trap condition code may cause a T-bit pending trap.

Table 5-1
PSL Low-Order Word Alteration Values

Bit	Key	Key Number (Hex)	Description
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	DV	80	Decimal overflow trap enable
6	FU	40	Floating underflow trap enable
5	IV	20	Integer overflow trap enable
4	T	10	Trace trap condition code
3	N	8	Negative condition code
2	Z	4	Zero condition code
1	V	2	Overflow condition code
0	C	1	Carry condition code

5.6 DEBUGGING EXIT HANDLERS

Exit handlers are invoked when your program terminates. This section describes the way the debugger interacts with any exit handlers that are declared in your program. It does not describe how to write exit handlers. See the VAX/VMS System Services Reference Manual for this information.

The debugger declares its own exit handler but its handler is executed after any user-declared exit handlers. When your program terminates, the system executes any exit handlers that are declared in your program. If any breakpoints are set in the user-declared handler, execution is interrupted and the debugger displays its prompt. Thus you can debug your exit handler in the same way that you can debug any routine in your program.

When you enter the EXIT or CTRL/Z command to the debugger prompt, the system also executes any exit handlers declared in your program, but the debugger will not get control even if you have set breakpoints. When you enter the EXIT or CTRL/Z command the debugger removes all breakpoints, tracepoints, opcode tracing, and watchpoints. Consequently, if you want to debug exit handlers, your program must terminate; you cannot debug them by entering the EXIT or CTRL/Z command.

CHAPTER 6

DEBUGGER COMMANDS

This chapter contains descriptions of all the debugger commands. The commands are arranged in alphabetical order. Appendix C, a brief command summary, specifies the minimum abbreviation for each command, keyword, and qualifier.

@file-spec

Description

Instructs the debugger to begin taking commands from the indicated file. Command procedures, also called indirect command files, can be invoked wherever any other DEBUG command can be given. Any valid debugger command can occur in a command procedure.

The command procedure can contain any debugger command, including another @file-spec command. When the debugger executes an EXIT command in a command procedure or reaches the end of the file, it returns control to the command stream that invoked the command procedure. A command stream can be the terminal, a previous command procedure, or a DO command sequence in a SET BREAK command. If you enter SET OUTPUT VERIFY, all commands read from a command procedure are echoed on the terminal.

Format

@file-spec

Command Parameters

file-specification

Specifies the command procedure to be executed. The default file type is COM. Do not use quotation marks to delimit the file-spec.

Command Qualifiers

None.

Examples

```
DBG>SET OUT VER
DBG>@MAIN
%DEBUG-I-VERIFYICF, entering indirect command file "MAIN"
!  Command Procedure to set up breakpoints in MAIN and SUB1
SET MODU/ALL                ! Make sure all the modules are in
SET BR SUB1                 ! Set up breakpoint at SUB1 routine
                    ! entry
GO                          ! Start the program
routine start at MAIN\MAIN
routine break at SUB1\SUB1
E/WORD SUB1                 ! Display entry mask
SUB1\SUB1:  4800
 E/I                       ! Display first instruction
SUB1\SUB1+02:  MOVAL  L^SUB1\Y,R11
 EXIT
%DEBUG-I-VERIFYICF, exiting indirect command file "MAIN"
```

CALL

Description

Calls the specified procedure. You can specify the procedure's name or virtual address. If the procedure requires an argument list, you must specify it in the CALL command.

The debugger assumes that the called procedure conforms to the VAX-11 procedure calling standard (see the VAX-11 Architecture Handbook). You can call a procedure and debug it independently of the rest of the program. When the debugger encounters a RET instruction in the procedure, it displays the values returned by the procedure (value returned in R0). The debugger then restores the values stored in the registers before the CALL command.

Format

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

Command Parameters

name

Specifies the name or the virtual address of the procedure to be called. The virtual address can only be expressed as a number; it cannot be an address expression.

argument-list

The arguments required by the procedure.

Command Qualifiers

None.

Examples

```
DBG> CALL SUB1(X)
routine start at SUB1
value returned is 7FFF07DC
```

CANCEL

Description

Cancels breakpoints, tracepoints, and watchpoints, and restores scope and user-set entry/display modes and types to their default values. The item canceled depends on the keyword specified in the command.

See the individual command descriptions following for more information.

Format

CANCEL keyword [/qualifier] [parameters]

Command Qualifiers

Depends on keyword

Command Parameters

keyword

Specifies the item to be canceled. Keyword can be ALL, BREAK, EXCEPTION BREAK, MODE, MODULE, SCOPE, TRACE, TYPE, or WATCH.

parameters

Depends on the keyword specified.

Command Qualifiers

Depends on the keyword specified.

CANCEL ALL

Description

Cancels all breakpoints, tracepoints, watchpoints, and restores scope and user-set entry/display modes and types to their default values.

CANCEL ALL does not affect the modules included in the debugger symbol table or the current language.

Format

CANCEL ALL

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>CAN ALL
```

CANCEL BREAK

Description

Cancels specified breakpoint or all breakpoints.

You must either include an address-expression or the /ALL qualfier in the CANCEL BREAK command.

You can also cancel all breakpoints with the CANCEL ALL command.

Format

CANCEL BREAK [/qualifier] [address-expression]

Command Qualifiers

/ALL

Command Parameters

address-expression

Specifies the address of the breakpoint to be canceled.

Command Qualifiers

/ALL

Specifies that all breakpoints are to be canceled.

Examples

```
DBG>CAN BRE MAIN\LOOP+10
DBG>CAN BRE/ALL
```

CANCEL EXCEPTION BREAK

Description

Cancels the request that the debugger treat external exception conditions as breakpoints.

When the debugger encounters an external exception condition, it executes your program's condition handler if it exists. Otherwise, the debugger allows VAX/VMS to handle the exception.

CANCEL EXCEPTION BREAK cancels the effects of SET EXCEPTION BREAK and causes the debugger to respond to an exception condition either by executing your program's condition handler or by returning control to VAX/VMS.

Format

```
CANCEL EXCEPTION BREAK
```

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>CAN EXC BRE
```

CANCEL MODE

Description

Cancels all modes and types and sets them to the default values for the current language.

The CANCEL MODE command sets the default data type to long integer, the override data type to none, and address display mode to symbolic. It sets the radix mode to the default for the current language. For VAX-11 MACRO, its sets the default radix to hexadecimal.

Format

CANCEL MODE

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>CANCEL MODE
```

CANCEL MODULE

Description

Deletes symbols of specified modules or all modules from the debugger symbol table. The CANCEL MODULE command is used when the symbol table is full and it is necessary to delete some modules in order to add others.

You can delete one module, a list of modules, or all modules. The minimum abbreviation of MODULE is MODU.

Format

CANCEL MODULE [/qualifier] module [,module ...]

Command Qualifiers

/ALL

Command Parameters

module

Specifies the name of the module to be deleted from the symbol table.

Command Qualifiers

/ALL

Specifies that all modules are to be deleted from symbol table.

Examples

```
DBG>CAN MODU SUB1
DBG>CAN MODU/ALL
```

CANCEL SCOPE

Description

Resets scope to its default value (PC scoping).

The CANCEL SCOPE command deletes any scope search list that you have specified with SET SCOPE. After you have entered CANCEL SCOPE, the debugger searches for symbols in the scope that contains the current PC. CANCEL SCOPE is equivalent to SET SCOPE 0.

Format

CANCEL SCOPE

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>CAN SCO
```

CANCEL TRACE

Description

Cancels a tracepoint set at a specified address, opcode tracing at branch instructions, opcode tracing at call instructions, or all tracepoints and all opcode tracing.

If you specify an address-expression, the tracepoint at that address is cancelled. If you specify /BRANCH or /CALL, opcode tracing on branch instructions or call instructions is canceled. If you specify /ALL, all tracepoints and opcode tracing are canceled.

You can also cancel all tracepoints with the CANCEL ALL command.

Format

CANCEL TRACE [/qualifier] [address-expression]

Command Qualifiers

/ALL
/BRANCH
/CALL

Command Parameters

address-expression

Specifies the address of the tracepoint to be canceled.

Command Qualifiers

/ALL

Cancels all tracepoints and all opcode tracing.

/BRANCH

Cancels opcode tracing on branch instructions.

/CALL

Cancels opcode tracing on call instructions.

Examples

```
DBG>CAN TRA MAIN\LOOP+10
DBG>CAN TRA/CALL
DBG>CAN TRA/ALL
```

CANCEL TYPE/OVERRIDE

Description

Cancels the current override type and sets the override type to "none".

After you enter CANCEL TYPE/OVERRIDE, symbols are displayed in their compiler-specified data type (if they have one) unless you specify a type qualifier in the EXAMINE, EVALUATE, or DEPOSIT command. Symbols that do not have a compiler-specified data type are displayed in the default data type specified in the last SET TYPE command. The CANCEL TYPE command must be specified with the /OVERRIDE command qualifier.

Format

```
CANCEL TYPE/OVERRIDE
```

Command Parameters

None.

Command Qualifiers

/OVERRIDE

Must be specified. The minimum abbreviation is /OVERR.

Examples

```
DBG>CAN TYPE/OVERR
```

CANCEL WATCH

Description

Cancels watchpoint set at specified address or cancels all watchpoints.

If you specify an address-expression, the watchpoint at that address is cancelled. If you specify /ALL, all watchpoints are canceled.

You can also cancel all watchpoints with the CANCEL ALL command.

Format

CANCEL WATCH [/qualifier] [address-expression]

Command Qualifiers

/ALL

Command Parameters

address-expression

Specifies the address of the watchpoint to be canceled.

Command Qualifiers

/ALL

Specifies that all watchpoints should be canceled.

Examples

```
DBG>CAN WAT SUB2\D
DBG>CAN WAT/ALL
```

CTRL/C, CTRL/Y, CTRL/Z

Description

The CTRL/C and CTRL/Y key commands interrupt execution of the program and return control to VAX/VMS. The CTRL/Z key command is equivalent to the EXIT command.

After you interrupt execution of your program with CTRL/C or CTRL/Y, you can return control to the debugger by entering the DEBUG command. This is useful if you have inhibited initiation of the debugger with RUN/NODEBUG command or if you have entered the GO command and your program is executing an infinite loop that does not contain a breakpoint. If you enter any DCL command other than DEBUG or CONTINUE after you have typed CTRL/C or CTRL/Y, the program you are debugging is terminated.

You can type the CTRL/Z command at your terminal instead of entering the EXIT command.

The CTRL/C and CTRL/Y key commands are echoed on the terminal as ^Y. The CTRL/Z key command is echoed as ^Z.

Format

```
<CTRL/C>
<CTRL/Y>
<CTRL/Z>
```

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>
^Y

$ DEBUG
DBG>^Z
$
```

DEFINE

Description

Defines symbol(s) and assigns them specified address(es). You can use these symbols to reference an address in the image.

The DEFINE command lets you define symbols during a debugging session. You can, for example, create explicit symbols for unlabeled locations or for locations with long pathnames. The debugger always searches symbols defined in the debugging session first. Consequently, symbols you define with the debugger have precedence over symbols in your program with the same name. To refer to the symbol in your program in a subsequent command, you must specify the pathname with the symbol.

Once you have defined a symbol, you cannot explicitly cancel its definition, but you can redefine it by specifying a different value with the DEFINE command.

You cannot define a pathname, that is, scope\symbol, but can only define symbol names.

You can use the EVALUATE command to find out the current value of a symbol.

Format

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

Command Parameters

symbol

Specifies the name of the symbol to be defined. The following rules must be followed when defining a symbol name:

- Symbols can be composed of alphanumeric characters (A-Z and 0-9), underscores (_), dollar signs ($), and periods (.). Periods cannot be used in some languages. The debugger interprets lowercase alphabetics to be uppercase. Thus "LOOP" and "loop" are the same to the debugger.
- The first character must not be a number (0-9) or a period (.).
- The symbol must be no more than 31 characters long.

In addition, by DIGITAL convention:

- The dollar sign is reserved for names defined by DIGITAL. This convention ensures that a user-defined name will not conflict with a DIGITAL-defined name.
- The period (.) should not be used because some languages, such as FORTRAN, do not allow periods in symbol names.

expression

Either a virtual address or a symbolic address. The expression must have a value between 1 and 0FFFFFFFF (hexadecimal). Any numbers in the expression are interpreted in the current radix mode. You can use radix operators to override the current radix mode.

Command Qualifiers

None.

Examples

```
DBG>DEF CHK=MAIN\LOOP+10
DBG>DEF OLDR5=@R5, VALUE =15+30A
```

DEPOSIT

Description

Stores the specified value(s) at the specified location. The DEPOSIT command lets you alter the contents of memory locations and registers. If you enter more than one value, the values are deposited at sequential memory locations starting with the specified location. Data values can be in numeric, ASCII string, or instruction format.

Any data already existing in the memory locations you specify with the DEPOSIT command is lost. The size of the data being stored is dependent on the data type specified in a command qualifier, the compiler-specified data type, or the current default data type. The size of the data for each data type is:

Data Type	Size (in Bytes)
ASCII[:length]	length (defaults to 4)
Instruction	Variable dependent on the opcode and operands
Byte	1
Word	2
Long	4

Format

DEPOSIT [/qualifier] address-expression = data [,data...]

Command Qualifiers

/ASCII:length
/BYTE
/DECIMAL
/HEXADECIMAL
/INSTRUCTION
/LONG
/OCTAL
/WORD

Command Parameters

address-expression

Specifies the location in which to deposit the data. Any numbers in the address expression are interpreted in the radix mode specified in a qualifier or, if there is no radix qualifier, in the current radix mode. The address-expression can be one of the debugger's permanently defined symbols: R0 - R11, AP, FP, SP, PC, and PSL. The debugger will allow you to modify FP, SP, PC, or PSL, but a modification of these registers may cause a fatal error.

data

Specifies the data to be deposited. If more than one data item is specified, then the data is deposited in sequential locations starting at the specified address. The data is interpreted according to the radix mode qualifier and type qualifier specified in the command. If no radix mode qualifier or type qualifier is specified, then the data is interpreted according to the current radix mode and type. If the data is ASCII string data or INSTRUCTION data, then each data item must be enclosed in quotation marks or apostrophes.

Command Qualifiers

/ASCII:length

Specifies that the data is ASCII strings enclosed in quotation marks or apostrophes. The length specifies the number of bytes of ASCII data to be deposited for each data item. It is interpreted in the radix specified in a command qualifier that precedes the /ASCII qualifier or in the current radix mode. If length is omitted, the debugger assumes a length of 4.

/BYTE

Specifies that the data type is byte integer. For each data item specified the debugger deposits one byte of data. The data is interpreted in the radix mode specified by a command qualifier or in the current radix mode. Any value that cannot be stored in a byte (greater than 255, unsigned) will be truncated.

/DECIMAL

Specifies that all numbers following the qualifier in the command are to be specified in decimal radix.

/HEXADECIMAL

Specifies that all numbers following the qualifier in the command are to be specified in hexadecimal radix.

/INSTRUCTION

Specifies that the data is VAX-11 MACRO instructions enclosed in quotation marks or apostrophes. The debugger deposits the binary opcode and operands. The length of the instruction deposited depends on the opcode and the addressing modes used.

/LONG

Specifies that the data type is longword integer. For each data item specified, the debugger deposits four bytes of data. The data is interpreted in the radix mode specified by a command qualifier or in the current radix mode.

/OCTAL

Specifies that all numbers following the qualifier in the command are in octal radix.

/WORD

Specifies that the data type is word integer. For each data item specified, the debugger deposits two bytes of data. The data is interpreted in the radix mode specified by a command qualifier or in the current radix mode. Any value that cannot be stored in a word (greater than 65,535, unsigned) will be truncated.

Examples

```
DBG>DEP/BYTE WORK=0,1,2,3,4,5,6,7
DBG>DEP/ASCII:10 WORK+20='abcdefghijklmnop'
DBG>SET TYPE INS
DBG>DEP SUB2+2 = 'MOVL   #20A,R0', 'MOVL   #3,R1'
DBG>E
SUB2\SUB2+0C:  RET
DBG>DEP .  = 'MULL3 R0,R1,R2' , 'ADDL2 R2,R0' , 'RET'
DBG>CALL SUB2
routine start at SUB2\SUB2
value returned is 00000828
```

EVALUATE

Description

Evaluates an expression and displays the value. Can be used as a calculator to display the value of any expression. The EVALUATE command can also evaluate bit fields (in VAX-11 MACRO).

Expressions are evaluated as longword integer expressions unless a symbol with a compiler-generated data type is specified in the expression. In this case, the expression is evaluated according to the rules of the current language. If a radix command qualifier is specified, all numbers in the command are interpreted in the specified radix and all values displayed by the command are displayed in that radix.

When the expression contains symbols, the evaluation of the expression depends on the current language. If the language is VAX-11 MACRO, the debugger considers the address of a symbol to be its value. To evaluate the data stored at that address, you must use the indirect operator (@). In some other languages, such as FORTRAN, COBOL, and BASIC, the debugger considers the contents of the address of a symbol to be its value. To evaluate the address of the symbol for these languages, you must use the /ADDRESS command qualifier.

Format

```
EVALUATE [/qualifier] expression [,expression ...]
EVALUATE [/qualifier] value<high-bit:low-bit>
```

Command Qualifiers

```
/ADDRESS (not in VAX-11 MACRO or VAX-11 BLISS)
/DECIMAL
/HEXADECIMAL
/OCTAL
```

Command Parameters

expression

Any legal expression in the current language.

value<high-bit:low-bit>

A value and a bit range to be evaluated. The value can be a number or an expression enclosed in angle brackets. The high-bit and low-bit values must be in the range of 0 through 31, for example, SYMB<31:16>. This feature is available only in VAX-11 MACRO.

Command Qualifiers

/ADDRESS

Specifies that the address of the symbol rather than its contents should be evaluated. /ADDRESS is not allowed in VAX-11 MACRO or VAX-11 BLISS.

/DECIMAL

Specifies that decimal is the default radix for numbers entered in the command and for the display of values.

/HEXADECIMAL

Specifies that hexadecimal is the default radix for numbers entered in the command and for the display of values.

/OCTAL

Specifies that octal is the default radix for numbers entered in the command and for the display of values.

Examples

```
DBG>DEP R0=WORK
DBG>DEP R1=10
DBG>EVAL @R0 + <2*@R1>
0000088F
DBG>EVAL @MAIN<8:0>
0000003C
DBG>EVAL @MAIN<4:3>
00000003
DBG>SET LANG FORTRAN
DBG>EVAL 100.34 * ( 14.2 + 7.9)
   2217.514
DBG>EVAL/ADDR X
1512
```

EXAMINE

Description

Displays the current contents of specified addresses or ranges of addresses. The contents of the addresses can be displayed in numeric form, in ASCII string form, or decoded to VAX-11 MACRO instruction form.

You can use the EXAMINE command to display a single location, multiple locations, a range of contiguous locations, or any combinations of these. The length of each data item displayed and its format depends on the type and mode command qualifiers or the current type and modes.

The debugger displays the address being examined, a colon, and then the contents of the address. The address is displayed symbolically, if possible, if the /SYMBOLIC qualfier is specified or if the display mode is set to SYMBOLIC. If the /NOSYMBOLIC qualifier is specified, the address is displayed as an absolute virtual address.

An EXAMINE command with no address-expressions specification displays the locations after the last location examined or deposited.

Format

EXAMINE [/qualifier] [addr-expr[:addr-expr] [,addr-expr[:addr-expr]...

Command Qualifiers

/ASCII:length
/BYTE
/DECIMAL
/HEXADECIMAL
/INSTRUCTION
/LONG
/NOSYMBOLIC
/OCTAL
/SYMBOLIC
/WORD

Command Parameters

addr-expr

Specifies the address to be examined. If a range of addresses is specified with a colon, the first address must be less than the second address. If a list of address expressions is specified, they can be in any order.

Command Qualifiers

/ASCII:length

Specifies that contents of memory should be displayed as ASCII strings. The length specifies the number of bytes of memory to be examined and the number of characters to be displayed. It is interpreted in the radix specified in a command qualifier that precedes the /ASCII qualifier or in the current radix. If length is omitted, the debugger assumes a length of 4. The number of characters actually displayed is limited by the maximum size of the debugger's output line, 132 characters.

/BYTE

Specifies that the data type is byte integer. The contents of each byte of memory is displayed in the radix mode specified in a command qualifier or in the current radix mode.

/DECIMAL

Specifies that the radix is decimal. The debugger displays all numbers and interprets all numbers in the command in decimal radix.

/HEXADECIMAL

Specifies that the radix is hexadecimal. The debugger displays all numbers and interprets all numbers in the command in hexadecimal radix.

/INSTRUCTION

Specifies that the contents of memory should be displayed as VAX-11 MACRO instruction. The debugger attempts to decode the address specified as an instruction. If the debugger can decode the instruction, it displays the instructions. If the debugger cannot decode the instruction, it displays a warning message. The length of the instruction displayed is variable depending on the opcode and addressing modes.

/LONG

Specifies that the data type is longword integer. The contents of each longword (4 bytes) of memory is displayed in the radix mode specified in a command qualifier or in the current radix mode.

/NOSYMBOLIC

Specifies that addresses should be displayed as absolute virtual addresses. The debugger displays the addresses being examined and any addresses in decoded instructions as absolute virtual addresses.

/OCTAL

Specifies that the radix is octal. The debugger displays all numbers and interprets all numbers in the command in octal radix.

/SYMBOLIC

Specifies that addresses should be displayed as symbols or offsets from symbols. The debugger displays the addresses being examined and any addresses in decoded instructions as symbols or offsets from symbols.

/WORD

Specifies that the data type is word integer. The contents of each word (2 bytes) of memory is displayed in the radix mode specified in a command qualifier or in the current radix mode.

Examples

```
DBG>EX/ASC WORK+20
DETAT\WORK+20:  abcd

DBG>E/WORD MAIN
MAIN\MAIN:  483C
DBG>E/I
MAIN\MAIN+02:  MOVAL   L^MAIN\A,R11

DBG>E/DEC/WORD WORK:WORK+10 , SUB1\D
DETAT\WORKDATA:  256
DETAT\WORKDATA+2:  770
DETAT\WORKDATA+4:  1284
DETAT\WORKDATA+6:  1798
DETAT\WORKDATA+8:  -1
DETAT\WORKDATA+10:  -19729
SUB1\D:  31

DBG>E/NOSYM WORKDATA
0000086F:  03020100
```

EXIT

Description

Ends the debugging session or specifies the end of the command procedure.

When you enter the EXIT command at the terminal, the debugger ends the session and returns control to VAX/VMS. You cannot reenter the debugger with either the DEBUG or CONTINUE commands but must reinitiate the debugger by running a program.

When the debugger executes an EXIT command in a command procedure, the debugger returns control to the command stream that invoked the command procedure. Note that if the command procedure was invoked from a DO command sequence in a SET BREAK command, the debugger gets the next command from that DO command sequence.

When the debugger executes an EXIT command in a DO command sequence, It ignores any following commands in the DO command sequence.

Format

EXIT

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>EXIT
$
```

GO

Description

Starts or continues program execution. The first GO command with no address specified starts the program at its initial (transfer) address. Succeeding GO commands continue execution from the point where execution was suspended or at the specified address.

A GO command with an address expression replaces the current contents of the program counter (PC) with the specified address and then executes the instruction at that address. Your program's execution can be unpredictable when you initiate execution at an address other than your transfer address or when you restart execution at the transfer address or at another address.

Note that if you enter a GO command with no address expression after your program has completed execution, the debugger tries to execute your program with a value of 0 in the PC and displays an error message.

Format

GO [address-expression]

Command Parameters

address-expression

Specifies the address to be executed. Execution starts or continues at the specified address.

Command Qualifiers

None.

Examples

```
DBG>GO
routine start at MAIN\MAIN
routine break at SUB1\SUB1
DBG>GO
routine start at SUB1\SUB1
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>GO MAIN
routine start at MAIN\MAIN
routine break at SUB1\SUB1
```

HELP

Description

Displays a description of the command specified. You can also get descriptions of command parameters and qualifiers.

The HELP command displays a description of debugger commands, their format, and the parameters and qualfiers you can use with them. You can find out the topics that have help descriptions by entering the HELP command with no topics.

Format

HELP topic [subtopic ...]

Command Parameters

topic

Specifies the command that you want information about.

subtopic

Specifies the command keyword, qualifier, or parameter that you want information about. If you want information about a specific qualifier, specify the qualifier (including the initial slash) as the subtopic. If you want information about all qualifiers or all parameters, specify QUALIFIER or PARAMETER, respectively. If you want all the information about a command, specify an asterisk (*) as the subtopic.

Command Qualifiers

None.

Examples

```
DBG>HELP DEFINE

  DEFINE

       Defines symbol(s) and assigns them specified addresses(s).
       You can use these symbols to reference an address in the
       image.

  Format:

       DEFINE symbol=expression [,symbol=expression ...]

  Additional information available:

  Parameters
```

SET

Description

Establishes breakpoints, tracepoints, or watchpoints or sets the language, modules, step conditions, and default modes and types. The item set depends on the keyword specified.

See the individual command descriptions following for more information.

Format

SET keyword [/qualifier] parameter

Command Qualifiers

Depends on keyword.

Command Parameters

keyword

Specifies the item to be set. Keyword can be BREAK, EXCEPTION BREAK, LANGUAGE, LOG, MODE, MODULE, OUTPUT, SCOPE, STEP, TRACE, TYPE, or WATCH.

parameters

Depends on the keyword specified.

Command Qualifiers

Depends on the keyword specified.

SET BREAK

Description

Establishes a breakpoint at a specified address. The debugger stops execution of the program before the instruction at that address is executed and prompts you for a command. You can optionally specify a list of debugger commands to be executed at the breakpoint.

SET BREAK command establishes breakpoints. You can optionally establish a DO command sequence that is executed when the breakpoint is reached. Breakpoints with DO command sequences allow you to examine a series of locations, call a procedure to evaluate results, or execute a command procedure.

If you do not specify the /AFTER qualifier, the debugger stops each time the breakpoint is reached and executes the DO command sequence (if there is one). The /AFTER qualifier allows you to specify the number of times that you want the breakpoint to be reached before the debugger stops execution.

Format

SET BREAK [/qualifier] address-expression [DO (cmd[;cmd...])

Command Qualifiers

/AFTER:count

Command Parameters

address-expression

Specifies the address of the breakpoint. The address must be at the beginning of an instruction.

cmd

Any debugger command. If you specify more than one command, you must separate them with semicolons. The debugger executes the commands that you specify when the breakpoint is reached.

Command Qualifiers

/AFTER:count

Specifies that the debugger should not stop execution of the program until the breakpoint is reached the number of times specified by count. Each time the breakpoint is reached the debugger decrements the value specified in count. When the value reaches 1, the debugger stops the program, executes any DO commands, and prompts you for a command. The debugger will continue to stop execution each time the breakpoint is reached until you cancel it. If count has a value of 0, the debugger stops execution of the program the first time the breakpoint is reached and then automatically cancels the breakpoint. The debugger interprets count as a decimal number even if the current radix mode is not decimal.

Examples

```
DBG>SET BREAK/AFTER:3 SUB2
DBG>SET BREAK MAIN+1F DO (EX SUB1\D; EVAL @R+3; GO)
DBG>SET BREAK SUB1\LOOP
```

SET EXCEPTION BREAK

Description

Requests that the debugger treat external exception conditions as a breakpoint.

Format

SET EXCEPTION BREAK

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>SET EXC BRE
```

SET LANGUAGE

Description

Specifies the source language to the debugger. The debugger lets you enter symbols, expressions, and addresses in terms that are compatible with the source language that you specify.

The debugger initially sets the language to the source language used in the first module specified in the LINK command. The SET LANGUAGE command changes the current language. The language affects how the debugger interprets and displays expressions, addresses, radixes, step conditions, and special symbols. The debugger accepts expressions that are valid in the source language with some restrictions. The debugger accepts and displays addresses as offsets from line numbers and labels accepted in the source language. In FORTRAN, the debugger interprets %LINE 10 as the address of the source line 10 and interprets %LABEL 20 as the address of the FORTRAN label 20, and in BASIC, the debugger interprets %LINE 500.3 as the address of the third statement on line 500.

Format

SET LANGUAGE language-name

Command Parameters

language-name

Specifies the name of the language. Valid languages include the following: BASIC, BLISS, COBOL, FORTRAN, and MACRO.

Command Qualifiers

None.

Examples

```
DBG>SET LANG COBOL
```

SET LOG

Description

Specifies the file specification of the log file that the debugger uses when the output is directed to the log file.

The SET LOG command controls only the name of the log file; it does not control whether a log file is being written. The SET OUTPUT command does this. By default, the debugger uses the file name DEBUG and the file type LOG. If the debugger is currently creating a log file and you enter SET LOG, the debugger closes the existing log file and opens the newly specified file.

If you specify a version number in the file specification and that version of the file exists, the debugger appends the log of the debugging session to that existing file.

Format

```
SET LOG file-spec
```

Command Parameters

file-spec

Specifies the file specification for the log file. You must enclose the file specification in quotation marks or apostrophes only if the file specification begins with a special symbol (such as a square bracket).

Command Qualifier

None.

Examples

```
DBG>SET LOG CALC
DBG>SET LOG "[CODEPROJ]FEB29.TMP"
```

SET MODE

Description

Sets the default entry and display modes. The entry and display modes control the current radix and whether addresses are displayed symbolically or as absolute virtual addresses. The default radix controls how the debugger interprets numbers that you enter and how it displays numbers.

You can override the current radix mode by using a radix mode qualifier in a DEPOSIT, EVALUATE, or EXAMINE command or by specifying a radix control operator (^D, ^O, and ^X in VAX-11 MACRO).

In SYMBOLIC mode, addresses are displayed as symbols or offsets from symbolic locations. The debugger first looks for an exact match with a symbol created with the DEFINE command, a local symbol, or a global symbol. If no exact match can be found, the debugger searches for the symbol whose value is less than the address and closest to it. If it cannot find any symbolic definition, it displays the address as a virtual address in the current radix mode. Probable causes of the debugger displaying a virtual address are that the module's symbols are not included in the debugger symbol table or that the address is outside the program's address space.

In NOSYMBOLIC mode, addresses are displayed as virtual addresses in the current radix mode. Note that you can always enter symbolic addresses even when the mode is NOSYMBOLIC.

Format

SET MODE mode-keyword [,mode-keyword]

Command Parameters

mode-keyword

Specifies the entry and display mode. Mode-keyword can be DECIMAL, HEXADECIMAL, OCTAL, NOSYMBOLIC, or SYMBOLIC.

- DECIMAL -- Sets the current radix to decimal.
- HEXADECIMAL -- Sets the current radix to hexadecimal.
- OCTAL -- Sets the current radix to octal.
- NOSYMBOLIC -- Specifies that addresses should be displayed as absolute virtual addresses.
- SYMBOLIC -- Specifies that addresses should be displayed as offsets from symbolic locations.

Command Qualifier

None.

Examples

```
DBG>SET MODE DEC, NOSYM
```

SET MODULE

Description

Adds the symbols of the specified modules or of all modules to the debugger symbol table. The debugger cannot access any local symbols unless they are in its symbol table.

When the debugger is initiated, the symbol table contains the symbols in the first module specified in the LINK command. For images with less than 2,000 (approximately) symbols, you can add all the modules to the symbol table with SET MODULE/ALL. For images with more symbols you must add the symbols for the modules that you need to debug. If you want to add a module and there is no room in the symbol table, you must use the CANCEL MODULE command to delete some modules from the symbol table. Note that when you enter the SET SCOPE command with the name of a scope, the module that contains that scope is added to the symbol table if it fits.

The minimum abbreviation of MODULE is MODU.

Format

SET MODULE [/qualifier] [module-name [,module-name] ...]

Command Qualifiers

/ALL

Command Parameters

module-name

Specifies the name of the module to be added to the debugger symbol table.

Command Qualifiers

/ALL

Specifies that all modules in the image should be added to the symbol table.

Examples

```
DBG>SET MODU SUB1
DBG>SET MODU/ALL
```

SET OUTPUT

Description

Controls whether the debugger displays output on the terminal or writes it to a log file and controls whether the debugger echoes commands in command procedures. The SET OUTPUT command controls whether a log file is being created; SET LOG command controls the file specification of the log file.

When the debugger is initiated, all debugger responses are displayed on the terminal, no log file is created, and command procedures and DO command sequences are not echoed on the terminal.

The log file contains all the commands that you enter at a terminal and all the responses of the debugger. The log file does not contain the DBG> prompt that the debugger displays. The debugger's responses are preceded with an exclamation mark. If you wish to reproduce a debugging session, you can use the log file as a command procedure.

Format

```
SET OUTPUT option [,option ...]
```

Command Parameters

option

Specifies the mode of output. Option can be LOG, TERMINAL, or VERIFY or the negative form of each.

- LOG -- Starts writing output to the log file. The log file contains all the commands that you enter at the terminal and all debugger responses. The debugger responses are preceded by an exclamation mark (comment indicator) to allow you to use the log file as a command procedure. NOLOG, the initial setting, inhibits output to the log file.

- TERMINAL -- Starts writing output to the terminal. TERMINAL, the initial setting, causes all debugger responses to be displayed on the terminal. NOTERMINAL causes the debugger to stop displaying all responses except error messages on the terminal. The NOTERMINAL parameter is useful when you want to write output only to a log file. If you specify SET OUTPUT NOLOG, NOTERMINAL, the debugger displays a warning message telling you that output is being lost.

- VERIFY -- Causes the debugger to echo commands executed from command procedures and DO sections of SET BREAK commands. The commands are displayed on the terminal and written to the log file depending on the other options specified for the SET OUTPUT command. NOVERIFY, the initial setting, causes the debugger to not echo commands as they are executed in a command procedure or DO command sequence.

Command Qualifiers

None.

Examples

```
DBG>SET OUT VER,LOG,NOTERM
```

SET SCOPE

Description

Specifies scopes to be searched to find a symbol. By default, the debugger searches for symbols (that are specified without pathnames) in the current scope (the scope that contains the current PC). If the debugger does not find the symbol, it searches its symbol table for a unique symbol. When you enter SET SCOPE, the debugger modifies its search rules: it searches the scopes in the order you specify. If it does not find the symbol in these scopes, it then searches for a unique symbol.

SET SCOPE command allows you to modify the default symbol search. You can specify the default scope (scope 0) in any position in the scope list. You can also specify that global symbols be searched in any position in the scope list by the symbol backslash (\).

If the debugger cannot find a symbol in the scopes specified with the SET SCOPE command, it searches all the modules in the symbol table for a unique symbol. There is no way to suppress the search for a unique symbol.

The debugger always searches the symbols defined with the DEFINE command first.

Format

```
SET SCOPE scope [,scope ...]
```

Command Parameters

scope

Specifies the name of a scope, the digit 0, a backslash (\), or the number of a scope. These scopes have the following meanings:

- Name of Scope -- In general, consists of a module name and block or routine names separated by backslashes. The simplest case (and the only name of scope valid in VAX-11 MACRO) is a scope consisting of a module name. When you specify a name of scope, the debugger adds the symbols for the module specified to the symbol table if they are not already included.

- 0 -- Specifies the current scope, the scope that contains the current PC. The current scope is used as the default scope if you enter the CANCEL SCOPE command. The current scope changes as different sections of your program are executed. The digit 0 is a special case of the number of the scope.

- \ -- Specifies the scope consisting of all the global symbols defined in your image.

- Number of Scope -- Specifies scope by the level of active calls. The number 0 represents the scope currently being executed; the number 1 represents the scope that contained the call to the current scope, the number 2 represents the scope that contained the call before that.

Command Qualifiers

None.

Examples

```
DBG>SET SCOPE MAIN,SUB1,0,\
```

SET STEP

Description

Specifies the current default step conditions. The SET STEP command controls the unit of the program to be executed (INSTRUCTION or LINE), whether a called procedure or a subroutine is treated as a series of instructions or as a single instruction (INTO or OVER), and whether the STEP command will stop in system space (SYSTEM, or NOSYSTEM). You can always override the current default step condition by specifying a mode qualifier in the STEP command.

The initial STEP conditions are dependent on the current language. When you enter the SET LANGUAGE command the step conditions are set to the default for the specified language. For VAX-11 MACRO, the initial step conditions are INSTRUCTION, OVER, and NOSYSTEM.

Format

```
SET STEP condition [,condition ...]
```

Command Parameters

condition

Specifies the default step condition. If you specify a list of step conditions, you can include one each from the following pairs: INSTRUCTION and LINE, INTO and OVER, and SYSTEM and NOSYSTEM.

- INSTRUCTION -- Steps in increments of instructions (the only valid increment for VAX-11 MACRO).
- LINE -- Steps in increments of lines for line-oriented languages, such as FORTRAN and BASIC (ignored for VAX-11 MACRO).
- INTO -- Steps into a routine entered by a call or branch to subroutine instruction (CALLG, CALLS, BSBB, BSBW, and JSB).
- OVER -- Treats a routine entered by a call or branch to subroutine instruction as part of the call or branch instruction. It is useful to step over a routine when you are debugging one procedure and are not interested in debugging any procedures that it calls.
- SYSTEM -- Treats instructions in system space the same as other instructions. SYSTEM is useful when you want to step through a system service routine. Note this condition allows you to step through code in system space that is executing in user mode.
- NOSYSTEM -- Ignores all instructions in system space. System space includes the procedures called to perform system services and input/output. The debugger does not count any instructions or lines executed in system space.

Command Qualifiers

None.

Examples

```
DBG>SET STEP INS,INTO
```

SET TRACE

Description

Establishes a tracepoint at a specified address or establishes opcode tracing. The debugger temporarily interrupts execution of a program before the instruction at that address is executed. It displays the address of the tracepoint and then continues execution of the program. Opcode tracing is when the debugger temporarily interrupts execution of a program whenever the specified type of instruction is about to be executed.

SET TRACE/BRANCH and SET TRACE/CALL cause the debugger to interrupt your program before every instruction is executed to check if the instruction should be traced.

Format

SET TRACE [/qualifier] [address-expression]

Command Qualifiers

/BRANCH
/CALL

Command Parameters

address-expression

Specifies the address of the tracepoint. The address must be at the beginning of an instruction.

Command Qualifiers

/BRANCH

Trace all branch, jump, and case instructions in the image (see the example for the list of branch, jump, and case instructions).

/CALL

Trace all instructions that call routines in the image (BSBB, BSBW, CALLG, CALLS, JSB, RET, and RSB instructions).

Examples

```
DBG>SET TRACE MAIN+1F
DGB>SET TRACE SUB1/LOOP+09
DBG>SE TR/BRA
DBG>SE TR/CA
DBG>SHOW TRACE
tracepoint at SUB1/LOOP+09
tracepoint at MAIN/MAIN+1F
tracing  /CALL instructions:  CALLS, CALLG, BSBW, BSBB, JSB,  RSB
         and RET
tracing  /BRANCH instructions:  BNEQ,  BEQL,  BGTR,  BLEQ,  BGEQ,
         BLSS,  BGTRU,  BLEQU,  BVC, BVS, BGEQU, BLSSU, BRB, BRW,
         JMP, BBS, BBC, BBSS, BBCS,  BBSC,  BBCC,  BBSSI,  BBCCI,
         BLBS,  BLBC,  ACBB,  ACBW,  ACBL,  ACBF,  ACBD,  AOBLEQ,
         AOBLSS, SOBGEQ, SOBGTR, CASEB, CASEW and CASEL
```

SET TYPE

Description

Sets the default types for the DEPOSIT and EXAMINE commands. The SET TYPE command sets the default type when there is no assigned data type in the symbol table. The SET TYPE/OVERRIDE command sets the default type for all data items in DEPOSIT and EXAMINE commands.

VAX-11 MACRO does not provide any symbol type information to the debugger. Consequently, you can use either SET TYPE or SET TYPE/OVERRIDE for modules written in VAX-11 MACRO.

Format

SET TYPE [/qualifier] type-keyword

Command Qualifiers

/OVERRIDE

Command Parameters

type-keyword

Specifies the default data type. Type-keyword can be ASCII:length, BYTE, INSTRUCTION, LONG, or WORD.

- ASCII:length -- Specifies that the default data type is an ASCII string. The number of characters in the string is specified by length. If you do not specify length, the debugger assumes a length of 4.
- BYTE -- Specifies that the default data type is byte integer.
- INSTRUCTION -- Specifies that the default data type is instruction.
- LONG -- Specifies that the default data type is longword integer.
- WORD -- Specifies that the default data type is word integer.

Command Qualifiers

/OVERRIDE

Specifies that the specified data type should be used even when a symbol has an assigned data type in the symbol table. These data types are assigned by language compilers. The CANCEL TYPE/OVERRIDE command cancels the effects of SET TYPE/OVERRIDE and specifies that the debugger should use the symbols' assigned data types. The minimum abbreviation is /OVERR.

Examples

```
DBG>SET TYP ASC:8
DBG>SET TYP/OVERR LONG
```

SET WATCH

Description

Establishes a watchpoint at the specified address. The debugger stops execution of the program whenever an instruction writes to the data to the specified address. The debugger then prompts you for a command.

The SET WATCH command instructs the debugger to stop execution of the program if any of a series of addresses is written to. The debugger usually watches the four bytes beginning at the specified address. However, if the address has an assigned data type in the symbol table, the debugger watches the number of bytes associated with that data type.

Whenever an instruction attempts to modify a location on the same page as the address specified with the SET WATCH command, the debugger gets control. The debugger allows the instruction to execute. If the watchpoint is modified, the debugger displays the old and new values and then prompts for a command.

You cannot set a watch point on a dynamically allocated variable such as a BASIC variable or array element or a FORTRAN dummy argument.

Format

SET WATCH address-expression

Command Parameters

address-expression

Specifies the address of the watchpoint.

Command Qualifiers

None.

Examples

```
DBG>SET WAT SUB2\TABLE+20
```

SHOW

Description

Displays the current breakpoints, calls, language, log file, entry and display modes, modules, output setting, scope, step conditions, tracepoints, data types, and watchpoints. The item that is displayed depends on the keyword specified.

See the individual command descriptions following for more information.

Format

SHOW keyword [/qualifier] [parameter]

Command Qualfiers

Depends on keyword.

Command Parameters

keyword

Specifies the item to be displayed. Keyword can be BREAK, CALLS, LANGUAGE, LOG, MODE, MODULE, OUTPUT, SCOPE, STEP, TRACE, TYPE, and WATCH.

parameter

Depends on the keyword specified.

Command Qualifiers

Depends on the keyword specified.

SHOW BREAK

Description

Displays the locations of current breakpoints and any /AFTER qualifier or DO command sequences associated with them.

If you entered a breakpoint with an /AFTER qualifier, the debugger decrements the value specified each time the breakpoint is reached. The SHOW BREAK command displays the current value, that is the number of times the breakpoint must be reached before the breakpoint is executed. The value is always displayed as a decimal number even if the current radix mode is not decimal.

Format

SHOW BREAK

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>SHOW BREAK
breakpoint at SUB1\LOOP
breakpoint at MAIN\MAIN+1F do (EX SUB1\D ; EVAL @R3+4 ; GO)
routine breakpoint /after:2 at SUB2\SUB2
```

SHOW CALLS

Description

Displays the current location in the program and the preceding calls that were executed. The SHOW CALLS command is useful in determining how your program reached the current position.

The SHOW CALLS command displays the same information as the traceback display. The traceback display appears after error messages when you have not initiated the debugger.

Format

SHOW CALLS [integer]

Command Parameters

integer

Causes the debugger to display the number of preceding calls specified by integer. If you omit integer, the debugger displays all preceding calls. Integer is always interpreted as a decimal number even if the current radix mode is not decimal.

Command Qualifiers

None.

Examples

```
DBG>SHOW CALLS
module name     routine name      line    relative PC  absolute PC
SUB2            SUB2                      00000002     0000085A
                SUB1                 5    00000014     00000854
                MAIN                10    0000002C     0000082C
```

SHOW LANGUAGE

Description

Displays the current language.

Format

SHOW LANGUAGE

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>SH LANG
language:  MACRO
```

SHOW LOG

Description

Displays the name of the log file and shows whether the debugger is writing to the log file.

Format

SHOW LOG

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>SH LOG
not logging to DEBUG.LOG
```

SHOW MODE

Description

Displays the current default entry and display modes and the default types.

The SHOW MODE command displays the current default modes, current type, and override type. If the current type or override type is ASCII:length, the debugger displays length as a decimal number even if the current radix mode is not decimal.

Format

SHOW MODE

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>SH MOD
modes: symbolic, hexadecimal
type: long integer
type/override: none
```

SHOW MODULE

Description

Lists the modules in the image and shows which modules have symbols in the debugger symbol table.

The SHOW MODULE command lists the program modules by name, indicates whether or not their symbols are currently included in the debugger symbol table, and lists the amount of space required to include the symbols in the table.

By default, the debugger includes the symbols in the first module specified in the link command. You can add modules with the SET MODULE command and you can delete modules from the symbol table with the CANCEL MODULE command.

The debugger lists each module by name. It lists a "yes" if the module's local symbols are included in the symbol table, a "no" if they are not. If the modules are not all written in the same language, the debugger lists the language each module is written in.

If a module is not currently included in the symbol table, the debugger lists the maximum amount of space that the symbols in the module can require in the symbol table. Once a module is included in the symbol table, the debugger lists the actual amount of memory needed to include the symbols. The debugger also lists the total number of modules and the amount of free space remaining in the symbol table.

The debugger has no knowledge of any modules that it does not list.

The minimum abbreviation of MODULE is MODU.

Format

SHOW MODULE

Command Parameters

None.

Command Qualifier

None.

Examples

```
DBG> SHOW MODU
module name       symbols    language     size
FOO               yes        MACRO        432
MAIN              no         FORTRAN      280
SUB1              no         FORTRAN      164
SUB2              no         FORTRAN      204
total modules:  4.           remaining size:  60720.
```

SHOW OUTPUT

Description

Displays the current setting of the debugger's output configuration. Reports whether the debugger is displaying output on the terminal, writing output to a log file, verifying command procedures, and verifying DO command sequences.

Format

SHOW OUTPUT

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>SH OUT
output:  noverify, terminal, not logging to DEBUG.LOG
```

SHOW SCOPE

Description

Displays the list of default scopes currently used by the debugger to resolve symbol references. If you have included a 0 (current scope) or a number in the SET SCOPE command, the debugger also displays, if possible, the name of the scope currently associated with the number.

Format

SHOW SCOPE

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>SH SCO
scope:  MAIN, SUB1, 0[=SUB2],\
```

SHOW STEP

Description

Displays the current default step conditions.

Format

SHOW STEP

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>SH STEP
step type:  no system, by instruction, over routine calls
```

SHOW TRACE

Description

Displays all current tracepoints and whether any opcode tracing is in effect.

Format

SHOW TRACE

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG> SHOW TRACE
tracepoint at CALC\MULT
tracing /CALL instructions: CALLS, CALLG, BSBW, BSBB, JSB, RSB
and RET
```

SHOW TYPE

Description

Displays the current default data type. If you specify the /OVERRIDE qualifier, the debugger displays the current default override data type. Note that the SHOW MODE command displays both the current data type and the override data type.

If the data type is /ASCII:length, the debugger displays length as a decimal number even if the current radix is not decimal.

Format

SHOW TYPE [/qualifier]

Command Qualifiers

/OVERRIDE

Command Parameters

None.

Command Qualifiers

/OVERRIDE

Displays the current override default data type. The minimum abbreviation is /OVERR.

Examples

```
DBG>SH TY
type:  long integer
```

SHOW WATCH

Description

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

Format

SHOW WATCH

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG>SH WATCH
watchpoint at MAIN\ALPHA for 4. bytes
watchpoint at SUB2\TABLE+20 for 4. bytes
```

STEP

Description

Executes one or a specified number of instructions or lines. STEP is useful when you want to go through your program a step at a time.

The unit of increment that is executed depends on the step conditions specified in command qualifiers or the current default step conditions. The initial conditions are dependent on the current language but can be changed by the SET STEP command.

Format

STEP [/qualifier] [integer]

Command Qualifiers

/INSTRUCTION
/INTO
/LINE
/OVER
/NOSYSTEM
/SYSTEM

Command Parameters

integer

Specifies the number of units of the program to be executed before the debugger interrupts execution. The debugger always interprets the integer as a decimal number even if the current radix mode is not decimal. If you do not specify integer, the debugger executes one unit of the program.

Command Qualifiers

/INSTRUCTION

Steps in increments of instructions (the only valid increment for VAX-11 MACRO). Displays the next instruction to be executed after the step is completed.

/INTO

Steps into a routine entered by a call or branch to subroutine instruction (CALLG, CALLS, BSBB, BSBW, and JSB).

/LINE

Steps in increments of lines for line-oriented languages, such as FORTRAN and BASIC (ignored for VAX-11 MACRO).

/NOSYSTEM

Ignores all instructions in system space. System space includes the procedures called to perform system services and input/output. The debugger does not count any instructions or lines executed in system space.

/OVER

Treats a routine entered by a call or branch instruction as part of the call or branch instruction. It is useful to step over a routine when you are debugging one procedure and do not want to debug any other procedures that it calls.

/SYSTEM

Treats instructions in system space the same as other instructions. The /SYSTEM qualifier is useful when you want to step through a system service routine. This qualifier allows you to step through code in system space that is executing in user mode.

Examples

```
DBG>STEP
start at MAIN\MAIN+09
stepped to MAIN\MAIN+14: MOVC5    #05,L^0000020A, #20, #14,B^0EC(R11)
DBG>STEP/LINE
start at MAIN\MAIN+14
stepped to MAIN\MAIN+1F
DBG>STEP
start at MAIN\MAIN+1F
stepped to MAIN\MAIN+24: CALLG    B^14(R11,L^SUB1
DBG>STEP/INTO
start at MAIN\MAIN+24
stepped to routine SUB1: MOVAL    L^0000060C,R11
```

APPENDIX A

VAX-11 SYMBOLIC DEBUGGER MESSAGES

If the debugger encounters an error, it displays a message on the terminal. The general format of a debugger message is:

```
%DEBUG-l-code, text
```

l
: A severity level indicator. It has a value of I for informational messages, W for warning messages, E for error messages, and F for fatal messages.

code
: An abbreviation of the message text; the message description in this appendix are alphabetized by this code.

text
: The explanation of the message.

For example:

```
%DEBUG-W-DIVBYZERO, attempted to divide by zero
```

Listed below are the messages displayed by the debugger. Each message is accompanied by an explanation of the cause of the error and the recommended user action to correct the error.

BADOPCODE, opcode xxx is unknown

Explanation: The opcode xxx specified in the deposit command is unknown to the debugger. If the opcode is a valid VAX-11 MACRO opcode, then it is an opcode that has a synonomous opcode. These opcodes, such as MOVAF and MOVAL, generate the same instruction. The debugger only recognizes one of them. Severity is warning.

User Action: Specify a valid opcode or specify the opcode's synonym that the debugger accepts.

BADSCOPE, invalid pathname xxx, SCOPE not changed

Explanation: The scope xxx specified in the SET SCOPE command contained a pathname that does not exist. Severity is warning.

User Action: Specify a valid scope.

BADSTARTPC, cannot access start PC = xxx

Explanation: Location xxx is not an accessible address and therefore cannot be executed. This is often caused when a GO command with no address specification is entered after the program has terminated. The debugger tries to execute an instruction at location 0, which is not accessible. Severity is warning.

User Action: Specify a different address specification in the GO command or, if the program has terminated, you can exit from the debugger and initiate the program with the DCL RUN command.

BADWATCH, cannot watch protected address xxx

Explanation: A SET WATCH command specified a protected address. Note that you can not place a watchpoint on a dynamically allocated variable because these variables are stored on the stack. FORTRAN dummy arguments and most BASIC variables and arrays are dynamically allocated. Severity is warning.

User Action: Do not use watchpoint on this address.

BITRANGE, bit range out of limits

Explanation: The EVALUATE command specified a bit field that is too wide. Severity is warning.

User Action: The low limit of the bit field is 0 and the high limit is 31; the maximum range is <0:31>.

BRTOOFAR, destination xxx is too far for branch operand

Explanation: The DEPOSIT command specified a branch instruction with a destination, xxx, too far from the current PC. Severity is warning.

User Action: Change a BRB instruction to BRW or a BRW to JMP or specify a closer address.

DBGBUG, internal DEBUG coding error, please report no. nnn

Explanation: An internal debugger error has been encountered. Severity is informational.

User Action: If the error is reproducible, submit a Software Performance Report (SPR) and if possible, enclose both a copy of the program being debugged and a logged debugging session that reproduces the error.

DBGERR, internal DEBUG coding error

Explanation: An internal debugger error has been encountered. Severity is error.

User Action: If the error is reproducible, submit a Software Performance Report (SPR) and if possible, enclose both a copy of the program being debugged and a logged debugging session that reproduces the error.

DEBUGBUG, internal DEBUG coding error, please report no. nnn

Explanation: An internal debugger error has been encountered. Severity is error.

User Action: If the error is reproducible, submit a Software Performance Report (SPR) and if possible, enclose both a copy of the program being debugged and a logged debugging session that reproduces the error.

DIVBYZERO, attempted to divide by zero

Explanation: An expression contained a division by 0. Severity is warning.

User Action: Reformulate the expression.

EXARANGE, invalid range of addresses

Explanation: The range of addresses specified was in the wrong order. The higher address preceded the lower address. Severity is warning.

User Action: Reenter the command with a valid address range.

EXCEEDACT, number of access actuals supplied exceeds limit of nnn

Explanation: There are too many access actuals in the BLISS structure reference. This structure has nnn access actuals. Severity is warning.

User Action: Specify nnn or less access actuals.

EXITSTATUS, is xxx

Explanation: The program has exited with the status xxx. See the VAX/VMS System Services Reference Manual for more information about the VAX/VMS exit status codes. Severity is informational.

User Action: None.

EXPSTKOVR, expression exceeds maximum nesting level

Explanation: The expression is too complex. Severity is warning.

User Action: Reduce the nesting of parentheses and simplify the expression.

FRERANGE, storage package range error

Explanation: Data used to control internal storage allocation is corrupt. Severity is error.

User Action: If DEPOSIT commands or the user program has not modified the debugger's storage area, submit an SPR.

FRESIZE, storage package size error

Explanation: Data used to control internal storage allocation is corrupt. Severity is error.

User Action: If DEPOSIT commands or the user program has not modified the debugger's storage area, submit an SPR.

IMPTERMNO, improperly terminated numeric string nnn

Explanation: Numeric string nnn with radix control did not have a terminating apostrophe. Severity is warning.

User Action: Terminate the string with an apostrophe.

INITIAL, language is xxx, module set to yyy

Explanation: This message is displayed when the debugger is invoked by the image activator. The language is set to xxx, and the module to yyy. Module yyy is the first module specified in the LINK command and language xxx is the language used in that module. Severity is informational.

User Action: None.

INTEGER, this operation only valid on integers

Explanation: Command specified an operation with operands that did not have integer values when integer values are required. Severity is warning.

User Action: Use only operands that have integer values in specified operation.

INVACCESS, structure requires nnn access actuals, mmm were supplied

Explanation: A reference to a VAX-11 BLISS structure had nnn actual arguments, the structure requires mmm. Severity is warning.

User Action: Supply correct number of actual arguments.

INVARRDSC, invalid array descriptor

Explanation: An array descriptor in the image does not have the correct format. This can be caused by a reference to a VAX-11 BASIC array when the first line of the program has not been executed. The array is not set up correctly until the BASIC program initialization is done. This message can also be caused by a user program or DEPOSIT commands altering a compiler generated array descriptor. Severity is warning.

User Action: If the reference is to a VAX-11 BASIC array, enter a step or GO command to ensure that the BASIC program initialization is done and then repeat the reference. Otherwise, if an array descriptor has not been altered, submit an SPR.

INVCHAR, invalid character

Explanation: The command contained a character that is invalid in the command's context. Severity is warning.

User Action: Reenter command.

INVDIM, subscript error, was declared dimension (string)

Explanation: A reference to an array contained either an incorrect number of subscripts or the value of the subscripts is outside of the bounds of the array. The dimension of the array was specified by string. Severity is warning.

User Action: Specify the correct number of subscripts with values in the correct range.

INVDSTREC, invalid DST record

Explanation: An invalid debug symbol table record was encountered. Severity is error.

User Action: Submit an SPR unless the user program or a DEPOSIT command has altered a debug symbol table record.

INVFLOAT, variable has invalid floating point format

Explanation: A floating point number has an invalid bit pattern. Can be caused by depositing a value with a type qualifier into an address associated with a floating point type variable. Severity is informational.

User Action: Examine the value of the symbol with a type qualifier to override the floating point format.

INVNUMBER, invalid numeric string 'nnn'

Explanation: The numeric string 'nnn' is invalid in the current language. Severity is warning.

User Action: Specify the value in another numeric format or set the language to one that accepts this type of numeric string.

INVOPR, unrecognized operator in expression

Explanation: An expression contained a character that the debugger does not recognize. Severity is warning.

User Action: Reenter the command with a valid expression.

INVPATH, improperly terminated pathname beginning with xxx

Explanation: The pathname beginning with xxx is not a valid pathname. Severity is warning.

User Action: Check the pathname for errors and reenter command.

LABNOTFND, search for %label xxx using scope failed

Explanation: The label xxx could not be found using the specified pathname or the current default scope. Severity is warning.

User Action: Reenter the command with the correct pathname.

LASTCHANCE, stack exception handlers lost, re-initializing stack

Explanation: Error in user program caused the exception handling mechanism to fail. Can be caused when the stack is overwritten by the user program or by deposit commands. Severity is warning.

User Action: Identify and correct the error in the user program.

LINNOTFND, search for %line nnn using scope failed

Explanation: The line nnn specified does not exist in the default scope(s). Note that the debugger does not search for a unique line number but only searches the current default scope list. Severity is warning.

User Action: Specify the scope of the line or the correct line number.

LONGDST, too many modules - some ignored

Explanation: There are too many modules in the image for the debugger to keep track of. The excess modules are ignored. You cannot set the module to any of the ignored modules.

User Action: Use SHOW MODULE command to determine which modules are included. If crucial modules were omitted, relink the image, specifying first the modules needed for debugging.

MAXDIMSN, maximum number of subscripts is nnn

Explanation: An array reference specified too few or too many subscripts. The array has nnn subscripts. Severity is warning.

User Action: Reenter the command with correct number of subscripts.

MODNOTADD, no space to add module yyy

Explanation: There was no room to add the modules specified in the SET MODULE command to the symbol table. Severity is informational.

User Action: Use the SHOW MODULE command to show the modules currently in the symbol table and the remaining space, and then use the CANCEL MODULE command to free the needed space.

MULTOPR, multiple successive operators in expression

Explanation: There were two adjacent operators in expressions. Severity is warning.

User Action: Use angle brackets or parentheses (depending on the current language) to separate the operators or enter a valid expression.

NEEDMORE, unexpected end of command line

Explanation: The command entered was not complete. A required part of a command was omitted. Severity is warning.

User Action: Reenter the complete command.

NOACCESSR, no read access to virtual address nnn

Explanation: The debugger does not have read access to the address specified. Can be caused when an EXAMINE command with no address specification is entered at the beginning of a debugging session. Severity is warning.

User Action: Specify an address that is within the image.

NOACCESSW, no write access to virtual address nnn

Explanation: A DEPOSIT, SET BREAK, or SET TRACE command specified the address nnn that the debugger does not have write access to the page. The debugger requires write access in order to be able to set up breakpoints and tracepoints. Severity is warning.

User Action: You cannot do the requested operation.

NOANGLE, unmatched angle brackets in expression

Explanation: An expression did not have a closing right angle bracket. Severity is warning.

User Action: Reenter the command with a complete expression.

NOBRANCH, instruction requires branch-type operand

Explanation: A DEPOSIT command specified a branch-type instruction that specified an illegal addressing mode as the operand. Severity is warning.

User Action: Reenter the command using a valid branch operand in the destination field.

NOBREAKS, no breakpoints are set

Explanation: The SHOW BREAK command was entered and no breakpoints were set. Severity is informational.

User Action: None.

NOCALLS, no active call frames

Explanation: The SHOW CALLS command was entered and there were no active calls. There are no active calls after your program has terminated. Severity is warning.

User Action: None.

NOCNVT, incompatible types, no conversion

Explanation: DEPOSIT command specified incompatible data for the variable type. Severity is warning.

User Action: Reenter the command and either specify the correct data type or use a type qualifier.

NODECODE, cannot decode instruction

Explanation: The address specified in the EXAMINE command is not the beginning of a valid VAX-11 instruction. This can be caused by specifying an address that is in the middle of an instruction or is in a data area. Severity is warning.

User Action: Specify an address that contains a valid instruction.

NODELIMTR, missing or invalid instruction operand delimiter

Explanation: A DEPOSIT command specified an invalid instruction operand format. Severity is warning.

User Action: Reenter the command with valid operands.

NOEND, string beginning with xxx is missing end delimiter y

Explanation: A DEPOSIT command specified an ASCII or INSTRUCTION string beginning with characters xxx that did not have a terminating apostrophe. Severity is warning.

User Action: Reenter the command with a terminating apostrophe.

NOFREE, no free storage available

Explanation: No free storage is available for the debugger to execute the command. Severity is error.

User Action: Free storage by using the CANCEL MODULE command and then reenter command.

NOGLOBALS, some or all global symbols not accessible

Explanation: The image was linked with the /NODEBUG qualifier and there are no global symbols in the symbol table. This message can also be caused if the image has too many global symbols. Severity is informational.

User Action: Relink the image with the /DEBUG qualifier or, if the message was caused by an overflow condition, remove some of the global symbol definitions from the image (if possible).

NOINSTRAN, cannot translate opcode at location xxx

Explanation: The address specified in the EXAMINE command is not the beginning of a valid VAX-11 instruction. This can be caused by specifying an address that is in the middle of an instruction or is in a data area. Severity is warning.

User Action: Specify an address that contains a valid instruction.

NOLABEL, routine xxx has no %label nnn

Explanation: The label nnn does not exist in the scope xxx that is specified in the pathname. Severity is warning.

User Action: Specify a valid pathname -- either change the label or the scope.

NOLINE, routine xxx has no %line nnn

Explanation: The line nnn does not exist in the scope xxx that is specified in the pathname. This can be caused by a source line number that does not exist or that does not contain executable code in the source program. Severity is warning.

User Action: Specify a valid pathname -- either change the line number or the scope.

NOLITERAL, no literal translation exists for xxx

Explanation: The command attempted to find a literal translation for a value. The debugger does not support this operation. Severity is warning.

User Action: None.

NOLOCALS, image does not contain local symbols

Explanation: All of the modules in the image were compiled or assembled without traceback information. There is no local symbol information in the image. Severity is informational.

User Action: Recompile or reassemble the modules and then relink them.

NOOPRND, missing operand in expression

Explanation: One or more operands are missing from an expression. Severity is warning.

User Action: Reenter command with complete expression.

NORSTBLD, cannot build symbol table

Explanation: The debugger is unable to build a symbol table because of errors in the format of the image file. Severity is error.

User Action: Relink the image and, if the error is reproducible, submit an SPR explaining how the image file was created.

NOSUCHBPT, no such breakpoint

Explanation: The CANCEL BREAK command specified an address that is not the address of a breakpoint. Severity is informational.

User Action: Use the SHOW BREAK command to find the location of the current breakpoints and then cancel any breakpoints that you want to cancel.

NOSUCHLAN, language xxx is unknown

Explanation: The debugger does not recognize the language specified. This message can be caused by mistyping a language that the debugger supports or by entering a language that the debugger does not currently support. Severity is warning.

User Action: Specify a valid language in the SET LANGUAGE command.

NOSUCHMODU, module xxx is not in module chain

Explanation: The module xxx, specified in the SET MODULE command, does not exist in the image. This message can be caused when a module name has been entered incorrectly or when the image had too many modules for the debugger to handle. Severity is warning.

User Action: Specify a module that is in the image.

NOSUCHTPT, no such tracepoint

Explanation: The CANCEL TRACE command specified an address that was not the address of a tracepoint. Severity is informational.

User Action: Use the SHOW TRACE command to display the current tracepoints and then cancel any that you want to cancel.

NOSUCHWPT, no such watchpoint

Explanation: The CANCEL WATCH command specified an address that was not the address of a watchpoint. Severity is informational.

User Action: Use the SHOW WATCH command to display the current watchpoints and then cancel any that you want to cancel.

NOSYMBOL, symbol xxx is not in the symbol table

Explanation: The symbol xxx cannot be located in the debugger's symbol table. This can be caused when the module that defines the symbol has not been added to the debugger's symbol table or when a symbol name that is not in the image has been entered. Severity is warning.

User Action: Add the required module to the debugger's symbol table with the SET MODULE command or specify the correct symbol name.

NOTALLSYM, cannot initialize symbols for default module

Explanation: The debugger could not put the symbol table information for the first module specified in the LINK command into the symbol table. Severity is informational.

User Action: Use the SET MODULE command to add modules to the symbol table.

NOTASTRUCT, xxx was not declared as a structure

Explanation: A VAX-11 BLISS structure reference specified a symbol xxx that was not declared a structure. Severity is warning.

User Action: Reenter the command with a valid symbol reference.

NOTDONE, xxx not yet a supported feature

Explanation: A DEPOSIT command specified an instruction with a literal that the debugger does not yet support. Severity is warning.

User Action: None.

NOTIMPLAN, xxx is not implemented at command level

Explanation: The SET LANGUAGE command specified a language that the debugger recognizes but does not yet support. Severity is warning.

User Action: Specify a language that the debugger supports.

NOTLINBND, program is not at a line boundary

Explanation: The GO command specified an address that contains threaded code data. The debugger cannot execute starting from this address. Severity is warning.

User Action: Specify an address that contains executable instructions.

NOTRACES, no tracepoints are set, no opcode tracing

Explanation: There are no tracepoints or opcode tracing set. Severity is informational.

User Action: None.

NOUNIQUE, SYMBOL xxx IS NOT UNIQUE

Explanation: The symbol specified was not in a default scope and was defined in more than one scope. Severity is warning.

User Action: Specify the scope of the symbol in a pathname or change the default scope.

NOWATCHES, no watchpoints are set

Explanation: No watchpoints are set. Severity is informational.

User Action: None.

NOWBPT, cannot insert breakpoint

Explanation: Internal debugger error. Severity is fatal.

User Action: Submit an SPR.

NOWOPCO, cannot replace breakpoint with opcode

Explanation: Internal debugger error. Severity is fatal.

User Action: Submit an SPR.

NOWPROT, cannot set protection

Explanation: Internal debugger error. Severity is fatal.

User Action: Submit an SPR.

NUMOPRNDS, xxx instructions must have nnn operands

Explanation: A DEPOSIT command specified the xxx instruction with an incorrect number of operands. This instruction requires nnn operands. Severity is warning.

User Action: Reenter the command with the instruction having the correct number of operands.

NUMTRUNC, number truncated

Explanation: The debugger truncated the numeric data because it exceeded the length of the data type. Severity is informational.

User Action: None.

OPSYNTAX, instruction operand syntax error

Explanation: The DEPOSIT command contained an instruction with an operand syntax error. Severity is warning.

User Action: Reenter the command with a valid instruction.

OUTPUTLOST, output being lost, both NOTERMINAL and NOLOG are in effect

Explanation: The SET OUTPUT command has set the output conditions to NOTERMINAL and NOLOG; consequently, the output is not displayed on the terminal or written to a log file. The output normally displayed by the debugger will not be available. Severity is informational.

User Action: Use the SET OUTPUT command to send output to the terminal or to a log file.

PARSEERR, internal parsing error

Explanation: Internal debugger error. Severity is warning.

User Action: Submit an SPR.

PARSTKOVR, parse stack overflow, simplify expression

Explanation: The expression was too complex for the debugger to evaluate. Severity is warning.

User Action: Simplify the expression.

PATHTLONG, too many qualifiers on name

Explanation: There were too many elements in a pathname. Severity is warning.

User Action: Reduce number of elements in pathname, if possible.

PCNOTINSCP, PC is not within the scope of the routine declaring symbol

Explanation: A dynamically allocated variable was referenced and the variable was not defined in the scope that contains the current PC. The value of the variable is undefined when you are not currently executing the scope in which it is defined. The debugger uses a value for the variable that may have no relation to the symbol's current value. Note that VAX-11 FORTRAN dummy arguments and most VAX-11 BASIC variables and arrays are dynamically allocated. Severity is informational.

User Action: Only reference a dynamically allocated variable when you are currently executing the scope in which it is defined.

REDEFREG, register name already defined

Explanation: DEFINE command attempted to redefine a register name. The command is ignored because you cannot redefine register names. Severity is warning.

User Action: None.

RESOPCODE, opcode xxx is reserved

Explanation: Opcode xxx is reserved for use by DIGITAL. Severity is warning.

User Action: None.

RSTERR, error in symbol table

Explanation: There is a format error in the symbol table. Severity is error.

User Action: If this is not caused by a user program error or a DEPOSIT command, submit an SPR.

SIZETRUNC, size field truncated to 32 bits

Explanation: The size entry in a VAX-11 BLISS field specification was larger than 32. The debugger set the size entry to 32 and executed the command. Severity is informational.

User Action: None.

STEPINTO, cannot step over PC = xxx

Explanation: The debugger was unable to step over the routine and executed a step into the routine instead. Severity is informational.

User Action: None.

STGTRUNC, string truncated

Explanation: The debugger truncated an ASCII string because it exceeded the size of the ASCII data type. Severity is informational.

User Action: None.

STMNOTFND, Line nnn.mmm not found.

Explanation: The VAX-11 BASIC statement number mmm was not found on line nnn. Severity is warning.

User Action: Reenter the command with valid line and statement numbers.

STRUCSIZE, structure size declared as xxx units, yyy was given

Explanation: The VAX-11 BLISS structure size was declared to be xxx units but was referenced with yyy units. Severity is informational.

User Action: None.

SUBSTRING, invalid substring (nnn:mmm), was declared CHARACTER*ppp

Explanation: The substring specification (nnn:mmm) is not within the bounds defined for the CHARACTER data type. Severity is warning.

User Action: Specify a substring specification within the bounds defined for the data type.

SYNTAX, command syntax error at or near xxx

Explanation: The debugger encountered a command syntax error near the element xxx. Severity is warning.

User Action: Reenter the command.

VERIFYICF, xxx indirect command file yyy

Explanation: The debugger is verifying an indirect command file. This message is displayed before the command file is executed and after all the commands have been displayed. Severity is informational.

User Action: None.

APPENDIX B

COMPATIBILITY FEATURES

This appendix describes features in the VAX-11 Symbolic Debugger, Version 2 that provide compatibility with the VAX-11 Symbolic Debugger, Version 1. The Version 2 debugger has a different definition of entry and display modes than the Version 1 debugger. Some entry and display modes in the Version 1 debugger are now considered data types. For compatibility with the Version 1 debugger, the Version 2 debugger accepts ASCII, BYTE, INSTRUCTION, LONG, or WORD in a SET MODE command, but treats such commands as if they were SET TYPE or SET TYPE/OVERRIDE commands.

Table B-1 lists the SET MODE commands accepted by the debugger and the equivalent SET TYPE command. These SET MODE commands may not be supported in future versions of the debugger.

Table B-1
Equivalent Commands

Compatible Version 1 Commands	Equivalent Version 2 Commands
SET MODE ASCII	SET TYPE/OVERRIDE ASCII:4
SET MODE BYTE	SET TYPE BYTE
SET MODE INSTRUCTION	SET TYPE/OVERRIDE INSTRUCTION
SET MODE LONG	SET TYPE LONG
SET MODE WORD	SET TYPE WORD

APPENDIX C

COMMAND SUMMARY

This appendix lists in alphabetical order each debugger command with a format description, a list of qualifiers, a list of parameters, and a brief description of the command's function. The boldface letters indicate the minimum abbreviation that you must type in order for the debugger to recognize the command name, qualifier or parameter.

Format	Function
@file-spec	Accept commands from specified command procedure.
CALL name [(argument-list)]	Calls the specified procedure.
CANCEL keyword [/qualifier] [parameters]	Cancels specified item.
CANCEL **ALL**	Cancels all breakpoints, tracepoints, and watchpoints and restores scope and entry/display modes and types to their default values.
CANCEL **B**REAK [/qualifier] [address-expression] /**ALL**	Cancels specified breakpoint or all breakpoint.
CANCEL **EXC**EPTION **B**REAK	Cancels effects of SET EXCEPTION BREAK.
CANCEL **M**ODE	Sets all modes and types to their default value for the current language.
CANCEL **MODU**LE [/qualifier] module [,module...] /**ALL**	Cancels specified modules or all modules.
CANCEL **SC**OPE	Sets scope to its default value (PC scoping).
CANCEL **T**RACE [/qualifier] [address-expression] /**ALL** /**BRA**NCH /**C**ALL	Cancels the specified tracepoint or the specified opcode tracing or all tracepoints and opcode tracing.
CANCEL **TY**PE/**OVERR**IDE	Sets the override type to none.
CANCEL **W**ATCH [/qualifier] [address-expression] /**ALL**	Cancels the specified watchpoint or all watchpoints.
<CTRL/C>	Interrupts execution of the program.
<CTRL/Y>	Interrupts execution of the program.
<CTRL/Z>	Equivalent to the EXIT command.
DEFINE symbol=expression [,symbol=expression...]	Defines the specified symbol(s) and assigns them the specified address(es).

Format	Function
DEPOSIT [/qualifier]...address-expression = data [,data ...] /**ASC**II:length /**BYT**E /**DEC**IMAL /**H**EXADECIMAL /INSTRUCTION /**LON**G /**O**CTAL /**WO**RD	Stores the specified value(s) at the specified location.
EVALUATE[/qualifier]...expression [,expression...] **EV**ALUATE[/qualifier]...value<high-bit:low-bit> /**AD**DRESS /**DEC**IMAL /**H**EXADECIMAL /**O**CTAL	Evaluates expressions or bit ranges, and displays value. /ADDRESS is not valid in VAX-11 MACRO or VAX-11 BLISS.
EXAMINE [/qualifier]...addr-expr[:addr-expr] [,addr-expr[:addr-expr]...] /**ASC**II:length /**BYT**E /**DEC**IMAL /**H**EXADECIMAL /INSTRUCTION /**LON**G /**NOSY**MBOLIC /**O**CTAL /**SY**MBOLIC /**WO**RD	Displays the contents of the specified addresses and range of addresses.
EXIT	Ends a debugging session or specifies the end of a command procedure.
GO [address-expression]	Starts or continues program execution.
HELP topic [subtopic ...]	Displays a description of the specified command.
SET keyword [/qualifier] parameter	Sets specified item.
SET **B**REAK[/qualifier] address-expression [**DO** (cmd[;cmd...])] /**AF**TER:count	Establishes a breakpoint at the specified address.
SET **EXC**EPTION **B**REAK	Requests that the debugger treat external exception conditions as breakpoints.
SET **LA**NGUAGE language-name	Sets the current language.
SET **LO**G file-specification	Sets the file specification of the log file.
SET **M**ODE mode-keyword [,mode-keyword]	Sets the entry/display modes. Mode-keyword can be: **DEC**IMAL, **H**EXADECIMAL, **O**CTAL, **NOSY**MBOLIC, or **SY**MBOLIC.
SET **MODU**LE [/qualifier] [module-name [,module-name] ...] /**ALL**	Adds the symbols in the specified modules or all modules to the debugger symbol table.
SET **OU**TPUT option [,option ...]	Controls the debugger's output configuration. Option can be **LOG**, **NOLOG**, **TERM**INAL, **NOTERM**INAL, **VERIFY**, or **NOVERIFY**.
SET **SC**OPE scope [,scope ...]	Specifies scopes to be searched to find a symbol.
SET **ST**EP condition [,condition ...]	Specifies the step conditions. Condition can be INSTRUCTION, **LINE**, **INTO**, **OVER**, **SYSTEM**, or **NOSYSTEM**.
SET **TR**ACE [/qualifier] [address-expression] /**BRA**NCH /**C**ALL	Establishes a tracepoint at the specified address or establishes the specified opcode tracing.
SET **TY**PE [/qualifier] type-keyword /**OVERR**IDE	Sets the default data type for the DEPOSIT and EXAMINE commands. Type-keyword can be **ASC**II:length, **BYT**E, **IN**STRUCTION, **LON**G, or **WO**RD.

Format	Function
SET WATCH address-expression	Establishes a watchpoint at the specified address.
SHOW keyword [/qualifier] [parameter]	Displays the current value of the specified item.
SHOW BREAK	Displays current breakpoints.
SHOW CALLS [integer]	Displays current location and previous calls.
SHOW LANGUAGE	Displays current language.
SHOW LOG	Displays the name of the log file.
SHOW MODE	Displays current entry/display modes and types.
SHOW MODULE	Lists the modules in the image and show which are currently included in the debugger symbol table.
SHOW OUTPUT	Displays the debuggers output configuration.
SHOW SCOPE	Displays the current scope search list.
SHOW STEP	Displays current STEP conditions
SHOW TRACE	Displays current tracepoints and opcode tracing.
SHOW TYPE **/OVERRIDE**	Displays current default type or override type.
SHOW WATCH	Display current watchpoints.
STEP [/qualifier] [integer] /INSTRUCTION **/LINE** **/INTO** **/OVER** **/NOSYSTEM** **/SYSTEM**	Executes one or a specified number of instructions or lines.

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

V

W

X

VAX-11
Symbolic Debugger
Reference Manual
AA-D026B-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 cut along this line.

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

Do Not Tear - Fold Here