VAX/VMS Debugger Reference Manual

Order Number: AA–Z411C–TE

April 1986

This manual explains the features of the VAX/VMS Debugger for programmers in high-level languages and assembly language.

Revision/Update Information:	This revised document supersedes the *VAX/VMS Symbolic Debugger Reference Manual* Version 4.2
Software Version:	VAX/VMS Version 4.4

digital equipment corporation maynard, massachusetts

April 1986

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

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem–10	PDP	VT
DECSYSTEM–20	PDT	
DECUS	RSTS	
DECwriter	RSX	digital

ZK–3031

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire
03061

CANADA

Digital Equipment
of Canada Ltd.
100 Herzberg Road
Kanata, Ontario K2K 2A6
Attn: Direct Order Desk

INTERNATIONAL

Digital Equipment Corporation
PSG Business Manager
c/o Digital's local subsidiary
or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.

In New Hampshire, Alaska, and Hawaii call 603-884-6660.

In Canada call 800-267-6215.

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a trademark of the American Mathematical Society.

Contents

PART II DEBUGGER COMMAND DICTIONARY

EXAMPLES

FIGURES

TABLES

Preface

Intended Audience

Programmers at all levels of experience can use this manual effectively.

New users should start with Chapter 1 (Introduction to the VAX/VMS Debugger). It contains an overview of debugger features, a brief interactive tutorial on the debugger, and a summary of debugger commands.

The debugger can be used with most VAX/VMS supported languages (language support is summarized in Appendix E). This manual emphasizes usage that is common to all or most languages. For additional information that is specific to a particular language, you should also refer to the documentation furnished with that language.

Note that you can use the VAX/VMS debugger only to debug code in user mode. You cannot debug any code in supervisor, executive, or kernel modes. If you need to debug code in other than user mode, refer to the *VAX/VMS Delta/XDelta Utility Reference Manual*, which describes the VAX/VMS DELTA /XDELTA Utility.

Structure of This Document

This manual is organized in three parts:

- Chapters 1 through 9 present task-oriented and conceptual information about the debugger.
- Part II, which follows Chapter 9, is the Debugger Command Dictionary. It lists all debugger commands alphabetically and provides complete information on command format, parameters, and qualifiers, as well as examples of each command's use.
- Part III, which follows the Command Dictionary, consists of appendices. These provide reference information on various topics such as predefined keypad-key functions, debugger support of specific languages, and so on.

Associated Documents

Information on compiling and debugging in a particular language may be found in the documentation furnished with that language.

Information on the linking of programs and on shareable images may be found in the *VAX/VMS Linker Reference Manual*.

Conventions Used in This Document

Convention	Meaning
RET	A symbol with a one- to three-character abbreviation indicates that you press a key on the terminal, for example, RET .
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, CTRL/O.
$ SHOW TIME 05-JUN-1985 11:55:22	Command examples show all output lines or prompting characters that the system prints or displays in black letters. All user-entered commands are shown in red letters.
[expression]	Square brackets indicate that the enclosed item is optional. (Square brackets are not, however, optional in the syntax of a directory name in a file specification.)
file-spec, . . .	Horizontal ellipsis within command syntax indicates that additional parameters, values, or information can be entered.
$ TYPE MYFILE.DAT . . .	Vertical series of periods, or ellipsis, means either that not all the data that the system would display in response to the particular command is shown or that not all the data a user would enter is shown.
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.

Summary of Technical Changes

The following technical changes have been made to the VAX/VMS Debugger since VAX/VMS Version 4.2.

The debugger now supports VAX DIBOL and VAX SCAN. Appendix E contains a summary of debugger support for these languages. Refer to the DIBOL and SCAN documentation for complete information.

In addition, the following debugger features and commands have been added and are fully documented in this manual.

Several enhancements have been made to screen mode (see Chapter 8 and Appendix C):

- There are four new display attributes to be used with the SELECT command: /INPUT, /ERROR, /PROGRAM, and /PROMPT. These give you the option of mixing various types of debugger output in a display, or directing different types of output to separate displays.
- There is a new PROMPT built-in display. PROMPT shows the debugger prompt and your input and may be manipulated like other displays, within certain restrictions.
- You can now divide display windows vertically. The SET WINDOW, SET DISPLAY, and DISPLAY commands have been changed accordingly.
- There are some new built-in window definitions. In addition to the full height and width of the screen, the predefined windows include all possible regions that result from dividing the screen vertically into halves, thirds, quarters, and sixths, and horizontally into left and right halves.
- The built-in symbols %PAGE and %WIDTH have been added. These specify the current screen height and width of the terminal.
- A new MOVE command lets you move displays across the screen.
- A new EXPAND command lets you expand and contract displays.
- A new EXTRACT command lets you save screen displays into a file, or create a file with all the DEBUG commands necessary to re-create the current screen state at a later time.
- All displays, except for register displays, are now dynamic by default. This means that, if you change the terminal screen height or width, display windows will be resized in proportion to maintain their relative dimensions. The new qualifier /[NO]DYNAMIC lets you control this feature when using the DISPLAY and SET DISPLAY commands.
- The /[NO]POP and /[NO]PUSH qualifiers have been added to the DISPLAY and SET DISPLAY commands. These qualifiers give you the option of either popping or pushing a display to the top or bottom of the display "pasteboard", to show or hide that display when you issue those commands.

- The SHOW DISPLAY and SHOW WINDOW commands have been enhanced. You can now specify a list of parameters, wildcards, and the /ALL qualifier with these commands.
- There are some new and changed keypad key definitions associated with screen mode (see Appendix B).

Other new features and commands are as follows:

- You can now debug shareable images. The SET IMAGE, SHOW IMAGE, and CANCEL IMAGE commands have been added.
- The new SET ATSIGN command lets you establish a default file specification that the debugger will use when searching for command procedures. There is also a new SHOW ATSIGN command.
- The new SET EDITOR command lets you establish an editor of your choice to be invoked by the EDIT command. There is also a new SHOW EDITOR command.
- The new SHOW STACK command provides detailed information about the current call stack.
- The /[NO]SHARE and /[NO]JSB qualifiers have been added to the SET BREAK, SET TRACE, and STEP commands. The [NO]SHARE and [NO]JSB parameter keywords have been added to the SET STEP command. These features let you qualify breakpoints, tracepoints, and the STEP mode.
- The /TYPE qualifier has been added to the EXAMINE and DEPOSIT commands. The TYPE parameter keyword has been added to the SET TYPE command. This feature lets you influence the debugger's interpretation of data in untyped locations.
- When the debugger searches for symbols, it now uses the default scope search list of 0,1,2,3, . . . n (see the description of the SET SCOPE command).
- You can now invoke the debugger from your program by signalling the condition SS$_DEBUG (see Appendix D).

Part I Using the VAX/VMS Debugger

1 Introduction to the VAX/VMS Debugger

This chapter introduces new users to the VAX/VMS Debugger. Section 1.1 summarizes the debugger's features. Section 1.2 gets you started using the basic functions and commands. Section 1.3 orients you to the debugger commands by grouping them in general categories, along with short descriptions.

1.1 Overview of the Debugger

Suppose you have compiled and linked your program successfully. You now run the program and discover some error. For example, the program terminates prematurely, or goes into an infinite loop, or gives incorrect output. The VAX/VMS Debugger helps you locate such run-time programming or logic errors, also known as bugs.

The debugger lets you observe and manipulate your program interactively as it executes. By issuing debugger commands at the terminal, you can

- start, stop, and resume the program's execution
- trace the execution path of the program
- monitor selected locations, variables, or events
- examine and modify the contents of variables, or force events to occur.
- in some cases, test the effect of modifications without having to edit the source code, recompile, and relink

These are the basic debugging techniques at your disposal. Once you are satisfied that you have found the error in the program, you can edit the source code and compile, link, and execute the corrected version.

As you use the debugger and its documentation, you will discover variations on the basic techniques. You will also find that you can tailor the debugger to your own debugging style. The features of the debugger are summarized below.

1.1.1 Functional Features

Programming Language Support

You can use the debugger with the following VAX/VMS supported languages: Ada, BASIC, BLISS, C, COBOL, DIBOL, FORTRAN, MACRO-32, PASCAL, PL/I, RPG, and SCAN. The debugger recognizes the syntax, expressions, data typing, and other constructs of a given language. If your program is written in more than one language, you can change from one language to another in the course of a debugging session.

Symbolic Debugging

The VAX/VMS Debugger is a symbolic debugger. You can refer to program locations by the symbols you used for them in your program (the names of variables, routines, labels, and so on). You do not have to use virtual addresses to refer to memory locations.

Support for All Data Types

The debugger understands all language data types, such as integer, floating point, enumeration, record, array, and so on. It displays program variables according to their declared type.

Flexible Data Format

The debugger permits a variety of data forms and types for entry and display. By default, the source language of the program determines the format used for the entry and display of data. You can also impose other formats. For example, the contents of a program location may be entered or displayed in ASCII, hexadecimal, octal, or decimal notation.

Starting and Resuming Program Execution

You start and resume program execution with the GO or STEP commands. The GO command causes the program to execute until a breakpoint is reached, a watchpoint is modified, an exception condition occurs, or the program terminates. The STEP command lets you execute a specified number of lines or instructions, or up to the next instruction of a specified class.

Breakpoints

By setting breakpoints with the SET BREAK command, you can suspend program execution at specified locations and check the current status of your program. Rather than specify a location, you can also suspend execution on certain classes of instructions or on every source line. Also you can suspend execution on certain types of events, such as exceptions and Ada tasking events.

Tracepoints

By setting tracepoints with the SET TRACE command, you can monitor the path of program execution through specified locations. When a tracepoint is triggered, the debugger reports that the tracepoint was reached and then continues execution. As with the SET BREAK command, you can also trace through classes of instructions and monitor events.

Watchpoints

By setting a watchpoint with the SET WATCH command, you can cause execution to stop whenever a particular variable or other memory area has been modified. When a watchpoint is triggered, the debugger suspends execution at that point and reports the old and new values of the variable.

Manipulation of Variables and Program Locations

With the EXAMINE command, you can determine the value of a variable or program location. The DEPOSIT command lets you change that value. You can then continue execution to see the effect of the change, without having to recompile, relink, and rerun the program.

Evaluation of Expressions

With the EVALUATE command, you can compute the value of a source-language expression or an address expression. You can specify expressions and operators in the syntax of the language to which the debugger is currently set.

Control Structures

You can use logical control structures (FOR, IF, REPEAT, WHILE) in commands to control the execution of other commands.

Shareable Image Debugging

You can debug shareable images (images that are not directly executable). The SET IMAGE command makes it possible for you to reference the symbols declared in a shareable image.

Terminal Support

The debugger supports all VT-series terminals and MicroVAX workstations.

1.1.2 Ease of Use Features

Online HELP

Online HELP is available during a debugging session. Typing HELP at the debugger prompt lists all of the debugger commands and selected topics for which detailed online HELP is available.

Source Code Display

In all supported languages except MACRO, the debugger lets you display lines of source code during a debugging session. For MACRO, the code stream of the executing routine is decoded into assembly language instructions.

Screen Mode

In screen mode, you can display and capture various kinds of information in scrollable windows that can be moved around the screen and resized. Automatically updated source, instruction, and register displays are available. You can selectively direct debugger input, output, and diagnostic messages to displays. You can also create "DO" displays that capture the output of specific command sequences.

Keypad Mode

When you invoke the debugger, several commonly used debugger command sequences are assigned by default to the keys of the numeric keypad. Thus, you can enter these commands with fewer keystrokes than if you were to type them at the keyboard. You can also create your own key definitions.

Source Editing

As you find errors during a debugging session, you can use the EDIT command to invoke any editor available on your system. You specify the editor you wish with the SET EDITOR command. If you use the VAX Language-Sensitive Editor, the editing cursor is automatically positioned within the source file whose code appears in the screen-mode source display.

Command Procedures

You can direct the debugger to execute a command procedure (a file of debugger commands) to recreate a debugging session, to continue a previous session, or to avoid typing the same debugger commands many times during a debugging session. You can pass parameters to command procedures.

Initialization Files

You can create an initialization file containing commands to set your default debugging modes, screen display definitions, keypad key definitions, symbol definitions, and so on. When you invoke the debugger, those commands will be executed automatically to tailor your debugging environment.

Log Files

You can record in a log file the commands you enter during a debugging session and the debugger's responses to those commands. You can use log files to keep track of your debugging efforts, or you can use them as command procedures in subsequent debugging sessions.

Symbol Definitions

You can define your own symbols to represent commands, address expressions, or values.

1.2 Getting Started

The way you use the debugger depends on several factors: the kind of program you are working on, the kinds of errors you are looking for, and your own personal style and experience with the debugger. This section explains the basic functions that apply to most situations. Additional details have been purposely left out so you can quickly get started.

The following subjects are covered:

- Starting and terminating a debugging session
- Entering debugger commands and getting online HELP
- Viewing your source code in screen mode and with the TYPE command
- Controlling program execution with the GO, STEP, and SET BREAK commands, and monitoring execution with the SHOW CALLS, SET TRACE, and SET WATCH commands
- Examining and manipulating data with the EXAMINE, DEPOSIT, and EVALUATE commands
- Controlling symbol references with path names and the SET MODULE and SET SCOPE commands

At the end of this section, a sample debugging session with a simple program illustrates how to locate an error and correct it.

Language-specific examples are in FORTRAN, COBOL, or Ada. However, the general concepts are readily adaptable to other languages.

1.2.1 Starting and Terminating a Debugging Session

To execute a program with the debugger, you first compile and link the program with the /DEBUG command qualifier. The following example shows how to use the /DEBUG qualifier with a COBOL program consisting of a single compilation unit named INVENTORY.

```
$ COBOL/DEBUG/NOOPTIMIZE INVENTORY
$ LINK/DEBUG INVENTORY
```

The /DEBUG qualifier on the compiler command (COBOL in this case) causes the compiler to write the symbol records associated with INVENTORY into the object module, INVENTORY.OBJ. These records will permit you to use, in debugger commands, the names of variables and other symbols declared in INVENTORY. (If your program has several compilation units, you need to compile each unit that you wish to debug with the /DEBUG qualifier).

Depending on the default behavior of the compiler command, the resulting object code may be optimized to reduce the size of the program and make it run faster. This may cause the contents of some program locations to be inconsistent with what you might expect from the source code. So, when using the debugger, it is best to compile your program with the /NOOPTIMIZE as well as /DEBUG qualifiers.

The /DEBUG qualifier on the LINK command causes the linker to include in the executable image all symbol information that is contained in INVENTORY.OBJ. The qualifier also causes the VAX/VMS image activator to start the debugger at run time. (Again, if your program has several object modules, you may need to specify other modules in the LINK command).

You can now invoke the debugger by issuing the DCL RUN command. The following example shows how the debugger identifies itself after you invoke it:

```
$ RUN INVENTORY

                    VAX DEBUG Version 4.4

%DEBUG-I-INITIAL, language is COBOL, module set to 'INVENTORY'
DBG>
```

The "INITIAL" message indicates that the debugger is initialized for a COBOL program and that the name of the main program is INVENTORY. The initialization sets up any language-dependent debugger parameters. The prompt (DBG>) indicates that you can now enter debugger commands.

To terminate a debugging session any time DBG> is displayed, type EXIT or press CTRL/Z:

```
DBG> EXIT
$
```

The dollar sign ($) prompt indicates that you are at DCL command level.

If you need to interrupt a debugging session for any reason, press CTRL/Y to return to DCL level. This may be necessary if, for example, your program loops or otherwise fails to complete execution, or if you need to interrupt a debugger command that is still in progress. From DCL level, if you then type the CONTINUE command, you return to where you interrupted the debugging session.

If you interrupted because of an infinite loop, type the DCL DEBUG command instead. This will return you to the debugger prompt so that you can enter another command. For example:

```
DBG> GO
   .
   .
   .
(infinite loop)
CTRL/Y
Interrupt
$ DEBUG
DBG>
```

The following message, displayed during a debugging session, indicates that your program has completed normally:

```
%DEBUG-I--EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>
```

1.2.2 Entering Debugger Commands

You can enter debugger commands any time you see the debugger prompt (DBG>). To enter a command, type it at the keyboard and press RETURN. You can enter several commands on a line by separating the command strings with semicolons (;). As with DCL commands, you can continue a command string on a new line by ending the line with a hyphen (-), and you can abbreviate debugger commands and qualifiers to unique characters. Also, you can use the up-arrow and down-arrow keys to recall commands lines, and the left-arrow and right-arrow keys to position the cursor for editing the command line.

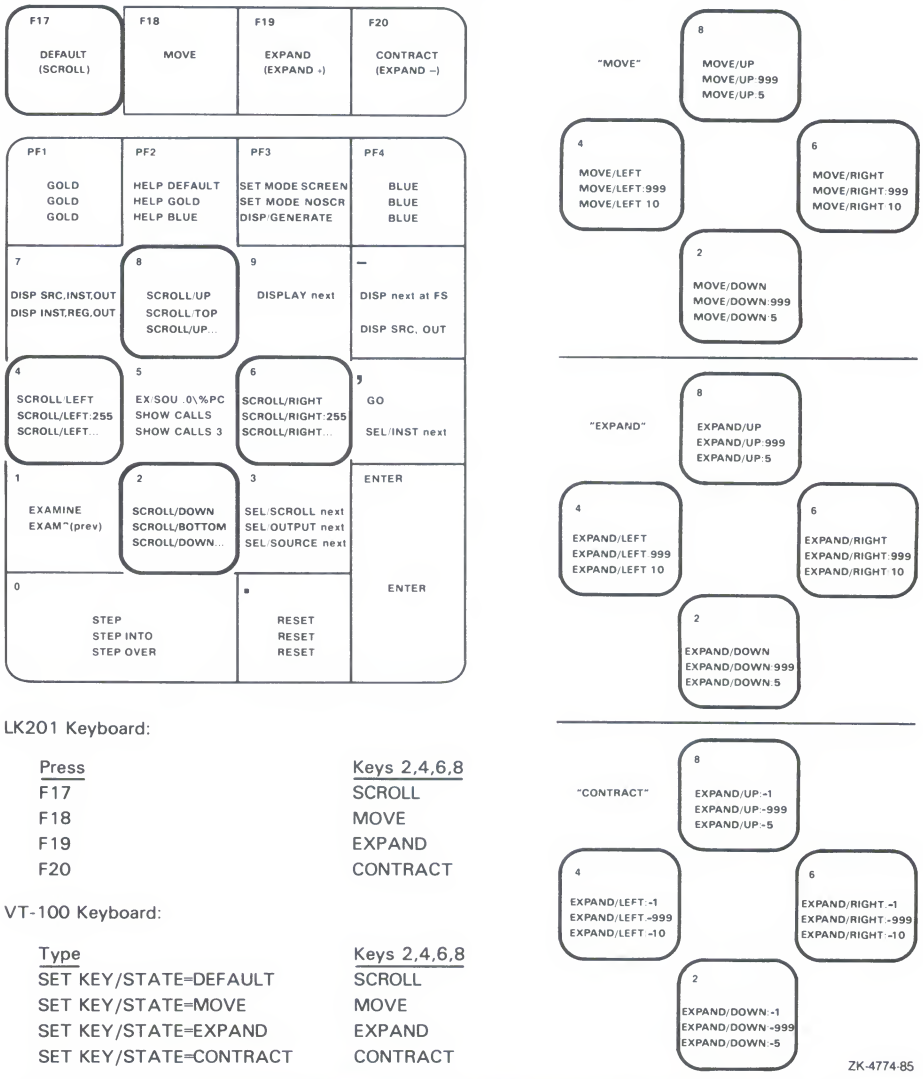
Typing HELP gives online HELP on debugger commands and selected topics. For example, if you type HELP STEP, help on the STEP command is displayed.

When you invoke the debugger, a few commonly used command sequences are automatically assigned to the keys on the numeric keypad (to the right of the main keyboard). By pressing keypad keys, you can enter these commands with fewer key strokes. The predefined key functions are identified in Figure 1–1. In addition to the STEP, GO, SHOW CALLS, and EXAMINE commands, several functions that manipulate screen-mode displays are bound to the keys. You can also redefine key functions with the DEFINE/KEY command.

Most keypad keys have three predefined functions—DEFAULT, GOLD, and BLUE. To obtain the DEFAULT function, simply press the key. To obtain the GOLD function, first press the PF1 key then the given key. To obtain the BLUE function, first press the PF4 key, then the given key. In Figure 1–1, the DEFAULT, GOLD, and BLUE functions are listed within each key's outline, from top to bottom respectively. For example, pressing keypad key 0 enters the command STEP (DEFAULT function); pressing key PF1 and then key 0 enters the command STEP/INTO (GOLD function); pressing key PF4 and then key 0 enters the command STEP/OVER (BLUE function).

Normally, keys 2, 4, 6, and 8 let you scroll screen displays down, left, right, or up, respectively. By putting the keypad in the MOVE, EXPAND, or CONTRACT state (as indicated in Figure 1–1), you can also use these keys to move, expand, or contract displays in four directions. Type HELP Keypad to get HELP on the keypad key definitions.

Figure 1–1 Keypad Key Functions Predefined by the Debugger



LK201 Keyboard:

Press	Keys 2,4,6,8
F17	SCROLL
F18	MOVE
F19	EXPAND
F20	CONTRACT

VT-100 Keyboard:

Type	Keys 2,4,6,8
SET KEY/STATE=DEFAULT	SCROLL
SET KEY/STATE=MOVE	MOVE
SET KEY/STATE=EXPAND	EXPAND
SET KEY/STATE=CONTRACT	CONTRACT

1.2.3 Viewing your Source Code

The easiest way to view your source code while debugging is to invoke screen mode, as described below. By default, when you invoke the debugger, you are in "noscreen" mode, which provides other techniques for viewing source code. Noscreen mode is described at the end of this section.

1.2.3.1 Screen Mode

You invoke screen mode by pressing keypad key PF3 (or by typing SET MODE SCREEN). In screen mode, by default the debugger splits the screen into three displays named SRC, OUT, and PROMPT, as illustrated below:

```
- SRC: module SWAP_ROUTINES -scroll-source-----------------
     2: with TEXT_IO; use TEXT_IO;
     3: package body SWAP_ROUTINES is
     4:    procedure SWAP1 (A,B: in out INTEGER) is
     5:       TEMP: INTEGER;
     6:    begin
     7:       TEMP := A;
->   8:       A := B;
     9:       B := TEMP;
    10:    end;
    11:
    12:    procedure SWAP2 (A,B: in out COLOR) is
    13:       TEMP: COLOR;
- OUT -output-----------------------------------------------
stepped to SWAP_ROUTINES.SWAP1.%LINE 8
SWAP_ROUTINES.SWAP1.A: 35



- PROMPT -error-program-prompt----------------------------
DBG> STEP
DBG> EXAMINE A
DBG>
```

The SRC display shows the source code of the module (compilation unit) that is currently executing. An arrow in the left column points to the source line corresponding to the current location of the program counter (PC). The line numbers, which are assigned by the compiler, match those in a listing file. As you execute the program, the arrow moves down and the source code is scrolled vertically to center the arrow in the display.

The OUT display captures debugger output from the commands that you enter. The PROMPT display shows the debugger prompt, your input, debugger diagnostic messages, and program output.

Both SRC and OUT are scrollable so you can see whatever information may scroll beyond the display window's edge. Use keypad key 3 to select the display to be scrolled. Use keypad key 8 to scroll up and keypad key 2 to scroll down. Scrolling a display does not affect program execution.

If the debugger cannot locate source lines for the currently executing module, it tries to display source lines in the next module down on the call stack for which source lines are available and issues the following message:

```
%DEBUG-I-SOURCESCOPE, Source lines not available for .0\%PC.
       Displaying source in a caller of the current routine.
```

Source lines may not be available for a variety of reasons:

- The PC is within a system routine, or a shareable image routine for which no source code is available.
- The PC is within a routine which was compiled without the /DEBUG compiler command qualifier (or with /NODEBUG).
- The source file was moved to a different directory after it was compiled (the location of source files is embedded in the object modules).

1.2.3.2 Noscreen Mode

"Noscreen" mode is the default, line-oriented mode of displaying input and output. To get into noscreen mode from screen mode, press the keypad key sequence GOLD-PF3 (or type SET MODE NOSCREEN). In that mode, you can use the TYPE command to display one or more source lines. For example, the following command displays line 7 of the module that is currently executing:

```
DBG> TYPE 7
7:          TEMP := A;
DBG>
```

The display of source lines is independent of program execution. You can use the TYPE command to display source code from a module other than the one currently executing. In that case, you need to use a "path name" to specify the module. For example, the following command displays lines 16 through 21 of module TEST:

```
DBG> TYPE TEST\16:21
```

Path names are discussed in more detail in the next section, in relation to the STEP command.

1.2.4 Controlling and Monitoring Program Execution

This section covers the following topics:

- Starting and resuming program execution with the GO command
- Stepping through the program's code with the STEP command
- Determining the current location of the program counter (PC) with the SHOW CALLS command
- Suspending program execution with breakpoints
- Tracing program execution with tracepoints
- Monitoring changes in variables with watchpoints

With this information you will be able to pick program locations where you can test and manipulate the contents of variables as described in Section 1.2.5.

1.2.4.1 Starting and Resuming Program Execution

There are two commands for starting or resuming program execution—GO and STEP. GO simply starts execution. STEP (discussed in the next section) lets you execute a specified number of source lines or instructions. You use one or the other command according to the circumstances.

One possible use of the GO command is immediately after you invoke the debugger. With most programming languages, execution is suspended directly at the start of the main program. For example (the main program is INVENTORY in this case):

```
$ RUN INVENTORY

                    VAX DEBUG Version 4.4

%DEBUG-I-INITIAL, language is COBOL, module set to 'INVENTORY'
DBG>
```

In some cases, however, the debugger suspends execution *before* the start of the main program, so that you can choose to execute some initialization code under debugger control. This occurs, for example, with all Ada programs and with FORTRAN programs compiled with the /CHECK=UNDERFLOW qualifier. The debugger indicates these cases with a "NOTATMAIN" message. For example:

```
$ RUN HOTEL

                    VAX DEBUG Version 4.4

%DEBUG-I-INITIAL, language is ADA, module set to 'HOTEL'
%DEBUG-I-NOTATMAIN, type GO to get to start of main program
DBG>
```

Typing GO will then get you to the start of the main program.

You will probably use the GO command most often in conjunction with breakpoints, tracepoints, and watchpoints (described later in this section). If you set a breakpoint in the path of execution and then type GO, execution will be suspended at that breakpoint. Similarly, if you set a tracepoint, execution will be monitored as it passes through that tracepoint. And if you set a watchpoint, execution will be suspended when the value of the "watched" variable changes.

The GO command is also useful if you want to test for an exception condition or an infinite loop. If an exception condition that is not handled by your program occurs, the debugger will take over and display the DBG> prompt so that you can issue commands. If you are using screen mode, the pointer in the source display will indicate where execution stopped. The SHOW CALLS command (also explained later in this section) is useful at this point because it identifies the currently active routine calls (the call stack).

In the case of an infinite loop, the program will not terminate, so the debugger prompt will not reappear. To obtain the prompt, interrupt the program with CTRL/Y and then issue the DCL DEBUG command. You can then look at the source display and a SHOW CALLS display to locate the PC.

1.2.4.2 Stepping Through the Program's Code

The STEP command is useful when you want to execute a specified number of source lines or instructions, or if you want to execute the program to the next instruction of a particular kind, for example to the next CALL instruction.

By default, the STEP command executes a single source line at a time. In the following example, the STEP command executes one line, reports the action ("stepped to . . . "), and displays the line number (27) and source code of the next line to be executed:

```
DBG> STEP
stepped to TEST\COUNT\%LINE 27
     27:    X := X + 1;
DBG>
```

The PC is now at the first machine code instruction for line 27 of module TEST; line 27 is in COUNT, a routine within module TEST.

When displaying a program symbol (for example, a line number, routine name, or variable name), the debugger always uses a *path name*. A path name consists of the symbol plus a prefix that identifies the symbol's location. In the preceding example, the path name is TEST\COUNT\%LINE 27. The leftmost element of a path name is the module name. Moving toward the right, the path name lists any successively nested routines and blocks that enclose the symbol. A backslash character (\) is used to separate elements

(except when the language is Ada, where a period is used, to parallel Ada syntax).

A path name uniquely identifies a symbol of your program to the debugger. But in general, you need to use path names in commands only if the debugger tells you that a symbol you have specified is not unique (see Section 1.2.6.2). (Usually the debugger can figure out the symbol you mean from its context).

Let us now return to the STEP command. You can specify a number of lines for the STEP command to execute. In the following example, the STEP command executes three lines:

```
DBG> STEP 3
```

Note that only those source lines for which code instructions were generated by the compiler are recognized as executable lines by the debugger. The debugger skips over any other lines—for example, comment lines.

You can specify different stepping modes, such as stepping by instruction rather than by line (SET STEP INSTRUCTION). Also, by default, the debugger steps "over" called routines—execution is not suspended within a called routine, although the routine is executed. By issuing the SET STEP INTO command, you tell the debugger to suspend execution within called routines as well as within the currently executing module. (SET STEP OVER is the default mode).

1.2.4.3 Determining the Current Location of the Program Counter

The SHOW CALLS command is useful when you are unsure about the current location of the PC (for example, after returning to the debugger following a CTRL/Y interrupt). The command shows a traceback that lists the sequence of calls leading to the currently executing routine. For example:

```
DBG> SHOW CALLS
 module name      routine name       line      rel PC     abs PC

*TEST             PRODUCT              18    00000009   0000063C
*TEST             COUNT                47    00000009   00000647
*MY_PROG          MY_PROG              21    0000000D   00000653
DBG>
```

For each routine (beginning with the currently executing routine), the debugger displays the following information: the name of the module that contains the routine, the name of the routine, the line number at which the call was made (or at which execution is suspended, in the case of the current routine), and the corresponding PC addresses (the relative PC address from the start of the routine, and the absolute PC address of the program). This example indicates that execution is currently at line 18 of routine PRODUCT (in module TEST), which was called from line 47 of routine COUNT (in module TEST), which was called from line 21 of routine MY_PROG (in module MY_PROG).

1.2.4.4 Suspending Program Execution

The SET BREAK command lets you select locations for program suspension (breakpoints), where you can issue commands to check the call stack, examine the current values of variables, and so on.

A typical use of the SET BREAK command is illustrated in the following example:

```
DBG> SET BREAK COUNT
DBG> GO
   .
   .
   .
break at PROG2\COUNT
     54:  procedure COUNT(X,Y:INTEGER);
DBG>
```

In the example, the SET BREAK command sets a breakpoint on routine COUNT (at the start of the routine's code); the GO command starts execution; when routine COUNT is encountered, execution is suspended, the debugger announces that the breakpoint at COUNT has been reached ("break at . . . "), displays the source line (54) where execution is suspended, and prompts for another command. At this breakpoint, you could STEP through routine COUNT and then use the EXAMINE command (discussed further on) to check on the values of X and Y.

When using the SET BREAK command, you can specify program locations using various kinds of *address expressions* (for example, line numbers, routine names, instructions, virtual memory addresses, byte offsets). With high level languages, you typically use routine names, labels, or line numbers, possibly with path names to ensure uniqueness.

Routine names and labels should be specified as they appear in the source code. Line numbers may be derived from either a source code display or a listing file. When specifying a line number, use the prefix %LINE. Otherwise the debugger will interpret the line number as a memory location. For example, the next command sets a breakpoint at line 41 of the currently executing module: the debugger will suspend execution when the PC is at the start of line 41.

```
DBG> SET BREAK %LINE 41
```

Note that you can set breakpoints only on lines that resulted in machine code instructions. The debugger warns you if you try to do otherwise (for example on a comment line). If you want to pick a line number in a module other than the one currently executing, you need to specify the module's name in a path name. For example:

```
DBG> SET BREAK SCREEN_IO\%LINE 58
```

You do not always have to specify a particular program location, such as line 58 or COUNT, to set a breakpoint. You can also use the SET BREAK command with a qualifier, but no parameter, to break on every line, or on every CALL instruction, and so on. For example:

```
DBG> SET BREAK/LINE
DBG> SET BREAK/CALL
```

Also, you can set breakpoints on *events*, such as exceptions, or state transitions in Ada tasking programs.

You can conditionalize a breakpoint (with a "WHEN" clause) or specify that a list of commands be executed at the breakpoint (with a "DO" clause). For example, the next command sets a breakpoint on the label LOOP3. The debugger DO clause displays the value of the variable TEMP whenever the breakpoint is triggered.

```
DBG> SET BREAK LOOP3 DO (EXAMINE TEMP)
DBG> GO
   .
   .
   .
break at COUNTER\LOOP3
     37:    LOOP3: FOR I = 1 TO 10 DO
COUNTER\TEMP:   284.19
DBG>
```

To display the currently active breakpoints, type SHOW BREAK:

```
DBG> SHOW BREAK
breakpoint at SCREEN_IO\%LINE 58
breakpoint at PROG2\LOOP3
   do (EXAMINE TEMP)
   .
   .
   .
DBG>
```

To cancel a breakpoint, type CANCEL BREAK, specifying the program location exactly as you did when setting the breakpoint. CANCEL BREAK /ALL cancels all breakpoints.

1.2.4.5 Tracing Program Execution

The SET TRACE command lets you select locations for tracing the execution of your program (tracepoints), without stopping its execution. After setting a tracepoint, you can start execution with the GO command and then monitor the PC's path, checking for unexpected behavior. By setting a tracepoint on a routine, you can also monitor the number of times it is called.

As with breakpoints, every time a tracepoint is reached, the debugger issues a message and displays the source line. But the program continues executing, and the debugger prompt is not displayed. For example:

```
DBG> SET TRACE COUNT
DBG> GO
   .
   .
   .
trace at PROG2\COUNT
     54:  procedure COUNT(X,Y:INTEGER);
   .
   .
   .
```

This is the only difference between a breakpoint and a tracepoint. When using the SET TRACE command, you specify address expressions, qualifiers, and optional clauses exactly as with the SET BREAK command.

The /LINE qualifier causes the SET TRACE command to trace every line and is a convenient means of checking the execution path. By default, lines are traced within all called routines as well as the currently executing routine. If you do not want to trace system routines or routines in shareable images, use the /NOSYSTEM or /NOSHARE qualifiers. For example:

```
DBG> SET TRACE/LINE/NOSYSTEM/NOSHARE
```

The /SILENT qualifier suppresses the trace message and source code display. This is useful when you want to use the SET TRACE command simply to execute a debugger command at the tracepoint. For example:

```
DBG> SET TRACE/SILENT %LINE 83 DO (EXAMINE STATUS)
DBG> GO
   .
   .
   .
SCREEN_IO\CLEAR\STATUS:  OFF
   .
   .
   .
```

1.2.4.6 Monitoring Changes in Variables

The SET WATCH command lets you specify program variables that the debugger will monitor as your program executes. This process is called setting watchpoints. If the program modifies the value of a "watched" variable, the debugger suspends execution and displays information. The debugger monitors watchpoints continuously during program execution. (Note that the SET WATCH command may also be used to monitor arbitrary program locations, not just variables).

To set a watchpoint on a variable, specify the variable's name with the SET WATCH command. For example, the following command sets a watchpoint on the variable TOTAL:

```
DBG> SET WATCH TOTAL
```

Subsequently, every time the program modifies the value of TOTAL, the watchpoint is triggered.

The next example shows what happens when your program modifies the contents of a watched variable.

```
DBG> SET WATCH TOTAL
DBG> GO
   .
   .
   .
watch of SCREEN_IO\TOTAL\%LINE 13
     13:   TOTAL := TOTAL + 1;
    old value: 16
    new value: 17
break at SCREEN_IO.%LINE 14
     14:   POP(TOTAL);
DBG>
```

In this example, a watchpoint is set on the variable TOTAL and execution is started. When the value of TOTAL changes, execution is suspended. The debugger announces the event ("watch of . . . "), identifying where TOTAL changed (the start of line 13) and the associated source line. The debugger then displays the old and new values and announces that execution has been suspended at the start of the next line (14). Finally, the debugger prompts for another command. Note that when a change in a variable occurs at a point other than the start of a source line, the debugger gives the line number plus the byte offset from the start of the line.

1.2.5 Examining and Manipulating Data

This section explains how to use the EXAMINE, DEPOSIT, and EVALUATE commands to display and modify the contents of variables and evaluate expressions.

1.2.5.1 Displaying the Values of Variables

To display the value of a variable, use the EXAMINE command as follows:

```
EXAMINE variable-name
```

The debugger recognizes the compiler-generated data type of the variable you specify and retrieves and formats the data accordingly. The following examples illustrate how to use the EXAMINE command.

Examine a string variable:

```
DBG> EXAMINE EMPLOYEE_NAME
PAYROLL\EMPLOYEE_NAME:    "Peter C. Lombardi"
DBG>
```

Examine three integer variables:

```
DBG> EXAMINE WIDTH, LENGTH, AREA
SIZE\WIDTH:   4
SIZE\LENGTH:  7
SIZE\AREA:   28
DBG>
```

Examine a two-dimensional array of real numbers (three per dimension):

```
DBG> EXAMINE REAL_ARRAY
PROG2\REAL_ARRAY
   (1,1):       27.01000
   (1,2):       31.00000
   (1,3):       12.48000
   (2,1):       15.08000
   (2,2):       22.30000
   (2,3):       18.73000
DBG>
```

Examine element 4 of a one-dimensional array of characters:

```
DBG> EXAMINE CHAR_ARRAY(4)
PROG2\CHAR_ARRAY(4): 'm'
DBG>
```

Examine a record variable (COBOL example):

```
DBG> EXAMINE PART
INVENTORY\PART:
    ITEM:     "WF-1247"
    PRICE:     49.95
    IN-STOCK:  24
DBG>
```

Examine a record component (COBOL example):

```
DBG> EXAMINE IN-STOCK OF PART
INVENTORY\IN-STOCK of PART:
    IN-STOCK:  24
DBG>
```

Note that the EXAMINE command may be used with any kind of address expression (not just a variable name) to display the contents of a program location. The debugger associates certain default data types with untyped locations. You can override the defaults for typed and untyped locations if you want the data to be interpreted and displayed in some other data format.

1.2.5.2 Changing the Values of Variables

To change the value of a variable, use the DEPOSIT command as follows:

```
DEPOSIT variable-name = value
```

The DEPOSIT command is like an assignment statement in most programming languages.

In the following examples, the DEPOSIT command assigns new values to different variables. The debugger checks that the value assigned, which may be a language expression, is consistent with the data type and dimensional constraints of the variable.

Deposit a string value (it must be enclosed in quote characters or apostrophes):

```
DBG> DEPOSIT PART-NUMBER = "WG-7619.3-84"
```

Deposit an integer expression:

```
DBG> DEPOSIT WIDTH = CURRENT_WIDTH + 10
```

Deposit element 12 of an array of characters (you cannot deposit an entire array aggregate with a single DEPOSIT command, only an element):

```
DBG> DEPOSIT C_ARRAY(12) := 'K'
```

Deposit a record component (you cannot deposit an entire record aggregate with a single DEPOSIT command, only a component):

```
DBG> DEPOSIT EMPLOYEE.ZIPCODE = 02172
```

Deposit an out-of-bounds value (X was declared as a positive integer):

```
DBG> DEPOSIT X = -14
%DEBUG-I-IVALOUTBNDS, value assigned is out of bounds at or near 'DEPOSIT'
```

As with the EXAMINE command, you can specify any kind of address expression (not just a variable name) with the DEPOSIT command. You can override the defaults for typed and untyped locations if you want the data to be interpreted in some other data format.

1.2.5.3 Evaluating Expressions

To evaluate a language expression, use the EVALUATE command as follows:

```
EVALUATE language-expression
```

The debugger recognizes the operators and expression syntax of the currently set language. In the following example, the value 45 is assigned to the integer variable WIDTH; the EVALUATE command then obtains the sum of the current value of WIDTH and 7:

```
DBG> DEPOSIT WIDTH := 45
DBG> EVALUATE WIDTH + 7
52
DBG>
```

In the next example, the values TRUE and FALSE are assigned to the boolean variables WILLING and ABLE, respectively; the EVALUATE command then obtains the logical conjunction of these values:

```
DBG> DEPOSIT WILLING := TRUE
DBG> DEPOSIT ABLE := FALSE
DBG> EVALUATE WILLING AND ABLE
False
DBG>
```

1.2.6 Controlling Symbol References

In most cases, the manner in which the debugger handles the symbols (variable names, and so on) that you reference in debugger commands should be transparent to you. However, two areas deserve mention because they may require some intervention on your part:

- Module setting
- Multiply-defined symbols

1.2.6.1 Module Setting

To facilitate symbol searches, the debugger loads symbol records from the executable image into a run-time symbol table (RST), where they can be accessed efficiently. Unless a symbol record is in the RST, the debugger cannot recognize or properly interpret that symbol.

Because the RST takes up memory, the debugger loads it dynamically, anticipating what symbols you might want to reference in the course of execution. The loading process is called *module setting*, because all of the symbol records of a given module are loaded into the RST at one time.

At debugger start up, only the module containing the image transfer address is set. As your program executes, whenever the debugger interrupts execution it sets the module surrounding the current PC location. This lets you reference the symbols that should be visible at the current PC location.

If you try to reference a symbol in a module that has not been set, the debugger will warn you that the symbol is not in the RST. For example:

```
DBG> EXAMINE K
%DEBUG-W-NOSYMBOL, symbol 'K' is not in symbol table
DBG>
```

You must then use the SET MODULE command to set the module containing that symbol manually:

```
DBG> SET MODULE MOD3
DBG> EXAMINE K
MOD3\ROUT2\K: 26
DBG>
```

The SHOW MODULE command lists the modules of your program and identifies which modules are set.

Note that dynamic module setting may slow the debugger down as more and more modules are set. If performance becomes a problem, you can use the CANCEL MODULE command to reduce the number of set modules, or you can disable dynamic module setting by issuing the SET MODE NODYNAMIC command (SET MODE DYNAMIC enables dynamic module setting).

1.2.6.2 Resolving Multiply-Defined Symbols

The debugger finds the symbols that you reference in commands according to the scope and visibility rules of the currently set language. In general, the debugger first looks for a symbol within the block or routine surrounding the current PC location; if the symbol is not found in that scope region, the debugger searches the nesting program unit, then its nesting unit, and so on. (The precise manner depends on the currently set language and guarantees that the proper declaration of a multiply-defined symbol is selected.)

The debugger must let you reference symbols throughout your program, not just those that are visible at the current PC location, to allow you to set breakpoints in arbitrary areas or examine arbitrary variables, and so on. Therefore, if the symbol is not visible at the current PC location, the debugger also searches other scope regions. First, it looks within the currently executing routine, then the caller of that routine, then its caller, and so on. Symbolically, this search list is denoted *0,1,2, . . . ,n*, where *n* is the number of calls in the call stack. Within each of these scope regions, the debugger uses the visibility rules of the language to locate a symbol. If the symbol is not found, the debugger then searches the rest of the run-time symbol table.

If the debugger cannot resolve a multiply-defined symbol, it issues a message. For example:

```
DBG> EXAMINE Y
%DEBUG-W-NOUNIQUE, symbol 'Y' is not unique
DBG>
```

In that case, you can use a path-name prefix to uniquely specify a declaration of the given symbol. First, use the SHOW SYMBOL command to identify all path names associated with the given symbol; then use the desired path name when referencing the symbol. For example:

```
DBG> SHOW SYMBOL Y
data MOD7\ROUT3\BLOCK1\Y
data MOD4\ROUT2\Y
DBG> EXAMINE MOD4\ROUT2\Y
MOD4\ROUT2\Y: 12
DBG>
```

If you need to refer to a particular declaration of Y repeatedly, use the SET SCOPE command to establish a new default scope for symbol lookup. Then, references to Y without a path-name prefix will specify the declaration of Y that is visible in the new scope region. For example:

```
DBG> SET SCOPE MOD4\ROUT2
DBG> EXAMINE Y
MOD4\ROUT2\Y: 12
DBG>
```

To display the current scope for symbol lookup, use the SHOW SCOPE command. To restore the default scope, use the CANCEL SCOPE command.

1.2.7 A Sample Debugging Session

This section goes through a debugging session with a simple FORTRAN program that contains a logic error:

```
 1:       INTEGER INARR(20), OUTARR(20)
 2: C
 3: C     ---Read the input array from the data file.
 4:       OPEN(UNIT=8, FILE='DATAFILE.DAT', STATUS='OLD')
 5:       READ(8,*) N, (INARR(I), I=1,N)
 6: C
 7: C     ---Square all non-zero elements and store in OUTARR.
 8:       K = 0
 9:       DO 10 I = 1, N
10:       IF(INARR(I) .NE. 0) THEN
11:           OUTARR(K) = INARR(I)**2
12:       ENDIF
13:    10 CONTINUE
14: C
15: C     ---Print the squared output values.  Then stop.
16:       PRINT 20, K
17:    20 FORMAT(' Number of non-zero elements is',I4)
18:       DO 40 I = 1, K
19:       PRINT 30, I, OUTARR(I)
20:    30 FORMAT(' Element',I4,' has value',I6)
21:    40 CONTINUE
22:       END
```

As you read on, you can refer back to this code to identify source lines. The program reads a sequence of integer numbers from a data file (lines 4 and 5) and saves these numbers in the array INARR. The program then enters a loop (lines 8 through 13) where it copies the square of each nonzero integer into another array OUTARR. Finally, it prints the number of nonzero elements in the original sequence and the square of each such element (lines 16 through 21).

The error in the program occurs when variable K, which keeps track of the current index into OUTARR, is not incremented in the loop on lines 9 through 13. The statement K = K + 1 should be inserted just before line 11.

To find this error, first compile, link, and run the program:

```
$ FORTRAN/DEBUG/NOOPTIMIZE SQUARES
$ LINK/DEBUG SQUARES
$ RUN SQUARES

         VAX DEBUG Version 4.4

%DEBUG-I-INITIAL, language is FORTRAN, module set to 'SQUARES$MAIN'
DBG>
```

You can now enter debugger commands. To step forward 4 lines, enter this command:

```
DBG> STEP 4
stepped to SQUARES$MAIN\%LINE 9
DBG>
```

To check the current values of variables N and K, enter this command:

```
DBG> EXAM N, K
SQUARES$MAIN\N:       9
SQUARES$MAIN\K:       0
DBG>
```

The values of N and K are both correct at this point in the execution. Now enter the command STEP 2 to enter the loop that copies and squares all nonzero elements of INARR into OUTARR.

```
DBG> STEP 2
stepped to SQUARES$MAIN\%LINE 11
DBG>
```

To see if I and K have the expected values, enter EXAM I, K.

```
DBG> EXAM I,K
SQUARES$MAIN\I:       1
SQUARES$MAIN\K:       0
DBG>
```

I has the expected value (namely 1), but K has the value zero, which is not the expected value. Now you can see the error in the program: K should be incremented in the loop just before it is used in line 11. To check this hypothesis, "repair" the program with the following debugger commands:

```
DBG> DEPOSIT K = 1
DBG> SET TRACE/SILENT %LINE 11 DO(DEPOSIT K = K + 1)
DBG>
```

The first command gives K the value it should have now, namely 1. The second command specifies that the debugger should perform the debugger command DEPOSIT K = K + 1 each time line 11 is reached in the execution. The /SILENT qualifier suppresses the "trace at" message that would otherwise be announced each time line 11 is executed. The net effect is that the program has been "patched" to perform correctly by the SET TRACE command.

Line 22 is a suitable location for a breakpoint that will stop program execution after testing the correctness of your "patch." Set a breakpoint as follows:

```
DBG> SET BREAK %LINE 22
DBG>
```

You now want to run your program to test your patch. Enter GO to execute the program until it hits the breakpoint at line 22.

```
DBG> GO
Number of non-zero elements is   6
Element   1 has value    16
Element   2 has value    36
Element   3 has value     9
Element   4 has value    49
Element   5 has value    81
Element   6 has value     1

break at SQUARES$MAIN\%LINE 22
     22:        END
DBG>
```

The program output shows that the program seems to work properly with the DEPOSIT K = K + 1 patch. You can now use the EDIT command to invoke the VAX Language Sensitive Editor, or another editor previously established with the SET EDITOR command:

```
DBG> EDIT
```

The editor positions the cursor at the same line that is marked by the pointer in the debugger's source display.

The corrected portion of the source code follows.

```
    .
    .
    .
 8:       K = 0
 9:       DO 10 I = 1, N
10:       IF(INARR(I) .NE. 0) THEN
11:           K = K + 1
12:           OUTARR(K) = INARR(I)**2
13:       ENDIF
14:    10 CONTINUE
    .
    .
    .
```

Now you can compile, link, and run the program again under debugger control, to check that it behaves correctly:

```
$ FORTRAN/DEBUG/NOPTIMIZE SQUARES
$ LINK/DEBUG SQUARES
$ RUN SQUARES
```

To set a breakpoint at line 12 where the values of I and K will be displayed automatically, enter this command (the subsequent GO command starts execution):

```
DBG> SET BREAK %LINE 12 DO (EXAMINE I,K)
DBG> GO
    .
    .
    .
SQUARES$MAIN\I:         1
SQUARES$MAIN\K:         1
DBG> GO
    .
    .
    .
SQUARES$MAIN\I:         2
SQUARES$MAIN\K:         2
DBG> GO
    .
    .
    .
SQUARES$MAIN\I:         4
SQUARES$MAIN\K:         3
```

At the first breakpoint, the value of K is 1, indicating that the program is behaving correctly so far. Each additional GO command will show the current values of I and K. After two GO commands, K is now 3, as expected, but note that I is 4. This indicates that one of the INARR elements was zero so that lines 11 and 12 were not executed (and K not incremented) on one iteration of the DO loop. This confirms that the program is behaving correctly.

1.3 Debugger Command Summary

The following sections list all of the debugger commands and any related DCL commands in functional groupings, along with brief descriptions.

1.3.1 Starting and Terminating a Debugging Session

($) RUN[1]	Invokes the debugger if LINK/DEBUG was used
($) RUN/[NO]DEBUG[1]	Controls whether the debugger is invoked when the program is executed
EXIT, CTRL/Z	Ends a debugging session, executing all exit handlers
QUIT	Ends a debugging session without executing any exit handlers declared in the program
CTRL/Y	Interrupts a debugging session, returning you to DCL level
CTRL/C	Like CTRL/Y, unless the program has a CTRL/C service routine
($) CONTINUE[1]	Resumes a debugging session after CTRL/Y interruption
($) DEBUG[1]	Resumes a debugging session after CTRL/Y interruption but returns you to the debugger prompt
ATTACH	Passes control of your terminal from the current process to another process (like $ ATTACH)
SPAWN	Creates a subprocess. Lets you issue DCL commands without interrupting your debugging context (like $ SPAWN)

[1]This is a DCL command, not a debugger command.

1.3.2 Controlling and Monitoring Program Execution

GO	Starts or resumes program execution
STEP	Executes the program up to the next line, instruction, or specified instruction
(SET,SHOW) STEP	(Establishes, displays) the default qualifiers for the STEP command
(SET,SHOW,CANCEL) BREAK	(Sets, displays, cancels) breakpoints
(SET,SHOW,CANCEL) TRACE	(Sets, displays, cancels) tracepoints
(SET,SHOW,CANCEL) WATCH	(Sets, displays, cancels) watchpoints
(SET,CANCEL) EXCEPTION BREAK	(Sets, cancels) exception breakpoints
SHOW CALLS	Identifies the currently active routine calls

SHOW STACK	Gives additional information about the currently active routine calls
CALL	Calls a routine

1.3.3 Examining and Manipulating Data

EXAMINE	Displays the value of a variable or the contents of a program location
DEPOSIT	Changes the value of a variable or the contents of a program location
EVALUATE	Evaluates a language or address expression

1.3.4 Controlling Type Selection and Symbolization

(SET,SHOW,CANCEL) RADIX	(Establishes, displays, restores) the radix for data entry and display
(SET,SHOW,CANCEL) TYPE	(Establishes, displays, restores) the type to be associated with untyped program locations
SET MODE [NO]G_FLOAT	Controls whether double-precision floating-point constants are interpreted as G_FLOAT or D_FLOAT
SET MODE [NO]LINE	Controls whether code locations are displayed in terms of line numbers or routine-name + byte offset
SET MODE [NO]SYMBOLIC	Controls whether code locations are displayed symbolically or in terms of numeric addresses
SYMBOLIZE	Converts a virtual address to a symbolic address

1.3.5 Controlling Symbol Lookup

SHOW SYMBOL	Displays symbols in your program
(SET,SHOW,CANCEL) MODULE	"Sets" a module by loading its symbol records into the debugger's symbol table, identifies, cancels a "set" module
(SET,SHOW,CANCEL) IMAGE	"Sets" a shareable image by loading data structures into the debugger's symbol table, identifies, cancels a "set" image
SET MODE [NO]DYNAMIC	Controls whether or not modules are "set" automatically when the debugger interrupts execution
ALLOCATE	Expands the debugger's memory pool to let you "set" more modules
(SET,SHOW,CANCEL) SCOPE	(Establishes, displays, restores) the scope for symbol lookup

1.3.6 Displaying Source Code

TYPE	Displays lines of source code
EXAMINE/SOURCE	Displays the source code at the location specified by the address expression
(SET,SHOW,CANCEL) SOURCE	(Creates, displays, cancels) a source directory search list
SEARCH	Searches the source code for the specified string
(SET,SHOW) SEARCH	(Establishes, displays) the default qualifiers for the SEARCH command
(SET,SHOW) MAX_SOURCE_ FILES	(Establishes, displays) the maximum number of source files that may be kept open at one time
(SET,SHOW) MARGINS	(Establishes, displays) the left and right margin settings for displaying source code

1.3.7 Screen Mode

SET MODE [NO]SCREEN	Enables/disables screen mode
SET MODE [NO]SCROLL	Controls whether an output display is updated line by line or once per command
DISPLAY	Modifies an existing display
(SET,SHOW,CANCEL) DISPLAY	(Creates, identifies, deletes) a display
(SET,SHOW,CANCEL) WINDOW	(Creates, identifies, deletes) a window definition
SELECT	Selects a display for a display attribute
SHOW SELECT	Identifies the displays selected for each of the display attributes
SCROLL	Scrolls a display
SAVE	Saves the current contents of a display into another display
EXTRACT	Saves a display or the current screen state into a file
EXPAND	Expands or contracts a display
MOVE	Moves a display across the screen
(SET,SHOW) TERMINAL	(Establishes, displays) the height and width of the screen
CTRL/W,DISPLAY/REFRESH	Refreshes the screen

1.3.8 Source Editing

EDIT	Invokes an editor during a debugging session
(SET,SHOW) EDITOR	(Establishes, identifies) the editor invoked by the EDIT command

1.3.9 Defining Symbols

DEFINE	Defines a symbol as an address, command, or value
DELETE (UNDEFINE)	Deletes symbol definitions
(SET,SHOW) DEFINE	(Establishes, displays) the default qualifier for the DEFINE command
SHOW SYMBOL/DEFINED	Identifies symbols that have been defined

1.3.10 Keypad Mode

SET MODE [NO]KEYPAD	Enables/disables keypad mode
DEFINE/KEY	Creates key definitions
DELETE/KEY (UNDEFINE/KEY)	Deletes key definitions
SET KEY	Establishes the key definition state
SHOW KEY	Displays key definitions

1.3.11 Command Procedures and Log Files

DECLARE	Defines parameters to be passed to command procedures
(SET,SHOW) LOG	(Specifies, identifies) the debugger log file
SET OUTPUT [NO]LOG	Controls whether a debugging session is logged
SET OUTPUT [NO]SCREEN_LOG	Controls whether, in screen mode, the screen contents are logged as the screen is updated
SET OUTPUT [NO]VERIFY	Controls whether debugger commands are displayed as a command procedure is executed
SHOW OUTPUT	Displays the current output options established by the SET OUTPUT command
(SET,SHOW) ATSIGN	(Establishes, displays) the default file specification that the debugger uses to search for command procedures
@file-spec	Executes a command procedure

1.3.12 Control Structures

IF	Executes a list of commands conditionally
FOR	Executes a list of commands repetitively
REPEAT	Executes a list of commands repetitively

WHILE	Executes a list of commands conditionally
EXITLOOP	Exits an enclosing WHILE, REPEAT, or FOR loop

1.3.13 Debugging Special Cases

SET OUTPUT [NO]TERMINAL	Controls whether debugger output, except for diagnostic messages, is displayed or suppressed
(SET,SHOW) LANGUAGE	(Establishes, displays) the current language
(SET,SHOW) EVENT_FACILITY	(Establishes, identifies) the current run-time facility for language-specific events
SHOW EXIT_HANDLERS	Identifies the exit handlers declared in the program
(SET,SHOW) TASK	Modifies the tasking environment, displays task information
(DISABLE,ENABLE,SHOW) AST	(Disables, enables) the delivery of ASTs in the program, identifies whether delivery is enabled or disabled

2 Controlling the Debugging Environment

The following subjects are discussed in this chapter:

- Controlling what symbolic information is available to you during a debugging session.
- Options for running your program.
- Debugger activation.
- Debugger initialization files.
- Language-dependent and -independent parameters.
- Log files.
- Command procedures.
- Interrupting a debugging session.

2.1 Controlling Symbol Information

When you run your program under debugger control, you can debug your program symbolically, using the symbols you used in your source program.

Symbolic debugging is possible because the debugger uses a symbol table which it stores in the executable image file (EXE) that you run. Part of the symbol table—the debugger symbol table (DST)—is created by the compiler or the assembler. Another part—the global symbol table (GST)—is created by the linker. When you specify the /DEBUG command qualifier on both the compile and the LINK commands, you are controlling the creation of these symbol tables.

In general, if you intend to debug your program, you should both compile and link your program using the /DEBUG command qualifier. The symbol table information generated by the compiler is processed by the linker, so you can obtain the full range of symbol table information at run time.

The example below shows how to start a debugging session during which you can use all the symbolic information in the program MEANSUB.

```
$ BASIC/DEBUG MEANSUB
$ LINK/DEBUG MEANSUB
$ RUN MEANSUB

                    VAX-11 DEBUG Version *****

%DEBUG-I-INITIAL, language is BASIC, module set to 'MEANSUB$MAIN'
DBG>
```

Note that you do not have to specify RUN/DEBUG if you specify LINK /DEBUG. If, however, you specified the /DEBUG command qualifier when you linked your program but you want your program to run without debugger control, you can specify the /NODEBUG command qualifier at run time.

Once you have debugged your program, you may want to recompile and relink it without creating symbol table information. In this way, you can reduce the size of both your object (OBJ) and image (EXE) files.

2.1.1 Compiling with the /DEBUG and the /TRACE Command Qualifiers

You can specify the /DEBUG command qualifier on a compile command, the LINK command, and the RUN command.

When you specify the /DEBUG qualifier on a compile command, the compiler then creates a symbol table (the DST) containing information both about the names you used in your program as well as about line numbers. For this reason, you should always use the /DEBUG qualifier when you compile any program that you intend to debug.

If you do not specify the /DEBUG command qualifier on the compile command, information about names of local variables are not included in the symbol table. However, the compiler still includes information about routine names and line numbers so it can give a symbolic traceback if your program terminates with a severe error.

If you do not want any symbol table (including the traceback information) to be created, you can specify the /DEBUG=NONE qualifier on the compiler command. For language-specific details, refer to the user's guide for the language you are using.

2.1.2 Linking with the /DEBUG and the /TRACE Command Qualifiers

When you specify the /DEBUG command qualifier with the LINK command, the linker creates a global symbol table (the GST) with the names of all your global data. The linker includes all the symbol tables that have been generated by the compiler. In addition, the linker sets a flag in the image file which causes your program to run under debugger control when it is executed.

If you do not specify the /DEBUG qualifier with the LINK command, then the linker includes only the symbol table information that is necessary for a symbolic traceback. The linker also does not set the flag in the image file that indicates debugger control.

You can direct the linker to generate no symbol tables (including the traceback information) by specifying the /NOTRACE qualifier with the LINK command. As a result, you cannot debug your program using the debugger.

2.2 Options for Running Your Program

Use of the DCL commands RUN/[NO]DEBUG and DEBUG are explained in this section.

2.2.1 Using the RUN/[NO]DEBUG Command

You can use the /DEBUG and /NODEBUG command qualifiers on the RUN command to control whether your program runs under debugger control.

If you did not specify the /DEBUG qualifier when you linked your program, then you must specify RUN/DEBUG if you want the debugger to receive control. In this case, however, you cannot debug your program using local symbols.

If you linked your program with the /DEBUG command qualifier, you can specify the /NODEBUG qualifier if you do not want your program to run under debugger control.

Table 2–1 summarizes what symbolic information is available in the executable image when you specify the /DEBUG command qualifier with the compiler command, the LINK command, and the RUN command.

Table 2–1 The /DEBUG Qualifier and Symbolic Debugging

Compile (or Assembly) Command Qualifier	LINK Command Qualifier	RUN Command Qualifier	Symbolic Information
None	None	/DEBUG	Traceback
/DEBUG or /DEBUG=ALL	None	/DEBUG	Traceback
/NODEBUG or /DEBUG=NONE	None	/DEBUG	None
/DEBUG=TRACEBACK or =(TRACEBACK,NOSYMBOLS)	None	/DEBUG	Traceback
None	/DEBUG	None	Traceback global symbols
/DEBUG or /DEBUG=ALL	/DEBUG	None	Local symbols global symbols traceback
/NODEBUG or /DEBUG=NONE	/DEBUG	None	Global symbols
/DEBUG=TRACEBACK or =(TRACEBACK,NOSYMBOLS)	/DEBUG	None	Traceback global symbols

2.2.2 Using the DEBUG Command

If your program is running without debugger control and you want to invoke the debugger, you must

1. Interrupt the running program by entering CTRL/Y
2. Issue the DEBUG command

After you enter the DEBUG command, the debugger displays an informational message and the DBG> prompt, indicating that it is ready to accept a debugger command.

To use this feature, you must, as a minimum, have linked your program with the /TRACE qualifier. To reference your program's symbols, you must have linked with the /DEBUG qualifier. Note that, under these conditions, you must then use the DCL command RUN/NODEBUG to execute the program without the debugger.

You will find the DEBUG command particularly useful when

- Your program is in an infinite loop. After interrupting your program and entering the DEBUG command, you can debug your program to find the cause of the infinite loop.
- You have entered the RUN/NODEBUG command but later decide that you want debugger control.
- You have not specified the /DEBUG command qualifier at compile time, link time, or run time but want to debug your running program. Note that in this situation, most symbolic information is unavailable to the debugger; you must use virtual memory addresses and not symbols to refer to program and data locations.

After you interrupt your running program and enter the DEBUG command, you will not know at which instruction your program was interrupted. To find out, issue the SHOW CALLS command.

Note that you may enter certain DCL commands after you interrupt your program and still be able to start the debugger by subsequently entering the DEBUG command. See Section 2.7.2 for more information.

The following example demonstrates how to interrupt your program and start the debugger using the DEBUG command. Note that CTRL/Y is echoed as "Interrupt" in reverse video.

```
$ RUN PROG
   .
   .
   .
CTRL/Y
Interrupt
$ DEBUG

                    VAX-11 DEBUG Version *****

%DEBUG-I-INITIAL, language is COBOL, module set to 'INVENTORY'
DBG>
```

2.3 Debugger Activation

The debugger is a shareable image that the VAX/VMS operating system maps into the address space of a user process when SYS$IMGSTA is the first entry in the transfer address array. SYS$IMGSTA is the first entry when you specify any one of the following combinations of DCL commands in the process of program development:

- LINK/DEBUG and not RUN/NODEBUG
- RUN/DEBUG (unless LINK/NOTRACEBACK)
- DEBUG following program interruption (unless LINK/NOTRACEBACK)

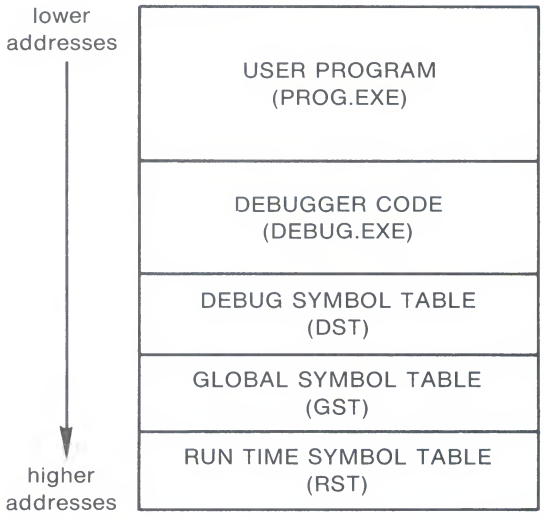
When the /NOTRACEBACK qualifier is specified at link time, the linker does not generate a debug symbol table (DST). Since a DST must be available at run time in order for the debugger to execute, specifying /NOTRACEBACK at link time inhibits the debugging of that image regardless of what other qualifiers are or are not specified at link or run time.

At run time, then, given any of the previously mentioned combinations of DCL commands, SYS$IMGSTA, a VAX/VMS component maps the debugger (that is, the shareable image DEBUG.EXE) into the highest-addressed portion of P0 space adjacent to the user image and passes control to the debugger.

When the debugger receives control, it initializes a portion of the address space adjacent to the debugger image with three symbol tables—the DST, the GST, and the RST. See Sections 4.2.1, 4.2.2, and 4.2.3 for a description of these symbol tables and how the debugger uses them.

Figure 2–1 depicts the layout of the virtual address space of a process that is running a program under debugger control.

Figure 2–1 Process Address Space Layout



ZK-573-81

2.4 Debugger Initialization Files

Debugger initialization files contain commands you always issue after you enter the debugger. Then, every time you invoke the debugger, those commands are automatically executed.

For example, you might have a file DEBUG.INI containing the following commands:

```
SET SOURCE DISK$
SET STEP LINE, SOURCE
SET MODULE/ALL
SET OUTPUT LOG,VER
SET LOG LINK.LOG
DEFINE/COMMAND W = "@WALK_LINK_LIST.COM"
```

To make this file a debug initialization file, use the DCL command DEFINE:

```
$ DEFINE DBG$INIT [JONES.DBGDIR]DEBUG.INI
```

The debugger will execute the commands in this file before it issues the DBG> prompt, as follows:

```
                    VAX-11 DEBUG Version *****

%DEBUG-I-INITIAL, language is BASIC, module set to 'MEANSUB$MAIN'
 SET LOG LINK.LOG
 DEFINE/COMMAND W = "@WALK_LINK_LIST.COM"
%DEBUG-I-VERIFYICF, exiting indirect command file "DBG$INIT"
DBG>
```

2.5 Language-Dependent and Independent Parameters

At start up, the debugger displays an informational message in the following format:

```
                VAX-11 DEBUG       Version Number

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

The first line of this message identifies the debugger's software version number. The second line contains initialization information that identifies the current language and program module. The third line contains the debugger prompt DBG> .

2.5.1 Language-Dependent Debugging Parameters

Debugging parameters influence how the debugger interprets commands and how it displays the results of command execution.

At start up, the debugger sets a default value for each debugging parameter. These values may be displayed using the appropriate SHOW command, may be altered using the appropriate SET command, and, in some cases, may be canceled using the appropriate CANCEL command.

The default values of some debugging parameters may vary, depending on the current language; that is, a particular parameter may have one default value if the current language is BASIC and another default value if the current language is MACRO. Such parameters are called language-dependent debugging parameters.

The following parameters are language-dependent debugging parameters:

- MODE
- RADIX
- STEP
- TYPE

When you change the current language by the SET LANGUAGE command, the default values for language-dependent debugging parameters may change.

To find out what the current language is, issue the SHOW LANGUAGE command.

2.5.2 Language-Independent Debugging Parameters

At start up, the default values of two debugging parameters are established independently of the current language setting:

- MODULE
- SCOPE

The module parameter allows for the insertion of a module's symbol information into the run-time symbol table (RST). At start up, the debugger inserts, into the RST, symbol records for the module containing the transfer address. The name of this module and the name of the language in which the module is written appears in the debugger informational message.

The transfer address is the location within the compiled code specified by the keyword(s) in the source language that signal the beginning of a program. For example, in VAX PASCAL the transfer address is the location within the compiled code specified by the keyword PROGRAM.

The scope parameter influences the debugger's interpretation of symbols.

When you start the debugger with the RUN command, the value of the scope parameter is the module that contains the transfer address. When you start the debugger with the DEBUG command, the value of the scope parameter is the module in which program execution was interrupted.

Chapter 4 contains a full description of how the module and scope parameters affect the debugger's interpretation of symbols.

You can direct the debugger to keep a record of a debugging session by creating a log file. You can also direct the debugger to execute debugger commands listed in an external file. Such a file, when executed, is called a command procedure. Further, you can direct the debugger to execute a log file as a command procedure. This section discusses these topics.

2.6 Log Files

A debugger log file is a file containing every debugger command you issue during a particular debugging session, together with a display of the debugger's response to those commands.

In a debugger log file, any command you issue in response to the debugger prompt (DBG>) begins a line, but the debugger prompt itself is not recorded. The debugger's response to the command appears on the following line or lines and, except for an exclamation point at the beginning of each line of response, is identical to what you see at your terminal.

The following is an example of a debugger log file. Note the absence of debugger prompts and the presence of exclamation points at the beginning of each line of debugger response.

```
SHOW OUTPUT
!output: noverify, terminal, logging to _DB2:[GMF]EV.LOG;1
SHOW TRACE
!%DEBUG-I-NOTRACES, no tracepoints are set, no opcode tracing
SET TRACE %LINE 30
SET BREAK %LINE 60
SHOW TRACE
```

```
!tracepoint at MEANSUB$MAIN %LINE 30
GO
!routine start at MEANSUB$MAIN
!trace at MEANSUB$MAIN\%LINE 30
!break at MEANSUB$MAIN\%LINE 60

SET TRACE %LINE 40
SHOW TRACE
!tracepoint at MEANSUB$MAIN\%LINE 40
!tracepoint at MEANSUB$MAIN\%LINE 30

GO %LINE 20
!start at MEANSUB$MAIN\%LINE 30
!trace at MEANSUB$MAIN\%LINE 30
!trace at MEANSUB$MAIN\%LINE 40
!break at MEANSUB$MAIN\%LINE 60
```

To create a debugger log file, use the SET OUTPUT command. To name a debugger log file, use the SET LOG command. The following subsections discuss these commands.

2.6.1 The SET OUTPUT and SHOW OUTPUT Commands

The SET OUTPUT command controls the debugger's current output configuration; that is, it controls the way in which the debugger's responses to commands you issue are displayed and recorded.

The format of the SET OUTPUT command is

```
SET OUTPUT parameter [,parameter...]
```

The following is a list of output parameters that you may specify in the SET OUTPUT command.

LOG	Writes debugger output to a log file. If you have not used the SET LOG command to name a log file, the debugger uses the default file name DEBUG.LOG.
NOLOG	Does not write debugger output to a log file. This output parameter is a default; the debugger does not write output to a log file unless you explicitly request it.
TERMINAL	Displays debugger output at the terminal. This output parameter is a default; the debugger displays output on the terminal unless you explicitly request it not to do so.
NOTERMINAL	Does not display debugger output, except for diagnostic messages, at the terminal.
VERIFY	Causes the debugger to include in its output each input command string in any command procedure or DO clause that it is executing; that is, the debugger not only displays the results of executing each command in a command procedure or DO clause but also displays the commands that caused execution.
NOVERIFY	Causes the debugger not to include as output each input command string that it executes from a command procedure or from a DO clause. This output parameter is a default; the debugger does not display commands that it executes from command procedures or DO clauses unless you explicitly direct it to do so.

To see which output parameters are currently in effect, issue the SHOW OUTPUT command.

2.6.2 The SET LOG and SHOW LOG Commands

If the output parameter LOG is in effect (by SET OUTPUT LOG), you can direct the debugger to write output to a specified file by issuing the SET LOG command in the following format:

```
SET LOG file-spec
```

The file-spec parameter is the file specification of the file to which you want the debugger to write its output.

If the output parameter LOG is in effect but you have not issued the SET LOG command to name a log file, the debugger writes output to a file with the default name DEBUG.LOG.

If the output parameter LOG is not in effect, the debugger does not write its output to the file specified in the SET LOG command. However, if you subsequently issue a SET OUTPUT LOG command, the debugger begins writing output to the file specified in the SET LOG command.

If the file-spec parameter in the SET LOG command includes both a file name and a file type, the debugger writes to the file so specified.

If the file-spec parameter in the SET LOG command does not include a file type, the debugger uses the default file type of LOG.

To find out the name of the current log file, issue the command SHOW LOG.

Example 2–1 demonstrates the SET LOG, SHOW LOG, SET OUTPUT, and SHOW OUTPUT commands.

Example 2–1 Using the SET/SHOW LOG and SET/SHOW OUTPUT Commands

```
DBG>SET LOG FILE                          !Name a log file.

DBG>SET OUTPUT LOG                        !Write output to the log
                                          !file.

DBG>SHOW OUTPUT
output: noverify, terminal, logging to _DB2:[GMF]FILE.LOG;1
                                          !Display current output
                                          !configuration.

DBG>SHOW LOG                              !Display name of current
logging to _DB2:[GMF]FILE.LOG;1           !log file.

DBG>SET OUTPUT NOTERMINAL,NOLOG
%DEBUG-I-OUTPUTLOST, output being lost, both NOTERMINAL and NOLOG
are in effect

DBG>SHOW LOG
not logging to _DB2:[GMF]FILE.LOG;1

DBG>SET LOG TERT.EEE                      !Name a new log file.

DBG>SHOW LOG                              !Not logging to TERT.EEE
not logging to _DB2:[GMF]TERT.EEE;1       !because output is set
                                          !to NOLOG.

DBG>SET OUTPUT LOG                        )!Change current output)
                                          !configuration.

DBG>SHOW LOG
logging to _DB2:[GMF]TERT.EEE;1

output: noverify, noterminal, logging to _DB2:[GMF]TERT.EEE;1
                                          !Display current output
                                          !configuration.
```

2.7 Command Procedures

A command procedure is a file containing one or more debugger commands. You can cause the debugger to execute debugger commands from a file by prefixing the file specification of that file with the at sign (@), in the following format:

```
@file-spec
```

If the file specification includes a file name but not a file type, the debugger assumes the default file type COM.

The @file-spec command may be issued from the terminal, from within a DO clause, or from within another command procedure.

If the @file-spec command is issued from the terminal, the debugger begins with the execution of the first command in the file and displays its prompt after all commands in the file have been executed.

If the @file-spec command is one of the commands in a DO clause, the debugger begins execution of the first command in the file upon reaching the @file-spec command in the DO clause. After execution of all commands in the file, the debugger resumes execution of any remaining commands in the DO clause and then issues its prompt.

If the @file-spec command is one of the commands in another (calling) command procedure, the debugger begins execution of the commands in the called command procedure when it reaches the @file-spec command. It resumes execution of any remaining commands in the calling command procedure after all commands in the called command procedure have been executed.

If the syntax of a command in a command procedure is incorrect, the debugger issues an error message and continues execution with the next command in the command procedure.

When the default output parameter NOVERIFY is in effect (see Section 2.5.1), the debugger does not display—either on the terminal or in a log file—the commands in a command procedure. The debugger simply executes the commands and displays the resulting output.

If you want the debugger to display the commands in a command procedure as it executes them, you must set the output parameter to VERIFY (see Section 2.5.1).

2.7.1 Editor-Created Command Procedures

You can use an editor to create a command procedure. You do this by creating a file and then entering debugger commands into the file.

For example, the following debugger commands make up the file whose file specification is BREAK.EDT:

```
SET BREAK/AFTER:3 %LINE 120 DO (EXAMINE K,N,J,X(K); GO)
SET BREAK/AFTER:3 %LINE 160 DO (EXAMINE K,N,J,X(K),S; GO)
SET BREAK %LINE 90
```

The purpose of the commands in this command procedure is to set breakpoints at three different program locations. Thus, whenever the command @BREAK.EDT is issued in the debugging session, the debugger sets the breakpoints specified in the command procedure.

The following example is a log file BREAK.LOG that records the use of the command procedure BREAK.EDT in a debugging session.

```
@BREAK.EDT              !Cause command procedure to be
                        !executed.
SHOW BREAK              !Show breakpoints set by @BREAK.EDT.
!breakpoint at MEANSUB$MAIN\%LINE 90
!breakpoint /after:3 at MEANSUB$MAIN\%LINE 160 DO (EX K,N,J,
X(K),S; GO)
!breakpoint /after:3 at MEANSUB$MAIN\%LINE 120 DO (EX K,N,J,
X(K); GO)

GO                       !Begin program execution.
!routine start at MEANSUB$MAIN\MEANSUB$MAIN
!break at MEANSUB$MAIN\MEANSUB$MAIN %LINE 120
!MEANSUB$MAIN\MEANSUB$MAIN\K:  3.000000
!MEANSUB$MAIN\MEANSUB$MAIN\N:  3.000000
!MEANSUB$MAIN\MEANSUB$MAIN\J:  1.000000
!MEANSUB$MAIN\MEANSUB$MAIN\X(3):    93.00000
!start at MEANSUB$MAIN\MEANSUB$MAIN\%LINE 120
!break at MEANSUB$MAIN\MEANSUB$MAIN\%LINE 160
```

```
!MEANSUB$MAIN\MEANSUB$MAIN\K:  3.000000
!MEANSUB$MAIN\MEANSUB$MAIN\N:  3.000000
!MEANSUB$MAIN\MEANSUB$MAIN\J:  1.000000
!MEANSUB$MAIN\MEANSUB$MAIN\X(3):    93.00000
!MEANSUB$MAIN\MEANSUB$MAIN\S:  252.0000
!start at MEANSUB$MAIN\MEANSUB$MAIN\%LINE 160
!break at MEANSUB$MAIN\MEANSUB$MAIN\%LINE 90
CANCEL BREAK/ALL        !Cancel all breakpoints.
```

2.7.2 Using Log Files as Command Procedures

You can use a debugger log file as a command procedure. A debugger log file contains debugger commands and debugger output; however, the debugger output is always preceded on the line by an exclamation point, which signals the debugger to ignore the line.

The exclamation point is called the comment delimiter because it is used to delimit comments within the command string. The debugger's use of the comment delimiter on its own output in log files makes possible the direct use, without modification, of debugger log files as command procedures.

You request that the debugger execute commands from a log file by specifying that file as a command procedure, as follows:

```
DBG>@file-spec
```

Thus, the log file BREAK.LOG shown in Section 2.6.1 may be used as a command procedure in another debugging session by issuing the @BREAK.LOG command. When you issue this command in a subsequent debugging session, the following debugger commands are executed:

- @BREAK.EDT
- SHOW BREAK
- GO
- CANCEL BREAK/ALL

Thus, you can reexecute a previous debugging session by using a log file as a command procedure. This feature is helpful if you want to continue debugging from where you left off at a previous session. You can also use the log file as a command procedure in another program or in a modified version of the same program.

2.8 Interrupting a Debugging Session

Since the debugger is itself an image within the context of the VAX/VMS operating system, it reacts to interruption like other images running in user mode; for instance, the image is interrupted (but unchanged), the terminal type-ahead buffer is purged, and the DCL command interpreter receives control. To interrupt a debugging session, enter CTRL/Y (echoed as ^Y) as follows:

```
DBG> CTRL/Y
Interrupt
```

After you press CTRL/Y, the DCL command interpreter takes control.

If you run your program without debugger control, interrupt it, and start the debugger with the DEBUG command, the debugger informational message indicates the name of the module that contains the program location at which program execution was interrupted and the name of the language in which that module was written.

If you run your program with debugger control, you may still need to interrupt it by pressing CTRL/Y and then giving the DEBUG command. For instance, if your program goes into an infinite loop after you issue the GO command, your only option is to press CTRL/Y. In this case, the CTRL/Y-DEBUG command sequence brings you back to the DBG> prompt, leaving all your debugging parameters as they were.

You may also need to interrupt a debugger command which is taking too long to complete or is generating a large amount of output. In this case, the CTRL/Y-DEBUG command sequence causes the debugger to abort the current command when all of the data structures are in a consistent state.

2.8.1 CTRL/Y and CTRL/C

The effect of issuing CTRL/Y or CTRL/C is the same unless your system or application program contains a routine coded to intercept CTRL/C.

If such a CTRL/C handling routine exists, entering CTRL/C causes control to be passed to the handling routine rather than to the command interpreter.

If a CTRL/C handling routine does not exist, both CTRL/Y and CTRL/C interrupt the debugging session and cause control to be passed to the DCL command interpreter, which signals with the DCL prompt.

When the DCL prompt is displayed, you can enter any DCL command. Section 2.7.2 discusses the effect of some of these commands in relation to restarting a debugging session.

2.8.2 Options After Interruption

After interruption of a debugging session by CTRL/Y or CTRL/C and the resulting display of the DCL prompt, you may want to enter any of the following DCL commands:

- DEBUG—See Section 2.1.4.
- CONTINUE—Passes control back to the debugger or to the program, whichever had control at the time of interruption.
- STOP—Causes abnormal termination of the debugger. The debugger exit handler is not given control.
- EXIT—Causes normal termination of the debugger. The debugger exit handler is given control.
- Commands that are performed within the DCL command interpreter—These commands may be entered and executed without causing termination of the debugger image. For example, after interruption of the debugger by CTRL/Y, if you enter the SHOW DAYTIME command and follow it with the DEBUG command, control will pass back to the debugger.
- Any other DCL command—Causes termination of the debugger. The debugger exit handler is executed.

- The SPAWN and ATTACH commands—Allows you to create a subprocess or attach to a detached process. See Section 2.7.3.

2.8.3 The SPAWN and ATTACH Commands

The debugger provides two commands—SPAWN and ATTACH—that allow you to interrupt your debugging session without ending it.

The debugger command SPAWN acts like the DCL command SPAWN. It provides a way of temporarily leaving the debugger for the DCL command interpreter. You can also specify SPAWN/NOWAIT with a DCL command. This command allows you to continue your debugging session without waiting for the spawned subprocess to complete.

However, if you plan to spawn several times during one debugging session, you should use the ATTACH command instead. The ATTACH command allows you to go back and forth between a debugging session and the DCL command interpreter, or between two debugging sessions. First, you must spawn a subprocess; then you can attach to it whenever you want. To return to your orginal process, use another ATTACH command. Since you are not creating a new subprocess every time you leave the debugger, you are more efficiently using system resources.

3 Controlling Program Execution

This chapter explains how to start, suspend, and monitor program execution, and how to display the current program status. It also discusses how to debug exit handlers.

You can control and monitor program execution by means of all the commands discussed in this chapter: STEP, GO, CALL, SHOW CALLS, SET BREAK, SET TRACE, and SET WATCH. Refer to the Command Dictionary for detailed descriptions of each of these commands.

3.1 Starting Program Execution

To begin program execution at debugger start up or to continue program execution after interruption, issue the STEP, GO, or CALL command. The following sections discuss each of these in turn.

3.1.1 The STEP Command

By means of the STEP command, you can control the execution of your program. When you issue a STEP command, the debugger responds as follows:

1. It executes one or more instructions.
2. It reports the line or instruction that follows the last instruction executed.
3. It reports the source line corresponding to the line or instruction that follows the last instruction executed only if the SOURCE parameter is in effect by virtue of STEP/SOURCE or SET STEP SOURCE and source lines are available.
4. It issues the DBG> prompt.

The format of the STEP command is

```
STEP [/qualifier...] [integer]
```

The decimal integer is an optional parameter that indicates how many lines or instructions are to be executed. If the decimal integer is omitted, the debugger uses a default value of 1.

The STEP command qualifiers specify whether the debugger

- Steps by line or by instruction
- Steps "into" or "over" called routines in the user program space (user-written routines)
- Steps "into" or "over" called routines in system space
- Steps to the next branch instruction, call instruction, return instruction, or exception condition
- Displays the messages associated with the step command

- Displays lines of source code as it steps

If STEP command qualifiers are not specified with the STEP command, the debugger uses STEP parameters established by the SET STEP command (see Section 3.1.1.1, or STEP parameters associated by default with the current language.

The following list describes the effects of the STEP command qualifiers:

/BRANCH	Steps to the next branch instruction.
/CALL	Steps to the next call instruction.
/EXCEPTION	Steps to the next exception condition.
/INSTRUCTION	Steps to the next instruction.
/INSTRUCTION=opcode	Steps to the instruction whose opcode you specify with the code parameter.
/INTO	Steps "into" called routines in the user program space (and "into" called routines in system space if /SYSTEM is also specified); that is, the debugger does not differentiate between instructions (or lines) within a called routine and those outside of the routine as it executes steps. Thus, the execution of a STEP command may result in suspension of execution within the called routine, in which case subsequent step commands will execute the instructions (or lines) within that routine.
/LINE	Steps to the next line.
/OVER	Steps "over" called routines in the user program space and in system space; that is, the debugger considers any instructions executed as the result of a CALL instruction, up to and including the corresponding RETURN instruction, to be part of a single step.
/NOSILENT	Overrides the effect of /SILENT. In other words, the "start at" and "stepped to" resume display.
/NOSOURCE	Specifies that the debugger not display the line of source code that corresponds to the instruction(s) being executed. This qualifier does not apply to all languages. See Section 3.7 for more information.
/NOSYSTEM	Steps "over" called routines in system space. That is, the debugger considers any instructions executed as the result of a CALL instruction to a routine in system space, up to and including the corresponding RETURN instruction, to be part of a single step.
/RETURN	Causes the debugger to step to the return instruction that returns from the current routine. It is valid for CALLS and CALLG routines only.
/SILENT	Specifies that the "stepped to" message associated with the STEP command not be displayed. The /SILENT qualifier is most useful in command procedures or in DO clauses when you do not want the customary output from a STEP command.

/SOURCE	Specifies that the debugger display the line of source code that corresponds to the instruction(s) being executed. This qualifier does not apply to all languages. See Section 3.7 for more information.
/SYSTEM	Steps "into" called routines in system space, provided that /INTO is also specified; that is, the debugger, in its execution of steps, does not differentiate between instructions (or lines) within a called routine in system space and those outside of system space. Thus, execution of a STEP command may result in suspension of execution in system space.

When a qualifier is specified with the STEP command, it temporarily overrides the STEP parameter established by the SET STEP command or the default STEP parameter associated with the current language.

3.1.1.1 The SET STEP and SHOW STEP Commands

The SET STEP command sets the default step parameters; that is, it establishes the STEP parameters that the debugger uses whenever a STEP command is issued without a STEP command qualifier.

When a STEP parameter is specified as a qualifier in the STEP command, it overrides, for the duration of the command, any step parameters established by the SET STEP command or any step parameters associated with the current language.

As described in Section 3.3.1, step parameters may be any of the following: BRANCH, CALL, EXCEPTION, LINE, INSTRUCTION, INSTRUCTION=opcode, INTO, OVER, NOSILENT, NOSOURCE, NOSYSTEM, RETURN, SILENT, SOURCE, or SYSTEM. Note, however, that a step parameter is not prefixed with a slash unless it is used as a STEP command qualifier.

The following is the format of the SET STEP command:

```
SET STEP step-parameter [,step-parameter...]
```

Note that default step parameters are associated with each language. Thus, when you change languages by using the SET LANGUAGE command, the default step parameters may also change.

To display the current step parameters, issue the SHOW STEP command.

At the assembly level, program locations are often specified by virtual addresses or by offsets from program labels, rather than by symbols. Thus, it is common to use virtual addresses as parameters in commands such as STEP that affect program execution.

So, if you are debugging assembly-level code, you may find it useful to set the STEP parameter to INSTRUCTION. Then, whenever you issue a STEP command, you will advance by one instruction or by the number of instructions you specify as a parameter in the STEP command. By single stepping, you can literally read your program's source code at the assembly level as VAX instructions.

Further, you may want to set STEP parameters to INTO to enable you to step through the called routines in your program. Then whenever program execution reaches a routine call, you can follow program flow through the called routine (a single instruction at a time or by lines).

Example 3–2 demonstrates the use of the STEP command.

Example 3–2 Using the SET/SHOW STEP Commands

```
DBG>SHOW STEP                                   !Display current step parameters.
step type: nosystem, source, nosilent, over routine calls, by line

DBG>SHOW MODE                                   !Display current modes.
modes: symbolic, d_float, noscreen, nokeypad
input radix : decimal
output radix: decimal

DBG>SET STEP INSTRUCTION, INTO, NOSOURCE        !Set STEP parameters.

DBG>SHOW STEP                                   !Display current step parameters.
step type: nosystem, nosource, nosilent, into routine calls, by instruction

DBG>STEP                                        !Step through the program.
stepped to CIRCLE %LINE 2 : PUSHAL  L^512

DBG>STEP
stepped to CIRCLE %LINE 2 +6: MNEGL   #2,-(SP)

DBG>STEP
stepped to CIRCLE %LINE 2 +9: CALLS   #2,L^FOR$WRITE_SF

DBG>STEP
stepped to FOR$WRITE_SF+2: JMP    L^56758   !Step into a called routine
                                            !named FOR$WRITE_SF.
                                            !Once the 2-byte entry mask is
                                            !stepped by, the debugger
                                            !references instructions by
                                            !their virtual address; this
                                            !is because FOR$WRITE_SF is
                                            !not in the run-time symbol
                                            !table.

DBG>STEP
stepped to 56758: MOVZBL  #1,R0

DBG>STEP
stepped to 56761: BRW     56870
    .
    .
    .
DBG>STEP/RETURN
stepped to 58561: RET                           !Return from FOR$WRITE_SF.

DBG>STEP
stepped to CIRCLE %LINE 4 : PUSHAL  L^535   !Begin executing main program.

DBG>STEP
stepped to CIRCLE %LINE 4 +6: MNEGL   #3,-(SP)

DBG>STEP
stepped to CIRCLE %LINE 4 +9: CALLS   #2,L^FOR$READ_SF

DBG>STEP
stepped to FOR$READ_SF+2: JMP     L^56646   !Step into the called
                                            !routine FOR$READ_SF.

DBG>STEP
stepped to 56646: MOVZBL  #2,R0

DBG>STEP
stepped to 56649: BRW     56870

DBG>GO
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
```

3.1.2 The GO Command

You can use the GO command under three circumstances:

- To begin execution of your program at debugger start up, in which case execution begins from the transfer address
- To resume execution at the instruction following the last instruction executed, in which case you do not specify an address expression as a parameter in the GO command
- To resume execution at a specified program location, in which case you specify the corresponding address expression as a parameter in the GO command

Unlike the STEP command, which suspends execution after a specified number of instructions or lines have been executed, the GO command will cause your program to continue executing until one of the following occurs:

- The program terminates.
- A breakpoint is reached.
- A watchpoint is activated.
- An exception occurs.
- The program is interrupted by CTRL/Y or CTRL/C.

The following is the format of the GO command:

```
GO [address-expression]
```

If an address expression is specified as a parameter in the GO command, the debugger begins program execution at the location specified by the address expression.

Using an address expression as a parameter in the GO command may be helpful when you want to reexecute sections of your program but do not want to start all over again. Note, however, that when you specify an address-expression parameter in the GO command, the program state at the location denoted by the address expression must be identical to the program state at the time you issue the GO command; otherwise, results may be unpredictable.

See Section 3.2.1.1 for examples of how to use the GO command.

3.1.3 The CALL Command

The CALL command is useful in debugging routines in your program.

When you issue a CALL command, the debugger takes the following action:

1. Saves the current values of the general registers
2. Constructs an argument list
3. Executes a call to the routine specified in the command and passes any arguments
4. Executes the routine
5. Displays the value returned by the routine in R0

6 Restores the values of the general registers to the values they had just previous to the CALL command

7 Issues its prompt

The debugger executes a routine called by the CALL command whether or not your program actually includes a call to that routine, so long as the routine was linked with your program.

You can also debug unrelated routines by linking them with a dummy main module that has a transfer address and then using the CALL command to execute them.

The following is the format of the CALL command:

```
CALL routine-name [(argument1[,argument2...])]
```

Note that any arguments must be enclosed in parentheses.

3.2 Suspending Program Execution

The following sections discuss how to suspend program execution by setting breakpoints, exception breakpoints, and watchpoints.

3.2.1 Breakpoints

A breakpoint is a program location at which the debugger performs the following actions:

1 Suspends program execution

2 Checks the AFTER count and resumes program execution if the specified number of breakpoint activations has not yet been reached

3 Evaluates the WHEN clause (if it is present) and resumes execution if it evaluates as FALSE in the current language

4 Displays the name or the virtual address of the location at which execution has been suspended

5 Executes commands in a DO clause if one was specified in the SET BREAK command

6 Issues the DBG> prompt

To set a breakpoint, issue the SET BREAK command in the following format.

```
SET BREAK [/qualifier...] [address-expression -
[,address-expression,...]] [WHEN (expression)]-
[DO (command [;command...])]
```

When you specify an address expression on the SET BREAK command, the breakpoint is then associated with the given program location. Specifying a list of address expressions is equivalent to setting more than one breakpoint (one for each location in the list). When the debugger breaks at the given address, it suspends execution before executing the instruction at that location.

Some of the command qualifiers (/BRANCH, /CALL, /EXCEPTION, /INSTRUCTION[=opcode], and /LINE) indicate classes of conditions on which the breakpoint should be taken. For example, SET BREAK/CALL tells the debugger to break at every call to a subroutine. If you specify one of these qualifiers, you should not specify an address expression.

The /RETURN qualifier specifies that the break is to be taken not at the given address but rather at the return statement of the routine containing the address. In other words, "SET BREAK/RETURN routine" breaks at the return from the routine rather than at the entry to the routine.

Other qualifiers on the SET BREAK command include /AFTER:n, /SILENT, /SOURCE, and /TEMPORARY. These qualifiers generally affect what output you see at the breakpoint or whether the breakpoint is taken. See the SET BREAK command in the Command Dictionary for more details.

If you specify a WHEN clause, the debugger evaluates it; if the expression in a WHEN clause evaluates as TRUE in the current language, break action occurs. However, if the clause evaluates as FALSE, break action does not occur, the commands specified in the DO command list are not executed, and program execution is continued.

If you specify a DO clause, the debugger executes commands in the DO clause when the breakpoint is activated. See Section 3.2.1.1.

If you use a virtual memory address or an address expression whose value is not a symbolic location as a parameter in the SET BREAK command, you should check that an instruction actually begins at the byte of memory so indicated. If an instruction does not begin at this byte, a run-time error may occur when an instruction including that byte is executed. Note that when the breakpoint is set, the debugger does not verify that the location specified marks the beginning of an instruction.

When a routine name is the parameter in a SET BREAK command, the debugger notes this fact when the breakpoint is activated by issuing the following message:

```
routine break at routine name
```

Routine breakpoints are special cases because the breakpoint is actually set at the memory address 2 bytes greater than the memory address of the routine name itself (called the entry point). This is done so that the breakpoint is not set on the 2-byte entry mask of the routine, but is set instead at the first instruction in the routine, which begins directly following the entry mask.

Note that the command

```
DBG>EXAMINE routine-name
```

does not skip over the routine entry mask. Therefore, to examine the first instruction in the routine, you must issue the command

```
DBG>EXAMINE routine-name + 2
```

To see what breakpoints are in effect, issue the *SHOW BREAK* command.

Once set, a breakpoint remains active for the duration of the debugging session unless you cancel it with the CANCEL BREAK command or set another breakpoint, watchpoint, or tracepoint at that program location. In that case the old breakpoint specification is overwritten.

To cancel one or more breakpoints, issue the CANCEL BREAK command in the following format:

```
CANCEL BREAK [/qualifier] [address-expression -
[,address-expression,...]]
```

If you specify an address-expression parameter, the breakpoint at the location denoted by the address expression is canceled. In this case, you cannot also specify a command qualifier.

If you specify the /ALL command qualifier, all breakpoints are canceled. In this case, you cannot also specify an address-expression parameter.

Example 3–3 shows how breakpoints and watchpoints may be set, canceled, and overwritten.

Example 3–3 Setting, Showing, and Canceling Eventpoints

```
DBG>SET BREAK %LINE 15                      !Set breakpoint.

DBG>SHOW BREAK                              !Show breakpoint.
breakpoint at TOY\%LINE 15

DBG>CANCEL BREAK %LINE 15                   !Cancel breakpoint.

DBG>SHOW BREAK
%DEBUG-I-NOBREAKS, no breakpoints are set

DBG>SET BREAK %LINE 15                      !Set breakpoint.

DBG>SET WATCH %LINE 15                      !Set watchpoint at
                                             !same location.

DBG>SHOW BREAK                              !Breakpoint is
%DEBUG-I-NOBREAKS, no breakpoints are set    !overwritten.

DBG>SHOW WATCH                              !Watchpoint set.
watchpoint at TOY\%LINE 15 for 4. bytes.

DBG>SET BREAK %LINE 15                      !Set breakpoint.

DBG>SET BREAK/AFTER:5 %LINE 15              !Set modified
                                             !breakpoint at
                                             !same location.

DBG>SHOW BREAK                              !Previous breakpoint
breakpoint /after:5 at TOY\%LINE 15          !overwritten.
```

Note that the following commands overwrite one another when they contain identical address-expression parameters:

- SET BREAK (in all its forms)
- SET TRACE
- SET WATCH

3.2.1.1 Command Sequence at Breakpoint

If you want the debugger to execute one or more debugger commands immediately after a breakpoint has been reached, you specify these commands in a DO clause as shown in the following format:

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

Whether the DO clause includes one or more commands or invokes a command procedure, parentheses are required delimiters, as shown in the format above. If the DO clause includes a list of commands, you must separate commands with a semicolon.

The debugger executes commands in a DO clause in the order in which they are listed. If one of the commands is a command procedure (see Section 2.6), the debugger begins execution of the commands in that file when it reaches the command file specification (@file-spec) in the DO clause. After the debugger has executed each command in the command procedure, it continues executing commands in the DO clause until it exhausts them or until it reaches another command file specification.

The DO clause may contain commands that resume program execution. For example, a typical DO clause might contain several EXAMINE commands followed by a GO command.

Note that the debugger does not check the syntax of the commands within a DO clause after you issue the SET BREAK command containing that DO clause; the debugger checks for syntax when it actually executes the commands in the DO clause at breakpoint activation.

Any debugger command, including a SET BREAK command with a DO clause, may appear in a DO clause. Nesting of DO clauses is permissible to any level, so long as syntax rules have not been violated.

You are not limited in the number and type of debugger commands you can include in a DO clause. If all the commands you want to include in the clause do not fit on a single input line, specify the line continuation character (-) at the end of the input line and then press RETURN. In this way, you can continue entering commands within the same DO clause.

If you want the commands in the DO clause to be executed conditionally, include a WHEN clause in the SET BREAK command, as follows:

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

The WHEN clause specifies any expression in the currently set language that you want logically evaluated every time the breakpoint occurs. If the expression is true, break action occurs. If it is false, break action does not occur, and the command specified by the DO command list is not executed.

Example 3–4 demonstrates the power and versatility of the DO clause at breakpoint.

Example 3–4 Using SET BREAK in a DO Clause

```
DBG>SET BREAK %LINE 16 DO (EXAMINE I; EXAMINE R; STEP)

DBG>GO
routine start at TOY                !Start at routine.
break at TOY\%LINE 16               !Break at line 16.
TOY\I: 1                            !EXAMINE I in DO clause.
TOY\R: 0.0000000E+00                !EXAMINE R in DO clause.
start at TOY\%LINE 16
stepped to TOY\%LINE 17             !STEP in DO clause.

DBG>SET BREAK %LINE 16 DO (EXAM I;EXAM R; GO %LINE 15)
                                    !This command causes the
                                    !execution of an infinite
                                    !loop once the breakpoint
                                    !at line 16 is activated.

DBG>GO %LINE 15                    !Begin execution.
start at TOY\%LINE 15
break at TOY\%LINE 16               !Breakpoint activated.
TOY\I: 1                            !EXAMINE I.
TOY\R: 0.0000000E+00                !EXAMINE R.
start at TOY\%LINE 15               !GO %LINE 15.
break at TOY\%LINE 16               !Breakpoint activated.
TOY\I: 1                            !EXAMINE I.
TOY\R: 0.0000000E+00                !EXAMINE R.
start at TOY\%LINE 15               !GO %LINE 15.
break at TOY\%LINE 16               !Breakpoint activated.
TOY\I: 1                            !and so on.
TOY\R: 0.0000000E+00
start at TOY\%LINE 15
break at TOY\%LINE 16

CTRL/Y                             !Interrupt by CTRL/Y.
Interrupt
$
```

3.2.2 Exception Breakpoints

An event arising within the context of an executing program may require the execution of software outside that program's explicit flow of control. The notification of such an event is called an exception, and the presence of an exception indicates that a "condition" exists. Exception conditions range in severity and vary in the way they affect an executing program. The following are examples of exception conditions:

- An arithmetic overflow or underflow
- A memory access violation
- An invalid operation code
- A division by zero

You direct the debugger to treat an exception generated by your program as a breakpoint by issuing the SET BREAK/EXCEPTION command. Specifying the command SET EXCEPTION BREAK has the same effect. As a result of this command, whenever your program generates an exception condition, the debugger responds by suspending program execution, reporting the exception condition, and prompting you for input.

Whenever an exception breakpoint is activated, therefore, you have the opportunity to issue debugger commands. When you want to continue program execution, you can specify one of the following commands:

- A GO command without an address-expression parameter—In this case, the debugger fields and resignals the exception, thus allowing any user-declared exception handlers to execute.
- A GO command with an address-expression parameter—In this case, the debugger allows program execution to continue at the specified location, thus inhibiting the execution of any user-declared exception handlers.

Note that you cannot issue a STEP command to resume program execution after breakpoint activation. The STEP command is illegal in this context and, if issued, will result in a warning message.

If you do not specify an exception breakpoint with the SET BREAK /EXCEPTION command or if you cancel an exception breakpoint with the CANCEL BREAK/EXCEPTION command, exception conditions generated by your program are handled in the following way:

1. The debugger fields and resignals the exception.
2. If you have defined a condition handler in your program, it is executed.
3. If you have not defined a condition handler or if a condition handler that you have defined resignals the exception condition, a diagnostic message is issued and control is returned to the debugger, which then displays its prompt.

Note that an exception handler executes until one of the following occurs:

- The exception handler "handles" the condition, thus allowing the program to continue execution.
- The exception handler resignals the exception condition.
- The exception handler encounters a breakpoint or watchpoint.
- The exception handler generates its own exception.
- The exception handler exits, thus terminating program execution.

You cancel the SET BREAK/EXCEPTION command by issuing the CANCEL BREAK/EXCEPTION command.

When debugging exceptions, you can use the following four built-in symbols which provide a means of qualifying exception breakpoints.

- %EXC_FACILITY
- %EXC_NAME
- %EXC_NUMBER
- %EXC_SEVERITY

The %EXC_FACILITY built-in symbol returns the facility of the current exception. For example:

```
DBG> EVALUATE %EXC_FACILITY
"SYSTEM"
DBG> SET BREAK/EXCEPTION WHEN (%EXC_FACILITY = "SYSTEM")
```

The %EXC_NAME built-in symbol returns the name of the current exception. For example:

```
DBG> EVALUATE %EXC_NAME
"FLTDIV_F"
DBG> SET BREAK/EXCEPTION WHEN (%EXC_NAME = "FLTDIV_F")
```

The %EXC_NUMBER built-in symbol returns the current exception number. For example:

```
DBG> EVALUATE %EXC_NUMBER
12
DBG> EVALUATE/CONDITION_VALUE %EXC_NUMBER
%SYSTEM-F-ACCVIO, access violation at PC virtual address
DBG> SET BREAK/EXCEPTION WHEN (%EXC_NUMBER = 12)
```

The %EXC_SEVERITY built-in symbol returns the severity code of the current exception. For example:

```
DBG> EVALUATE %EXC_SEVERITY
"F"
DBG> SET BREAK/EXCEPTION WHEN (%EXC_SEVERITY = "F")
```

For more information on exceptions, consult any of the following sources: the *VAX Architecture Handbook*, the *VAX/VMS System Services Reference Manual*, or the *VAX/VMS Run-Time Library Routines Reference Manual*.

3.2.3 Watchpoints

You can direct the debugger to watch an entity and notify you if its value is modified. To do this, you specify as a parameter in the SET WATCH command an address expression that identifies the location where the entity resides. The format of the SET WATCH command is

```
SET WATCH [/qualifier] address-expression -
[,address-expression,...] [WHEN (expression)] -
[DO (command_list)]
```

As a result of the SET WATCH command, the debugger sets a watchpoint at the program location specified by the address expression.

If the entity has a compiler-generated type, the debugger uses the length in bytes associated with that type to determine the length in bytes of the watched location. If the entity does not have a compiler-generated type, the debugger watches 4 bytes of virtual memory, beginning at the byte identified by the address expression.

The SET WATCH qualifiers include /AFTER:n, /SILENT, /SOURCE, and /TEMPORARY. For detailed information about these qualifiers, see the Command Dictionary.

If you specify a WHEN clause, the debugger evaluates it; if the expression in WHEN clause evaluates as TRUE in the current language, watch action occurs. However, if the clause evaluates as FALSE, watch action does not occur, the commands specified in the DO command list are not executed, and program execution is continued.

If you specify a DO clause, the debugger executes commands in the DO clause when the watchpoint is activated. See Section 3.2.1.1.

Whenever an instruction causes the modification of a watched entity, the debugger performs the following operations:

1. Suspends program execution after that instruction has completed execution
2. Checks the AFTER count and resumes program execution if the specified number of breakpoint activations has not yet been reached
3. Evaluates the WHEN clause (if it is present) and resumes execution if it evaluates as FALSE in the current language
4. Displays the name or the virtual address of the location at which execution has been suspended
5. Executes commands in a DO clause if one was specified in the SET WATCH command
6. Identifies the instruction that modified the entity
7. Reports the old value of the entity
8. Reports the new (modified) value of the entity
9. Issues the DBG> prompt

To display watchpoints currently in effect, issue the command SHOW WATCH.

To cancel a watchpoint, issue the CANCEL WATCH command in the following format:

```
CANCEL WATCH [/qualifier] [address-expression -
[,address-expression,...]]
```

If you specify an address-expression parameter, the watchpoint at the location denoted by the address expression is canceled. In this case, you cannot also specify a command qualifier.

If you specify the /ALL command qualifier, all watchpoints are canceled. In this case, you cannot also specify an address-expression parameter.

You can set watchpoints on aggregates (that is, entire arrays or records). A watchpoint set on an array or record will trigger if any element of the array or record changes. Thus, you do not need to set watchpoints on individual array elements or record components. Note, however, that you cannot set an aggregate watchpoint on a variant record.

Example 3–5 demonstrates the SET WATCH, SHOW WATCH, and CANCEL WATCH commands.

Example 3–5 Using the SET/SHOW/CANCEL WATCH Commands

```
DBG>SET WATCH RESULT                          !Set watchpoint at
                                               !RESULT.

DBG>SHOW WATCH                                !Display watch-
watchpoint of TOTAL\RESULT                     !points. There are
                                               !4 bytes watched
                                               !beginning at
                                               !TOTAL\RESULT.

DBG>GO                                        !Notification of
start at TOTAL\START+02                        !watchpoint modifi-
write to TOTAL\RESULT at PC TOTAL\G03          !cation.  Note that
old value = 0000000                            !TOTAL\G03 identi-
new value = 00000A66                           !fies the location
                                               !of the instruction
                                               !that caused the
                                               !modification of
                                               !RESULT.

DBG>SHOW MODE
modes:  symbolic, d_float, noscreen, nokeypad
input radix : hexadecimal
output radix: hexadecimal
                                               !Values are
                                               !displayed in hexa-
                                               !decimal.  Address
                                               !expressions are
                                               !displayed symboli-
                                               !cally where possi-
                                               !ble.

DBG>SHOW WATCH                                !Watchpoint still in
watchpoint at TOTAL\RESULT for 4. bytes.       !effect.

DBG>CANCEL WATCH RESULT                       !Cancel the watch-
                                               !point.

DBG>SHOW WATCH                                !Show current watch-
%DEBUG-I-NOWATCHES, no watchpoints are set     !points.

DBG> SET WATCH ARR                            !Set watchpoint on
                                               !array ARR.
DBG> GO                                       !Watchpoint triggers
watch of SUBR\ARR at SUBR\%LINE 12+8           !when any element
   old value:                                  !of ARR changes.
    (1):        7                              !Here, the third
    (2):        12                             !element changed.
    (3):        3
    (4):        0

   new value:
    (1):        7
    (2):        12
    (3):        28
    (4):        0

break at SUBR\%LINE 14
```

3.2.3.1 Watchpoint Restrictions

The mechanism for implementing watchpoints is to establish write protection for the page of memory that contains the watchpoint; consequently, when a write operation to that page is attempted, an exception occurs. The debugger handles the exception. It temporarily unprotects the page to allow the instruction to complete and then it determines whether the watched variable was modified.

Because problems would arise if the stack were write protected, currently you cannot set watchpoints on variables that are allocated on the stack.

3.3 Monitoring Program Execution

This section discusses how to monitor program execution by tracing the flow of program control and by displaying the current state of procedure calls.

3.3.1 Tracepoints

By setting tracepoints, you can monitor the sequence in which instructions in your program are executed and thereby check for unexpected control transfers. Moreover, tracepoints do not interrupt or otherwise disturb program execution.

When a tracepoint is activated, the debugger performs the following steps:

1. Suspends program execution
2. Checks the AFTER count and resumes program execution if the specified number of tracepoint activations has not yet been reached
3. Evaluates the WHEN clause (if it is present) and resumes execution if it evaluates as FALSE in the current language
4. Displays the name or the virtual address of the location at which execution has been suspended
5. Executes commands in a DO clause if one was specified in the SET TRACE command
6. Reports that execution has reached the traced location
7. Resumes execution at the point of suspension

To set a tracepoint, issue the SET TRACE command in the following format:

```
SET TRACE [/qualifier] [address-expression -
[,address-expression,...]] [WHEN (expression)] -
[DO (command-list)]
```

A tracepoint has the same effect as a breakpoint except that program execution resumes immediately after a tracepoint has been taken. The syntax of the SET TRACE command and the SET BREAK command is the same. Also, the command qualifiers and their meanings are the same for the two commands.

If you specify the /CALL or the /BRANCH command qualifier, the debugger sets tracepoints at all instructions that are members of the family of CALL or BRANCH instructions, respectively (see Section 3.3.1.1). In this case, you cannot also specify an address-expression parameter. SET TRACE /INSTRUCTION will trace every instruction that gets executed.

To see what tracepoints are in effect, issue the command SHOW TRACE.

To cancel one or more tracepoints, issue the CANCEL TRACE command in the following format:

```
CANCEL TRACE [/qualifier] [address-expression -
[address-expression,...]]
```

If you specify an address-expression parameter, the debugger cancels the tracepoint at the location denoted by the address expression. In this case, you cannot also specify a command qualifier.

If you specify the /CALL or the /BRANCH command qualifier, the debugger cancels tracepoints at all instructions that are members of the family of CALL or BRANCH instructions, respectively. In this case, you cannot also specify an address-expression parameter.

If you specify the /ALL command qualifier, the debugger cancels all tracepoints. In this case, you cannot also specify an address-expression parameter.

Example 3–6 demonstrates the use of the SET, SHOW, and CANCEL TRACE commands.

Example 3–6 Using the SET/SHOW/CANCEL TRACE Commands

```
DBG>SHOW TRACE
%DEBUG-I-NOTRACES, no tracepoints are set, no opcode tracing
                                          !Show tracepoints in
                                          !effect.

DBG>SET TRACE %LINE 30                    !Set tracepoint at %LINE
                                          !30.

DBG>SET BREAK %LINE 60                    !Set breakpoint at %LINE
                                          !60.

DBG>SHOW TRACE                            !Show tracepoints in
tracepoint at MEANSUB$MAIN\%LINE 30       !effect.

DBG>GO                                    !Begin program execution.
routine start at MEANSUB$MAIN             !Start of program.
trace at MEANSUB$MAIN\%LINE 30            !Tracepoint reached.
break at MEANSUB$MAIN\%LINE 60            !Breakpoint reached.

DBG>SET TRACE %LINE 40                    !Set another tracepoint.

DBG>SHOW TRACE                            !Show tracepoints in
tracepoint at MEANSUB$MAIN\%LINE 40       !effect.
tracepoint at MEANSUB$MAIN\%LINE 30

DBG>GO %LINE 20                           !Resume execution at
start at MEANSUB$MAIN\%LINE 20            !%LINE 20.
trace at MEANSUB$MAIN\%LINE 30            !Tracepoint reached.
trace at MEANSUB$MAIN\%LINE 40            !Tracepoint reached.
break at MEANSUB$MAIN\%LINE 60            !Breakpoint reached.

DBG>CANCEL TRACE %LINE 40                 !Cancel one tracepoint.

DBG>SHOW TRACE                            !Show tracepoints in
tracepoint at MEANSUB$MAIN\%LINE 30       !effect.

DBG>CANCEL TRACE/ALL                      !Cancel all tracepoints.

DBG>SHOW TRACE
%DEBUG-I-NOTRACES, no tracepoints are set, no opcode tracing
                                          !Show tracepoints in
                                          !effect.
```

3.3.1.1 Opcode Tracing

Opcode tracing is the tracing of all instructions or the tracing of a family of instructions.

When you issue SET TRACE/INSTRUCTION, the debugger traces all instructions. When you issue the command SET TRACE/CALL, the debugger traces any of the following instructions: CALLS, CALLG, BSBW, BSBB, JSB, RSB, RET.

If you issue the command SET TRACE/BRANCH, the debugger traces any of the following instructions:

ACBB	BBSS	BNEQ
ACBL	BBSSI	BRB
ACBQ	BEQL	BRW
ACBD	BGEQ	BVC
ACBG	BGEQU	BVS
ACBH	BGTR	CASEB
AOBLEQ	BGTRU	CASEL
AOBLSS	BLBC	CASEW
BBC	BLBS	JMP
BBCC	BLSS	SOBGEQ
BBCCI	BLSSU	SOBGTR
BBCS	BLEQ	
BBS	BLEQU	

Note that you can also trace general lists of opcodes. For instance, SET TRACE/INSTRUCTION=(ADDL3,MULL3) traces all ADDL3 and MULL3 instructions.

To see the tracepoints currently in effect, issue the command SHOW TRACE.

To cancel tracepoints at CALL or BRANCH instructions, use the CANCEL TRACE command with the appropriate qualifier, as follows:

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

Note that opcode tracing noticeably slows program execution.

Example 3–7 demonstrates the use of the /CALL and /BRANCH qualifiers in the SET, SHOW, and CANCEL TRACE commands.

Example 3–7 Using the /CALL and /BRANCH Qualifiers with SET TRACE

```
DBG>SET TRACE/CALL                      !Trace call instructions.

DBG>SHOW TRACE                          !Show tracepoints in effect.
tracing /CALL instructions:  CALLS, CALLG, BSBW, BSBB, JSB, RSB and RET

DBG>GO %LINE 20                         !Begin program execution.
start at MEANSUB$MAIN\%LINE 20
trace at PC MEANSUB$MAIN\%LINE 30 +9: CALLS \#1,L^BAS$INIT_GOSUB
trace at PC 24566:  JSB    @B^4(AP)
trace at PC MEANSUB$MAIN\%LINE 200 +7: CALLS \#1,L^BAS$PRINT
trace at PC 24918: CALLS  #4,W^24987
trace at PC 25003: JSB  L^8352
trace at PC 8407: JSB  L^94721
trace at PC 94877: RSB
trace at PC 8492: RSB
trace at PC 25190: JSB  L^34295
trace at PC 34333: RSB
   .
   .
   .
DBG>SHOW TRACE                          !Show tracepoints in effect.
tracing /CALL instructions:  CALLS, CALLG, BSBW, BSBB, JSB, RSB and RET

DBG>SET TRACE/BRANCH                    !Trace all branch
                                         !instructions.

DBG>SHOW TRACE                          !Show tracepoints in effect.
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,
        ACBG, ACBH, AOBLEQ, AOBLSS, SOBGEQ, SOBGTR, CASEB, CASEW
and CASEL
```

(Continued on next page)

Example 3–7 (Cont.) Using the /CALL and /BRANCH Qualifiers with SET TRACE

```
DBG>GO %LINE 20                          !Resume program execution
start at MEANSUB$MAIN\%LINE 20           !at %LINE 20.
trace at PC MEANSUB$MAIN\%LINE 30 +9: CALLS \#1,L^BAS$INIT_GOSUB
routine trace at PC BAS$INIT_GOSUB: JMP  L^24510
trace at PC 24552: BBC  #11,B^-26(R0),24559
trace at PC 24566: JSB  @B^4(AP)
trace at PC MEANSUB$MAIN\%LINE 200 +7: CALLS \#1,L^BAS$PRINT
routine trace at PC BAS$PRINT: JMP  L^24896
trace at PC 24906: BNEQ  24913
trace at PC 24911: BRB  24916
trace at PC 24918: CALLS  #4,W^24987
trace at PC 25003: JSB  1^8352
trace at PC 8359: BGTR  8366
trace at PC 8364: BGEQ  8377
trace at PC 8391: BEQL  8407
trace at PC 8407: JSB  L^94721
trace at PC 94728: BLBS  L^103336,94740
trace at PC 94752: BBCS  R2,L^104364,94762
trace at PC 94772: BEQL  94790
trace at PC 94804: BEQL  94817
trace at PC 94815: BRB  94829
trace at PC 94860: BLBC  B^4(SP),94869
trace at PC 94877: RSB
trace at PC 8413: CASEL  R0,#1,#2
                                 8431,
                                 8431,
                                 8458
trace at PC 8437: BNEQ  8444
    .
    .
    .
DBG>CANCEL TRACE/ALL                     !Cancel all tracepoints.

DBG>SHOW TRACE                           !Show tracepoints in effect.
%DEBUG-I-NOTRACES, no tracepoints are set, no opcode tracing
```

3.3.2 The SHOW CALLS Command

When procedure A calls procedure B (by means of a CALL instruction), the VAX/VMS operating system preserves information about the program state of procedure A at the time that control is transferred to procedure B. This information is stored on the stack in a call frame for procedure B. The execution of a CALL instruction, therefore, results in the construction of a call frame for the called routine; this call frame contains information about the calling routine.

If procedure B calls procedure C, VAX/VMS builds a call frame for procedure C. The call frame for procedure C is built on top of the call frame for procedure B. In other words, the call frame for the most recently called procedure is on the top of the stack.

When a routine returns to its caller, the call frame for that routine is removed from the stack. Thus when procedure C finishes and control is returned to procedure B, the call frame for procedure C is removed from the stack.

The SHOW CALLS command provides information about the sequence of currently active procedure calls or the number of call frames on the stack. For example, if your program contains a recursive routine, you can use a SHOW CALLS command to examine the chain of recursion.

The format of the SHOW CALLS command is

```
SHOW CALLS [n]
```

The optional parameter n specifies the call count, or the number of call frames to be displayed, by a decimal integer in the range 0 through 32,767. If you do not specify the parameter n, then information on all call frames is displayed. If the call count represented by n exceeds the current number of call frames, information on all call frames is displayed. If the call count is 0, the command is accepted but no information is displayed. Otherwise, the number of call frames specified by the parameter n is displayed.

For each call frame, the debugger displays one line of information. The first line displayed contains information about the top call frame (the one representing the most recently called procedure); the next line contains information on the next most recently called procedure, and so on.

Each line of information displayed by the debugger contains

- The name of the module that contains the called routine. The debugger places an asterisk to the left of the module name if the module is set.
- The name of the called routine.
- The line number of the call (in line-oriented languages only).
- The value of the PC in the calling routine at the time that control was transferred to the called routine. Note that this value is the location of the instruction following the call. The value of the PC is expressed in two ways: as an absolute virtual address and as a virtual address relative to the virtual address of the name of the routine.

Note that even if your program contains no procedure calls, the debugger displays an active call when you issue the SHOW CALLS command. The reason for this is that your program has a stack frame built for it when it is first activated.

Thus, if the debugger responds that there are no active calls when you issue a SHOW CALLS command, either your program has terminated or the stack has been corrupted.

Example 3–8 demonstrates the use of the SHOW CALLS command.

3.4 Related Qualifier Functions

Many of the qualifiers that affect the SET BREAK, SET TRACE, SET STEP, SET WATCH, and STEP commands have the same functions. They can be divided into two broad categories: qualifiers that indicate location, and qualifiers that affect output.

Example 3–8 Using the SHOW CALLS Command

```
DBG>STEP
start at SUB2\%LINE 15
stepped to SUB2\%LINE 16

DBG>SHOW CALLS          !Display procedure calls.

module name    routine name      line    rel PC      abs PC

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

                       !SUB2 is the procedure that is
                       !currently executing.  SUB2 was
                       !called by SUB1, and SUB1 was
                       !called by MAIN.

DBG>GO                 !Continue program execution.
start at 19351
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful
completion'

DBG>SHOW CALLS
%DEBUG-W-NOCALLS, no active call frames
                       !Program has terminated.
```

3.4.1 Qualifiers That Indicate Location

This section describes a set of qualifiers that are common to SET BREAK, SET TRACE, and STEP. The qualifiers follow:

/BRANCH
/CALL
/EXCEPTION
/INSTRUCTION
/INSTRUCTION=opcode
/LINE
/RETURN address

When these qualifiers are used with the SET BREAK and SET TRACE commands, they tell the debugger where to break; and therefore take the place of an address expression. For example, the command SET BREAK /INSTRUCTION=ADDL3 breaks at every ADDL3 instruction.

When these qualifiers are used with the STEP command, they tell the debugger where to step. For example, STEP/INSTRUCTION=ADDL3 tells the debugger to step to the next ADDL3 instruction.

The seven qualifiers (or parameters) form a mutually exclusive set. For instance, if you issue the command SET STEP CALL but later issue the command SET STEP LINE, the LINE setting replaces the CALL setting as the default STEP condition.

3.4.2 Qualifiers That Affect Output

This section describes another set of qualifiers that are common to the SET BREAK, SET TRACE, SET WATCH, and STEP commands. These qualifiers are:

/SILENT
/NOSILENT
/SOURCE
/NOSOURCE

The /SILENT qualifier suppresses debugger output when the breakpoint, tracepoint, or watchpoint is taken. This feature is particularly useful with a DO clause on a SET BREAK, SET TRACE, or SET WATCH command. For example, you might set up a DO clause on a SET TRACE command to count the number of times your program accesses a given location. You can specify the /SILENT qualifier so you do not have to see the "trace at . . . " message every time the tracepoint takes effect.

```
DBG>DEFINE/VALUE COUNT=0
DBG>SET TRA/SIL %LINE 10 DO (DEF/VAL COUNT=COUNT+1)
```

The /SOURCE qualifier controls whether you see the source code. If you issue the SET STEP SOURCE command, the debugger displays the source code at every step, breakpoint, tracepoint, and watchpoint.

You can, however, override the default STEP setting by specifying a qualifier. For instance, if you want to see the source code at a particular breakpoint but the default setting is NOSOURCE, you can issue the command SET BREAK /SOURCE.

3.5 Exit Handlers

Exit handlers are procedures that are called whenever an image requests the $EXIT system service. A user program may declare one or more exit handlers. The debugger always declares its own exit handler.

This section discusses the sequence in which exit handlers are executed and explains how to debug them.

3.5.1 Sequence of Exit Handler Execution

At program termination, exit handlers are executed in last-in/first-out order (LIFO); that is, the most recently declared exit handler is executed first, then the next most recently declared exit handler, and so on.

The debugger exit handler is the first in and therefore the last out. Consequently, at program termination, the debugger exit handler executes after all user-declared exit handlers have executed.

3.5.2 Debugging Exit Handlers

To debug an exit handler, you must first set a breakpoint in that exit handler. Then, you must cause that exit handler to execute either by including in your program an instruction that invokes the exit handler or by allowing your program to terminate. (At program termination, the system begins executing exit handlers in last-in/first-out order.) When the exit handler executes, the breakpoint will be activated and control returned to the debugger, which will display its prompt. You can then enter debugger commands.

3.5.3 Identifying Exit Handlers

The SHOW EXIT_HANDLERS command gives a display of the exit handlers that your program has declared. The exit handler routines are displayed in the order that they will be called (that is, last in, first out). The routine name is displayed symbolically, if possible. Otherwise its address is displayed. The debugger's exit handlers are not displayed.

For example:

```
DBG> SHOW EXIT_HANDLERS
exit handler at STACKS\CLEANUP
```

4 Symbol References and Their Interpretation

In the context of debugging, symbols are strings of characters that represent files, program modules, routines, program locations, variables, arrays, or any other definable units. This chapter explains what you need to know about symbols to debug your program.

4.1 Symbolic Debugging

Symbolic debugging means debugging using symbols, instead of virtual addresses, to refer to memory locations.

For the debugger to interpret symbols, information about them (in the form of symbol records) must be present in the executable image at run time.

You control which symbol records are available at run time by using the /DEBUG command qualifier with the compile and LINK commands.

By default, symbol records for traceback information are available at run time. Traceback information consists of symbol records that describe module names, routine names, and compiler-assigned line numbers. Traceback information is used both by the debugger and by the traceback utility. The traceback utility uses these symbol records to display the call stack when a program terminates abnormally.

Example 4–9 shows a traceback display of the call stack.

Example 4–9 Traceback Information

```
%PAS-F-ERRACCFIL, error in accessing file PAS$OUTPUT
%PAS-F-ERROPECRE, error opening/creating file
%RMS-F-FNM, error in file name
%TRACE-F-TRACEBACK, symbolic stack dump follows

module name     routine name     line      rel PC      abs PC

PAS$IO_BASIC    _PAS$CODE                  00000192    00001CED
PAS$IO_BASIC    _PAS$CODE                  0000054D    000020A8
PAS$IO_BASIC    _PAS$CODE                  0000028B    00001DE6
EIGHTQUEENS     EIGHTQUEENS      59        00000020    000005A1
```

To prevent the inclusion of symbol records for traceback information in the executable image, you can specify /NODEBUG with the compile command or /NOTRACEBACK with the LINK command. Note however that specifying /NOTRACEBACK also disables the debugger. In general, if you intend to debug your program, you want to include traceback information in the executable image. On the other hand, when your program is fully debugged, you might want to prevent the inclusion of traceback information in the executable image in order to reduce the size of the executable image file.

Sections 4.2.1 and 4.2.2 describe how to use the /DEBUG command qualifier to include symbol records in the executable image. Section 4.2 describes the symbol tables that contain these records.

However, even if the required symbol records are present in the executable image at run time, the debugger will be able to access them only if they are present in its run-time symbol table (RST). To put symbol records in the RST, you use the SET MODULE command. Section 4.5.2 describes how to use these commands.

Finally, given that the required symbol records are present in the RST, it might also be necessary to specify the program region (or scope) in which a particular symbol is to be interpreted. For example, symbols with the same name, but with declarations in different routines, must be differentiated from one another. In this case, you must prefix the symbol name with a path name or use the SET SCOPE command to define the scope. Section 4.4 discusses these topics.

4.2 Symbol Tables Used by the Debugger

Symbol tables contain symbol records that the debugger uses to associate a symbol with a program location (that is, a virtual address). In addition, by means of some symbol records, the debugger can associate attributes with a program location, such as the length of that location and its data type.

The debugger uses the following three symbol tables, which are described in the following sections.

4.2.1 Debug Symbol Table

When the linker creates an executable image, it creates a debug symbol table (DST) unless you specify the /NOTRACEBACK qualifier with the LINK command. The linker includes the DST in the executable image.

If you specify the /DEBUG qualifier with the LINK command, the linker includes in the DST all symbol records that are present in the object module(s) from which it creates the executable image. Thus, if symbol records for local symbols are present in the object module(s) (because you specified /DEBUG at compile time), the DST will contain symbol records for local symbols and for traceback information.

If you do not specify the /DEBUG qualifier with the LINK command, the linker includes only symbol records for traceback information. Thus, even if symbol records for local symbols are present in the object module(s), they are not included in the executable image.

A DST symbol record typically contains the name of a symbol, its type, its length, and its value or address. The type information may simply mark the symbol as a routine or a label, or it may define a more or less complex data type. Similarly, the address information may be a simple virtual address or value, or it may specify a more or less complex way of computing the symbol address.

At run time, the debugger uses the symbol records in the DST to build the run-time symbol table (RST). For more information on the RST, refer to Section 4.2.3.

4.2.2 Global Symbol Table

When the linker creates an executable image, it creates a global symbol table (GST). The linker includes the GST in the executable image only if the /DEBUG qualifier is specified with the LINK command.

The linker creates the GST from its own internal symbol table. As a result, symbol records in the GST contain no more information than the linker needs to do its job. Symbol records in the GST describe all global symbols in the image, such as routine names, procedure entry points, and global data names. Symbol records for these global symbols associate symbol names with virtual addresses or values, but they do not contain information about data types. See the description of the VAX/VMS Linker in the *VAX/VMS Linker Reference Manual* for information on the content and format of these symbol records.

The debugger uses the GST only if it cannot locate needed information in the DST. If a symbol is represented in both the DST and the GST, the DST symbol record usually contains more information about the symbol than the GST symbol record.

4.2.3 Run-Time Symbol Table

When the debugger is activated at run time, it uses available symbol records in the DST and GST to generate symbol entries for the run-time symbol table (RST).

The RST allows the debugger random access to symbols during a debugging session. Its purpose is to allow efficient access to symbol records contained in the DST and GST. Whenever you mention a symbol in a debugger command, the debugger checks the RST for the information it needs to interpret that symbol. If there is no entry for a symbol in the RST, you cannot access that symbol.

When the debugger is initialized at run time, it allocates enough memory to allow the RST to contain symbol records for the six largest modules in the program, approximately. However, VAX/VMS does not allow memory allocation for the RST if such allocation causes the process to exceed its virtual address quota as established by system parameters.

At debugger start up, symbol records for all modules are not automatically included in the RST. Instead, if the debugger is activated by the RUN command, symbol records for the module containing the transfer address are included in the RST. If the debugger is activated by the DEBUG command, symbol records for the module containing the current PC are included in the RST.

Symbol records for all other modules are included in the RST by *setting the module*. Module setting is the process by which all the symbol records of a particular module are included in the RST. Module setting makes the symbols of a module accessible to debugger commands and is accomplished in two ways: automatically, by using dynamic module setting; and manually by using the SET MODULE command.

Dynamic module setting is initially enabled by default when you invoke the debugger. In dynamic module setting, the debugger sets certain modules automatically for you: whenever the debugger prompt is displayed (whenever the debugger interrupts execution), the debugger automatically sets the

module enclosing the PC location and issues an informational message. If the module is already set, dynamic module setting has no effect. For example, suppose you are at a call of routine SUBR and decide to step into SUBR:

```
DBG> STEP/INTO

stepped to routine SUBR
%DEBUG-I-DYNMODSET, setting module SUBR
```

Dynamic module setting, by setting the module surrounding the current PC location, makes all the symbols of that module accessible to debugger commands.

If you want to set a module that has not been set dynamically (to access symbols declared in arbitrary modules) you need to use the SET MODULE command. For example:

```
DBG> EXAMINE X
%DEBUG-W-NOSYMBOL, symbol 'X' is not in the symbol table
DBG> SET MODULE SWAP
DBG> EXAMINE X
SWAP\X:  17
```

Symbol records can be removed from the RST by means of the CANCEL MODULE command; removing symbol records from one or more modules makes room in the RST for the symbol records of other modules.

Note that dynamic module setting makes the debugger easier to use; however, it may slow the debugger down as more and more modules are set. The debugger does not cancel modules for you. If performance becomes a problem, you can use the CANCEL MODULE command selectively, or you can turn off dynamic module setting by issuing the SET MODE NODYNAMIC command. You can reenable dynamic module setting with the SET MODE DYNAMIC command.

If the debugger cannot find the information it needs to interpret a symbol, one of the following conditions exists:

- The symbol reference is incorrect. In this case, you can check the symbol reference for spelling and syntax errors, and then correct them.
- The symbol refers to more than one entity. In this case, you can make the symbol reference unique by specifying the symbol with a path-name prefix or by using the SET SCOPE command.
- Symbol records for the module containing that symbol are not present in the RST. In this case, you can issue the SET MODULE command to include the symbol records for that module.
- Symbol records for the symbol are not present in the executable image. In this case, you will not be able to use the symbol during the debugging session. To use the symbol, you will have to compile, link, and run the program again, taking care to specify the /DEBUG command qualifier. See Sections 4.2.1 and 4.2.2 for information on using the /DEBUG command qualifier.

4.3 Kinds of Symbols

Symbols differ from one another not only in what they represent but also in the contexts in which they can be interpreted.

This section describes

- Symbols that can always be interpreted in a debugging session
- Symbols that can be interpreted in a debugging session if you define them during the session
- Symbols whose interpretation depends on many factors, such as whether the /DEBUG qualifier was specified at compile time and which modules' symbols are in the RST

4.3.1 Debugger Permanent Symbols

Debugger permanent symbols are symbols that are known to the debugger in any debugging context. Thus, you can use them at any time in a debugging session. The following are the debugger permanent symbols and their referents:

- The characters **%R** followed by a numeral from 0 through 11 represent the corresponding VAX general purpose registers, such as %R0, %R1, %R2, %R3, %R4, %R5, . . . %R11. In general, these symbols are debugger permanent symbols.
- **%PC** represents the program counter and is a debugger permanent symbol.
- **%PSL** represents the processor status longword and is a debugger permanent symbol.
- **%SP** represents the stack pointer and is a debugger permanent symbol.
- **%AP** represents the argument pointer and is a debugger permanent symbol.
- **%FP** represents the frame pointer and is a debugger permanent symbol.
- The period represents the current entity; you can also use %CURLOC. You may use them to refer to the program location last referenced by an EXAMINE or DEPOSIT command. See Section 5.2.6 for more information.
- The circumflex (^) represents the logical predecessor of the current entity; you can also use %PREVLOC.
- The backslash (\) represents the value last displayed by an EVALUATE, EXAMINE, or DEPOSIT command; you can also use %CURVAL.
- <RET> represents the next location; you can also use %NEXTLOC.

You can use %CURLOC, %PREVLOC, %CURVAL, and %NEXTLOC when you cannot use the symbols they represent because the symbols would be ambiguous or because you cannot use <RET> in the middle of an expression.

You may leave out the percent sign in a reference to a register (for example, abbreviate %R0 as R0). However, if you do not use the percent sign, the debugger may interpret these symbols as program variables you have defined, not as debugger permanent symbols. The debugger interprets these symbols as debugger permanent symbols only if your program does not contain variables of the same names.

4.3.2 Symbols Created by the DEFINE Command

During a debugging session, you can use the DEFINE command to create a new symbol or to change the value of a currently existing symbol.

For example, if you find yourself frequently referencing a difficult-to-remember, nonsymbolic program location, you can define a symbol to represent that program location. Then, for the remainder of the debugging session, you can refer to that program location by the symbol that represents it.

Symbols created with the DEFINE command are deleted when you end the debugging session.

As a result of the following command, MIN1 may be specified during a debugging session instead of PROGRAM\ROUTINEA\BLOCKB\MINIM+20:

```
DBG>DEFINE MIN1 = PROGRAM\ROUTINEA\BLOCKB\MINIM+20
```

See the Command Dictionary for a full description of the DEFINE command.

4.3.3 Program Symbols

Program symbols (also called identifiers) are the symbols you use when you write your program.

A program symbol is interpreted according to how and where it is declared and where it is used.

A declaration of a symbol specifies attributes permanently associated with the symbol, such as the data type, how much storage is allocated, and so on. The symbol is interpreted by means of these attributes.

Program symbols fall into one of two categories:

- Data names, which identify data operated on by the program's executable statements
- Program unit labels and program location labels, which identify or name programs, procedures, lines, or statements

The following sections discuss the syntactic and semantic differences among program symbols. Syntactically speaking, symbols range from simple characters or names to complex entities composed of several names separated by various delimiting characters.

The following subsections present the basic types of program symbols in order to demonstrate the debugger's capacity to interpret the full range of language-specific symbol syntax.

For those readers who are already familiar with the varieties of program symbols, it may be enough to know that the debugger interprets program symbols in the source-language syntax. Thus, in general, if a symbol is legal in the source language, the debugger is capable of interpreting it.

4.3.3.1 Simple Symbols

The debugger interprets simple symbols according to the syntactic rules of the source language.

A simple symbol is a program symbol that does not contain symbolic prefixes or suffixes that further qualify the data item represented by the symbol.

A simple symbol may refer to an individual data item or to an entire range of data items.

A simple symbol may be a data name that represents, for example, a numeric constant (X), an array (ARR), or a character string (A5$).

If a symbol is legal in the source language, the debugger is generally able to interpret it. For example, if language is set to VAX BASIC, the debugger interprets the symbol T$ as a legal symbol representing a character string.

However, in some languages, symbol names may exist that are not considered identifiers in the syntax of the currently set language. For instance, in COBOL, sections and paragraphs can have numeric names such as "12." In BLISS, the compiler generates names of the form "P.xxx" for PLITs, even though the period is not a valid character in BLISS identifiers.

The %NAME built-in symbol causes the debugger to interpret its argument as a name, whether or not it appears to be a name. It has either of the following forms:

```
%NAME id-char-string

%NAME 'any-char-string'
```

The quotes are only needed for one of the following reasons:

- If the name has a semicolon (;) (which would ordinarily terminate the command)
- If the name has lower-case alphabetics (which would ordinarily be uppercased)

Examples follow:

```
DBG> SET BREAK %NAME 12
DBG> EXAMINE %NAME 'P.AAA'
```

4.3.3.2 Subscript-Qualified Symbols

The debugger interprets subscript-qualified (or subscripted) symbols in the source language syntax.

A subscripted symbol is a symbol used to refer to a data item in an array.

Because an array consists of an ordered set of data items, you can identify an array item by specifying the array name and the item's position in the array. The position of an array item is indicated by one or more subscripts. For example, (I+2) is a valid subscript in the subscripted symbol ARR(I+2).

The syntax of the subscripted symbol depends on the language that is currently set. If, for instance, you want to examine the first item in the array ARR and the currently set language is FORTRAN, you would give the command EXAMINE ARR(1). However, a similar command in PASCAL would look like this: EXAMINE ARR[1].

The number of subscripts used to identify a data item in an array indicates the dimensions of the array. Thus, the third member of a one-dimensional array named ARR is identified by the subscripted symbol ARR(3), and the array member located in the third row, second column of a two-dimensional array named TAB is identified by the subscripted symbol TAB(3,2).

You can, however, specify or examine an *array slice* instead of only one item in the array. This capability allows you to see the items in the array as a whole. You specify array slices much the same way you specify ordinary subscripted symbols. For example, the symbol ARR(1,2:4,6) is a valid FORTRAN array slice. It would produce an array slice made up of the items ARR(1,4), ARR(1,5), ARR(1,6), ARR(2,4), ARR(2,5), and ARR(2,6).

In effect, an array slice is simply an array that is a subset of the parent array.

4.3.3.3 Structure-Qualified Symbols

The debugger interprets symbols that identify members of data aggregates according to the syntactic rules of the source language.

A data aggregate that consists of data items of different types is called a data structure or a record. Records contain data items that may contain subitems that may contain sub-subitems, and so on.

For example, the following display is a declaration, in VAX PASCAL, of a record Person:

```
TYPE PERSON = RECORD
  NAME : PACKED ARRAY[1..20] OF CHAR;
  AGE  : INTEGER;
  SEX  : (MALE,FEMALE);
  SALARY : INTEGER
  END;

VAR COURTNEY : PERSON;
```

In most languages, to reference a field in a record, you must use a structure-qualified symbol containing the name of the desired field and the names of all other fields that contain that field.

For example, if you wanted to reference the field Age in the record above, you would use the structure-qualified symbol COURTNEY.AGE.

The period is a delimiter used to separate field and record names.

Note that in those languages that permit the nesting of arrays and records, symbolic references to fields within those records are a combination of subscripted symbol and structure-qualified symbol.

Generally, a legal symbol in the source language is capable of being interpretated by the debugger.

You can also specify or examine an entire aggregate, which means all the items that make up the aggregate. You do not have to specify each of the items individually. The aggregate output from a record shows the name and value of each item in the record. Similarly, the aggregate output from an array shows the subscript and value of each array item.

4.3.3.4 Pointer-Qualified Symbols

Some languages use pointer-qualified symbols to reference data items that are not bound by the compiler to specific memory addresses. As with other source-language symbols, the debugger interprets pointer-qualified symbols in the syntax of the source language.

For example, to indicate a pointer type in VAX PASCAL, you specify the name of the base type preceded by a circumflex (^). Thus, if the record PERSON in Section 4.3.3.3 is the base type, the pointer variable P is declared as

```
VAR P : ^PERSON;
```

To access the entire record, the pointer-qualified symbol P^ is used.

To access a field within the record, include the name of the field, as shown below.

```
P^ .SALARY
```

4.4 Symbol Resolution in the Source Language

The recognition or resolution of a symbol during a debugging session depends on the resolution of the symbol in the source language and on the debugger context in which the symbol is mentioned.

This section discusses the factors that influence the resolution of a symbol within the source language. These factors are the context in which the symbol declaration occurs and the presence or absence of a global attribute in the symbol declaration.

4.4.1 Program Context of Symbol Declarations

Typically, programs contain symbol declarations. A symbol declaration associates a symbol to an entity and associates with that entity certain properties (such as type or storage allocation). Declarations may be implicit; however, this discussion does not distinguish between implicit and explicit declarations.

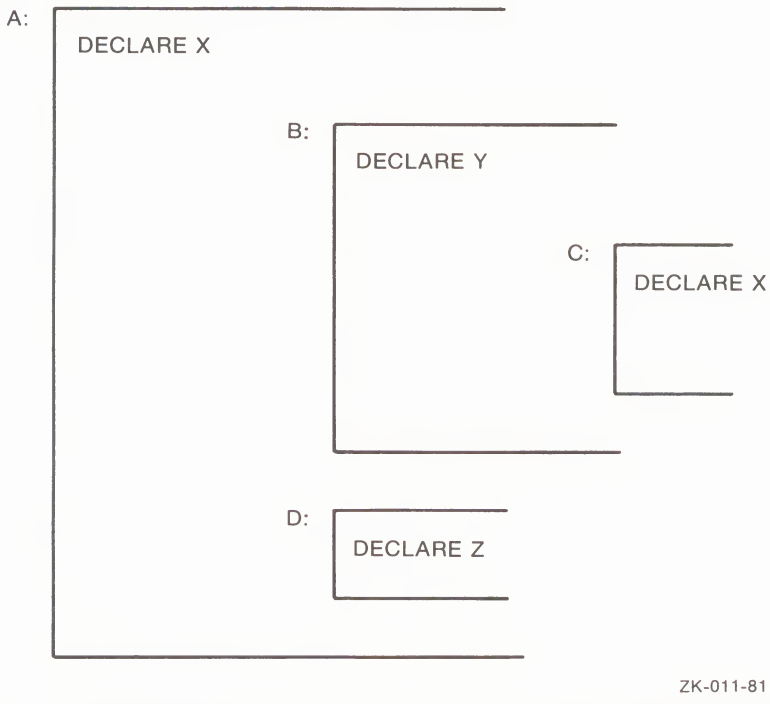
The scope of a declaration is the set of program locations wherein the symbol is interpreted as representing the entity bound to it in that declaration.

The same symbol may be declared more than once. Declarations of the same symbol have disjoint or nonoverlapping scopes, and each declaration binds the symbol to a different entity (unless the symbol is global).

If a declaration occurs in a program unit that does not contain other program units, the scope of the declaration is the program unit in which it occurs. If a declaration occurs in a program unit that contains other (nested) program units, the scope of the declaration is the program unit in which the declaration occurs and all other nested program units that do not themselves contain declarations of the same symbol.

Figure 4–2 shows nested program units containing several symbol declarations. The symbol X is declared in two program units. The scope of X declared in A is A, B, and D, but not C, because X is redeclared in C. The scope of X declared in C is C; the scope of Y is B and C; and the scope of Z is D.

Figure 4–2 Scope of Symbol Declarations



4.4.2 Global Symbols

A global symbol is a symbol whose declaration contains a global attribute.

The presence or absence of a global attribute in a declaration influences the resolution of that symbol.

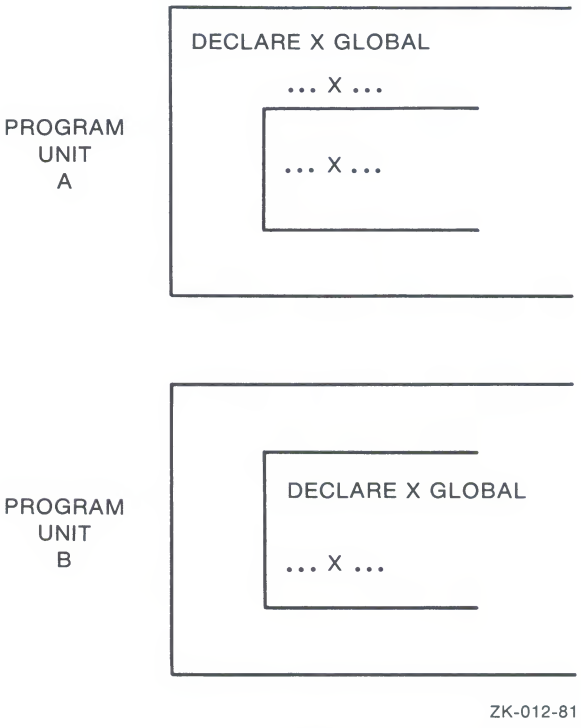
Global symbols make possible references to the same entity by program units that do not contain one another (nonnested program units). To accomplish this, each program unit declares the symbol with the global attribute, though not necessarily with all other attributes identical.

Thus, a global symbol is bound by several declarations to the same entity. This contrasts with multiple declarations of the same local (or nonglobal) symbol where each declaration associates the symbol with a different entity.

Strictly speaking, the scope of the declaration of a global symbol is no different from the scope of the declaration of a nonglobal symbol; however, the entity referred to by that symbol may be capable of interpretation outside of the scope of its declaration in any one program unit, namely, in those other program units that also declare it.

Figure 4–3 depicts declarations of global symbol X in two nonnested program units. Both program units also make references to X. Because X is declared with the global attribute in both program units, references to X in either program unit are references to an identical memory location, the entity represented by X.

Figure 4–3 Global Symbol X



4.5 Symbol Resolution in the Debugger

As described in Section 4.4.1, a symbol in the source language is interpreted as representing the entity associated with it in that declaration within whose scope the symbol occurs.

In debugging, however, wherein random access to symbols anywhere in the program may be necessary, you use path names to distinguish multiple declarations of the same symbol.

A path name consists of one or more program location labels that serve to specify a program location or range of program locations. Hence, a path name is used to "locate" a symbol within the scope of the declaration that binds it to the entity you want to reference. Of course, the program location(s) indicated by the path name must be within the scope of some declaration of the symbol; otherwise the symbol cannot be interpreted at all.

Thus, when you are debugging a program containing multiple declarations of the same symbol and you want to indicate to the debugger that you wish the symbol to be interpreted as representing one of several possible entities, you simply specify a path name with the symbol. The debugger then interprets the symbol as representing the entity associated with it in the declaration whose scope includes the program location(s) denoted by the path name.

If you specify a path name as a parameter in the SET SCOPE command, you are in effect establishing that path name as a default path-name prefix, to be used in all symbol references that do not already contain a path-name prefix. See Section 4.5.3 for more information on the SET SCOPE command.

You may specify as many path-name parameters as you like in the SET SCOPE command; the debugger attempts to interpret the symbol using the first path name listed. If that fails, the debugger uses the second path name and continues in this manner until it finds a path name that identifies a program location within the scope of a declaration of the symbol. The debugger then interprets the symbol as representing the entity identified in the declaration.

If you do not specify a path-name prefix when you use a symbol in a debugger command and you have not specified default path names using the SET SCOPE command, the debugger looks up a symbol according to the following default scope. By default, the debugger looks up a symbol according to the scope search list *0,1,2, . . . ,N*, where *N* is the number of calls in the call stack. This scope search list is based on your current PC and changes dynamically as your program executes. The default scope means that a symbol lookup such as "EXAMINE X" first looks for X in the routine that is currently executing (scope 0); if no X is visible there, the debugger looks in the caller of that routine (scope 1), and so on down the call stack; if X is not found in scope N, the debugger searches the rest of the run-time symbol table (RST), then searches the global symbol table (GST), if necessary.

When you use a symbol X in a debugger command, you may or may not use a path-name prefix. If you use a path-name prefix, the debugger interprets X as if it appeared in the program location(s) defined by that path name. The debugger issues an error message if there is no declaration of X whose scope includes the program location(s) defined by the path name. If you do not use a path-name prefix, the debugger attempts to interpret X using default path name(s), either (1) path name(s) you specified in the SET SCOPE command, or (2) if you have not specified a path name, the path name indicated by the program location containing the current PC.

For example, assume you enter the following commands:

```
DBG>SET SCOPE A\B,C
DBG>EXAMINE X
```

You have not specified a path-name prefix in the EXAMINE command. Consequently, the debugger uses A\B\ as a path-name prefix and attempts to locate a declaration of X whose scope includes the program location identified by the path name A\B\. If the debugger finds such a declaration of X, it uses that declaration to interpret X in the EXAMINE command. If it does not find such a declaration of X, the debugger then repeats the procedure using C\ as a path-name prefix.

If the debugger fails to find a declaration of X whose scope includes the program locations identified by the default path names, it searches the entire run-time symbol table (RST) for a declaration of X. If the debugger locates one, it resolves X using that declaration. If it locates more than one declaration of X, it issues an error message to the effect that an ambiguous reference has been made.

If the debugger does not find a declaration of X in the RST, it searches the global symbol table (GST). If the debugger does not find a declaration of X in the GST, it displays a message indicating that the symbol could not be found. See Section 4.2.3 for information on what to do.

To utilize fully the debugger's capacity for distinguishing among multiple declarations of the same symbol, you must be thoroughly familiar with how path names are specified.

4.5.1 Specifying Path names

A path name is a string of program location labels that identifies a program location or a range of program locations. A path name is used in two contexts:

- As a prefix to a symbol in a debugger command
- As a parameter in the SET SCOPE command

The labels in a path name are strung together so that each label designates a program unit that contains the program unit designated by any label to its right. Thus, if PROG is a label appearing in a path name, labels to the left designate "containing" program units, and the labels to the right designate "contained" program units.

The backslash character (\ ) is used both to separate path-name elements from one another and to separate the entire path name from the symbol to which it is prefixed. Note that if the path name is specified as a parameter in the SET SCOPE command, it is not followed by a symbol; therefore, the rightmost path-name element in the path name is not followed by a backslash. On the other hand, if the same path name is used to prefix a particular symbol in a command, the rightmost path-name element is followed by a backslash to separate it from the symbol.

The kinds of labels used in a path name vary somewhat from language to language; however, in general, a path name always includes a module name and usually includes one or more of the following elements:

- Routine name(s)
- Block name(s)
- Invocation number
- Line number
- Numeric label

A routine is a separately invocable program unit, that is, a program unit that may be activated by a call. A block is a program unit that is activated in normal execution sequence (inline); it is not separately invocable. A routine may contain one or more blocks and routines. A block may contain one or more routines and blocks; that is, there may be nesting of blocks and routines.

The following is a typical path name consisting of module, block, and routine names:

```
INV\PARTS\AUTO\
```

In languages that use line numbers, the %LINE path-name element specifies the source-code line. The following is a path name that uses the %LINE element:

```
MEDIAN\%LINE 330\
```

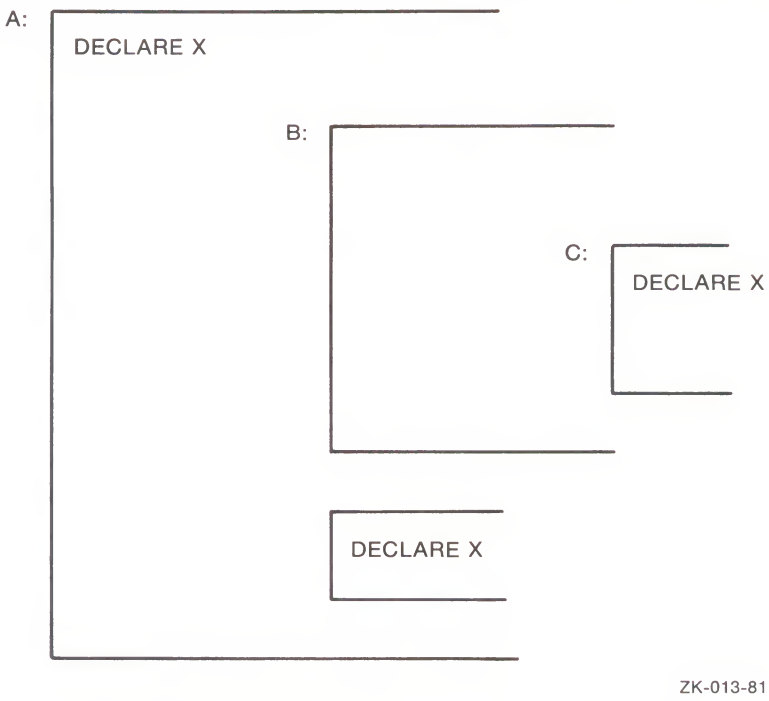
In languages that allow recursion, an invocation number is used in a path name to distinquish among multiple invocations of the same program unit (and therefore among multiple generations of symbols). An invocation number is always associated with a routine name, never a block name; one or more spaces must separate a routine name from an invocation number. (See

Section 4.5.1.2 on invocation numbers for more information.) The following is a path name that specifies an invocation number:

```
MOD\ROUT 1\BLK\
```

Figure 4–4 shows a program containing nested routines and blocks with declarations of the symbol X. Examples 1 through 4 refer to Figure 4–4. In these examples, path names are used as prefixes of the symbol X in the EXAMINE command and as parameters in the SET SCOPE command.

Figure 4–4 Path Names and Scope



4.5.1.1 Path Name Examples

This section contains four examples of path names.

```
DBG>EXAMINE A\X
```

The path-name prefix A\ identifies the range of program locations defined by the program unit A. The debugger attempts to locate a declaration of X whose scope includes A. Since such a declaration of X appears in A, the debugger interprets X as representing the entity declared in A. The debugger then examines the value of X.

The scope of the declaration of X in A consists both of those locations in A that are not also in other blocks and of B. C and the unnamed (anonymous) block are not in the scope of the declaration of X in A because they also contain declarations of X.

```
DBG>EXAMINE A\B\C\X
```

The path-name prefix A\B\C\ identifies program unit C. The debugger searches for a declaration of X whose scope includes C. It finds such a declaration in C and interprets X as representing the entity declared in C. The debugger then examines the value of X.

Remember that a symbol redeclared in a contained program unit is interpreted according to the declaration in that unit, not the declaration in the outer unit. Thus, the symbol X in C has a different meaning (as declared) than the symbol X that is declared in A.

```
DBG>EXAMINE A\%LINE 110\X
```

The path name A\%LINE 110\ identifies a program location within the anonymous block because line number 110 is contained in the anonymous block.

The debugger searches for a declaration of X whose scope includes line number 110. It finds such a declaration in the anonymous block and interprets X as representing the entity declared therein. The debugger then examines the value of X.

Note that only by means of the %LINE path-name element is it possible to identify program locations in an anonymous block. Thus, in this example, without using the %LINE path-name element, it would be impossible to reference the entity represented by X as declared in the anonymous block.

```
DBG>SET SCOPE A\B
DBG>EXAMINE X
```

Since no path-name prefix is used in the EXAMINE command, the default path-name prefix A\B\ is attached to X. The debugger searches for a declaration of X whose scope includes B, the range of locations specified by A\B\. It locates such a declaration in A and interprets X as representing the entity declared in A. The debugger then examines the value of X.

4.5.1.2 Path Name Completion

In programs where deep nesting of program units occurs, the naming of every containing program unit in a path name can be burdensome. To make it easier for you to specify path names, the debugger supports path name completion.

A complete path name is a path name that mentions every path-name element: all routines and blocks, as well as label and line numbers, if applicable. An incomplete or abbreviated path name is a path name that does not mention every path-name element in the specification.

When the debugger encounters a path name, it first determines whether the path name is complete or incomplete. If it is complete, the debugger uses the path name to resolve the symbol reference. If it is incomplete, the debugger determines whether it is an abbreviation for a complete path name and, if so, whether it is an unambiguous abbreviation.

An incomplete path name is an abbreviation for a complete path name if

- The path-name elements in the incomplete path name appear in the complete path name in the same order.
- The rightmost element in the complete path name is the rightmost element in the incomplete path name.

For example, the incomplete path name

```
RUG\%LINE 784\
```

is an abbreviation for the complete path name

```
MAT\CLR1\RUG\%LINE 784\
```

because (1) RUG\ and %LINE 784\ in the incomplete path name are in the same order as in the complete path name and (2) the rightmost element %LINE 784\ in the complete path name is the rightmost element in the incomplete path name.

The abbreviated path name is unambiguous if there is not more than one complete path name for which it is an abbreviation.

For example, let us assume that two different subroutines, both named SUB, are declared in program units PROG and QUAR, respectively. The incomplete path name SUB is ambiguous because there are two complete path names, PROG\SUB and QUAR\SUB, for which it is an abbreviation.

An incomplete path name is acceptable to the debugger so long as it is an abbreviation (according to the definition above) for one (and not more than one) complete path name.

Unique symbols can always be specified without path-name qualification.

Global symbols may be specified by preceding the symbol with a backslash (\). For example, \X indicates that X is a global symbol.

4.5.1.3 Invocation Numbers

Routines are separately invocable program units; that is, they are activated by calls, rather than by inline execution. Thus during the execution of a program, there may be several simultaneous invocations of a routine or none at all.

If a symbol representing a dynamic entity is declared in a routine or in a block within a routine, the entity represented by that symbol is generated anew each time the routine is invoked. For each invocation of the routine, there is a corresponding generation of the entity. Invocation numbers are used in path names to denote particular routine invocations (and thus particular generations of an entity).

Invocation numbers are nonnegative decimal integers inserted in the path name following the name of the rightmost routine in the complete path name. The number zero (0) denotes the most recent invocation of the innermost (most deeply nested) routine; the number one (1) denotes the invocation before that; and so on.

For example, if a module MOD contains a routine ROUT that contains a block BLK, then the path name that identifies the generation of BLK that resulted from the most recent invocation of ROUT is

```
MOD\ROUT 0\BLK
```

The path name that identifies the generation of BLK that resulted from the previous invocation of ROUT is

```
MOD\ROUT 1\BLK
```

Every complete path name that contains the name of a separately invocable entity has an invocation number. When an invocation number is not present in a path name that contains the name of a routine, the debugger implicitly assumes the most recent invocation of the routine and supplies the default invocation value zero (0).

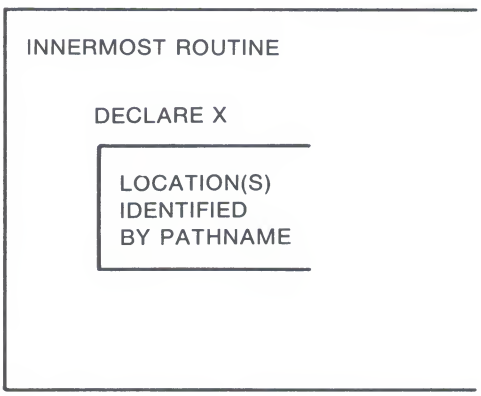
Every path name containing the name of more than one routine specifies an invocation of the innermost routine. In a path name, invocation numbers cannot be associated with a routine name that appears to the left of another routine name.

How you use a path name with an invocation number to specify a particular generation of a dynamic entity depends on the relative positions of the innermost routine and the program unit containing the declaration of the entity. The symbol bound to the dynamic entity you want to reference may be declared in any of the following:

- The innermost routine
- A block contained in the innermost routine
- A program unit that contains the innermost routine

These three possibilities give rise to the three program situations illustrated in Figures 4–5, 4–6, and 4–7.

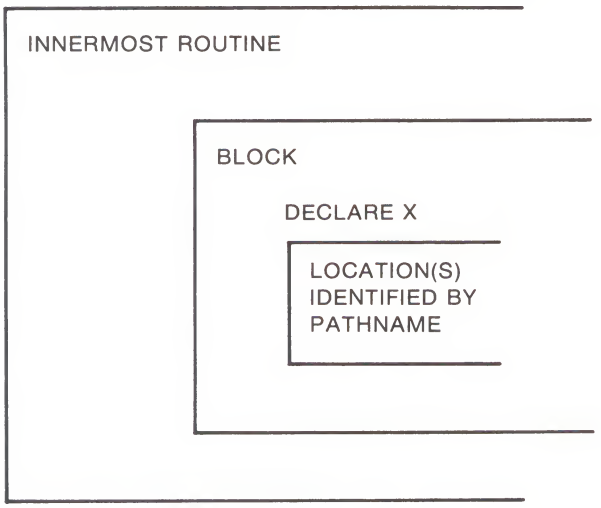
Figure 4–5 Symbol Declaration in the Innermost Routine



ZK-014-81

If the symbol is declared in the innermost routine, the debugger references the generation of the entity corresponding to the invocation of the routine (if there is one) denoted by the path name. Figure 4–6 illustrates this program situation.

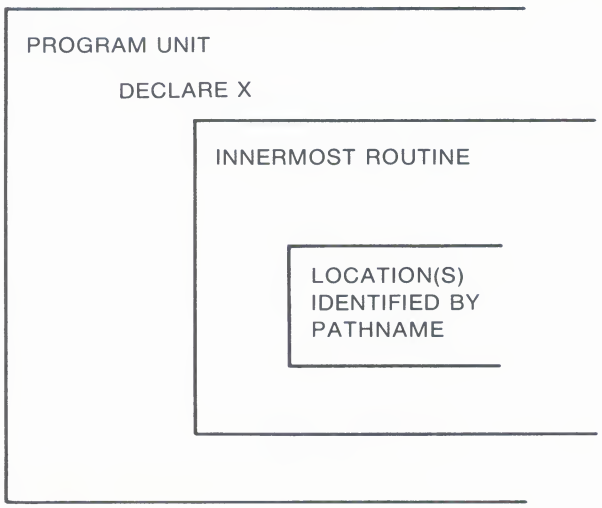
Figure 4–6 Symbol Declaration in a Contained Block



If the symbol is declared in a block contained in the innermost routine and if program execution has reached that block, then the debugger references the generation of the entity corresponding to that block activation.

However, if program execution has not reached the block containing the declaration (as would be the case, for example, if you set a breakpoint at a location in the routine before the location that marks the beginning of the block), then the generation of the symbol specified by the path name does not exist, and the debugger issues an error message to that effect. Figure 4–7 illustrates this program situation.

Figure 4–7 Symbol Declaration in a Containing Program Unit



Some invocation of a program unit provides the context for the invocation of the innermost routine denoted by the path name. The debugger references the generation of the entity corresponding to that invocation of the program unit. Figure 4–7 illustrates this program situation.

Note that when you specify an invocation number, you must leave one or more spaces or tabs between it and the separately invocable routine.

4.5.2 The SET, SHOW, and CANCEL MODULE Commands

The SHOW MODULE command displays the following information about one or more modules in your program:

- The module name
- The presence (or absence) of symbol records for that module in the RST
- The language in which the module is written
- The approximate space (in bytes) required in the RST for symbols in that module
- The total number of modules selected for display
- The number of unused bytes in the space allocated for the RST

The SHOW MODULE command has the following format:

```
SHOW MODULE [/SHARE] [module-name [,...]]
```

The /SHARE qualifier controls whether the debugger includes, in the SHOW MODULE display, any shareable images that have been linked with your program but are external to your program. These shareable images are primarily Run-Time Library images, but they also include any shareable

images called by your program. By default (/NOSHARE), no shareable images are selected for display.

To insert symbol records for one, several, or all modules in the RST, you use the SET MODULE command. It has the following format:

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

To include symbol records for one or several modules, specify the module name(s) in the SET MODULE command. In this case, do not specify a command qualifier.

To include symbol records for all modules, specify the /ALL command qualifier in the SET MODULE command. In this case, do not specify a module name or names.

Note that if a parameter in the SET SCOPE command designates a program location in a module whose symbol records are not already in the RST, the debugger copies symbol records of that module into the RST when the SET SCOPE command is executed.

When all the memory space allocated for the RST is occupied by symbol records and you want to include additional symbol records, you have two alternatives. You can either allocate more memory with either the ALLOCATE or the SET MODULE/ALLOCATE command, or you must issue the CANCEL MODULE command in the following format:

```
CANCEL MODULE [/qualifier] [module-name [,module-name...]]
```

To delete symbol records of one or several modules, specify the module name or names in the CANCEL MODULE command. In this case, do not specify a command qualifier.

To delete symbol records of all modules, specify the /ALL command qualifier in the CANCEL MODULE command. In this case, do not specify a module name or names.

Example 4–10 demonstrates the SHOW, SET, and CANCEL MODULE commands.

4.5.3 The SET, SHOW, and CANCEL SCOPE Commands

The purpose of the SET SCOPE command is to indicate one or more path-name prefixes to be used in the interpretation of symbols without path-name prefixes.

The numbers 0, 1, 2, and so on, which appear in path names as invocation numbers, may also be used as parameters in the SET SCOPE command. Used in this way, these numbers are numeric path names.

The numeric path name 0 specifies that symbols without path-name prefixes are to be interpreted as if they appeared in the currently active routine; numeric path name 1, in the program unit that contains the call to the currently active routine; numeric path name 2, in the program unit that contains the call to the program unit that contains the call to the currently active routine; and so on.

If you do not issue a SET SCOPE command, the debugger exhibits a default behavior equivalent to the following SET SCOPE command (here, *n* is the number of calls in the call stack):

```
SET SCOPE 0,1,2, . . . n
```

Example 4–10 Using the SET/SHOW/CANCEL MODULE Commands

```
                VAX-11 DEBUG Version 3.0-3

%DEBUG-I-INITIAL, language is BASIC, module set to "MEANSUB$MAIN"

DBG>SHOW MODULE
module name      symbols   language   size
BAS$MSGDEF       no        BLISS        68
BAS$STOP         no        BLISS       284
MEANSUB$MAIN     yes       BASIC       172
OTS$LINKAGE      no        MACRO       176
total modules: 4.                remaining size: 60784.

DBG>SET MODULE OTS$LINKAGE

DBG>SHOW MODULE
module name      symbols   language   size
BAS$MSGDEF       no        BLISS        68
BAS$STOP         no        BLISS       284
MEANSUB$MAIN     yes       BASIC       172
OTS$LINKAGE      yes       MACRO       100
total modules: 4.                remaining size: 60676.

DBG>CANCEL MODULE MEANSUB$MAIN

DBG>SHOW MODULE
module name      symbols   language   size
BAS$MSGDEF       no        BLISS        68
BAS$STOP         no        BLISS       284
MEANSUB$MAIN     no        BASIC       172
OTS$LINKAGE      yes       MACRO       100
total modules: 4.                remaining size: 60856.

DBG>SET MODULE/ALL

DBG>SHOW MODULE
module name      symbols   language   size
BAS$MGSDEF       yes       BLISS        36
BAS$STOP         yes       BLISS       172
MEANSUB$MAIN     yes       BASIC       172
OTS$LINKAGE      yes       MACRO       100
total modules: 4.                remaining size: 60452.

DBG> SHOW MODULE FOO,MAIN,SUB*
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.
```

This scope search list is based on your current PC and changes dynamically as your program executes. The default scope means that, when you do not specify a path name, a symbol lookup such as "EXAMINE X" first looks for X in the routine that is currently executing (scope 0); if no X is visible there, the debugger looks in the caller of that routine (*scope 1*), and so on down the call stack; if X is not found in scope N, the debugger searches the rest of the run-time symbol table (RST), then searches the global symbol table (GST), if necessary.

You can set up a symbol scope search list by specifying more than one parameter in the SET SCOPE command. The debugger uses the first parameter specified as a path-name prefix to interpret a symbol without a path-name prefix. If this interpretation fails, the debugger uses the next parameter listed in the SET SCOPE command in a similar fashion and

continues until it successfully interprets the symbol or until it exhausts the parameters specified in the SET SCOPE command.

The acceptable parameters to the SET SCOPE command are described below.

MODULE\ROUTINE\BLOCK	This is a legal path name. Path names may be complete or incomplete. Note that there may be one or more routines and/or blocks.
0, 1, 2, 3, . . .	These numbers are numeric path names. The numeric path name 0 specifies that symbols without path-name prefixes are to be interpreted as if they appeared in the currently active routine; numeric path name 1, in the program unit that contains the call to the currently active routine; numeric path name 2, in the program unit that contains the call to the program unit that contains the call to the currently active routine; and so on.
Backslash (\)	The backslash (\) specifies that a symbol without a path-name prefix is to be interpreted as a global symbol.

The following is an example of the SET SCOPE command:

```
DBG>SET SCOPE 1, MODB, \
```

This command establishes a symbol scope search list. As a result of this command, the debugger attempts to interpret a symbol without a path-name prefix as if (1) it appeared in the program unit that contains the call to the currently active routine, (2) it appeared in MODB, (3) it were a global symbol.

If you want to examine the current parameters in effect, issue the SHOW SCOPE command.

If you want to change the existing scope search list, issue a SET SCOPE command specifying the desired parameters.

If you want to cancel the existing scope search list, issue the CANCEL SCOPE command.

The CANCEL SCOPE command has the same effect as the SET SCOPE 0 command.

4.5.4 The SHOW SYMBOL Command

You can use the SHOW SYMBOL command to ask the debugger for information that it has in its symbol table. You can request information about a single symbol or about all symbols (by using wildcards).

The following example shows how an attempt to examine X fails because there are two X's in the program. SHOW SYMBOL displays the two instances of X.

```
DBG> EXAM X
%DEBUG-W-NOUNIQUE, X is not unique
DBG> SHOW SYMBOL X
data FOO\X
data FUM\X
DBG> EXAMINE FOO\X
FOO
```

To display type and address information about all the symbols in the RST, enter the following command string:

```
DBG> SHOW SYMBOL/TYPE/ADDRESS *
```

The asterisk (*) is the wildcard character. It matches all symbols.

For more information about SHOW SYMBOL, see the Command Dictionary.

4.6 Debugging Shareable Images

A shareable image is an image that is not directly executable. Shareable images are included in the linking of executable images, and then the shareable image is loaded at run-time when the executable image is run. A shareable image contains "universal symbols" that are visible from the executable image. See the *VAX/VMS Linker Reference Manual* for more information on shareable images.

You debug a shareable image in much the same way as any other image. You first compile and link the shareable image with the /DEBUG qualifier. You need not install the shareable image to debug it. Instead, you can debug your own private copy by pointing a logical name to it. You will be using the debugger's SET IMAGE and SHOW IMAGE commands.

The following is a simple example to illustrate the above points. Suppose MAIN.FOR is the source file for your main image, and SHARE.FOR is the source file for the shareable image that you want to debug. Assume that SHARE.FOR has a single universal symbol, SHARED_ROUTINE, which is a routine called from the main program MAIN.FOR.

To debug SHARE.FOR, first compile and link the shareable image with the /DEBUG option:

```
$ FORTRAN/DEBUG/NOOPT SHARE
$ LINK/SHARE/DEBUG SHARE,SYS$INPUT/OPTION
UNIVERSAL=SHARED_ROUTINE CTRL/Z
```

You have now built the shareable image SHARE.EXE in your current directory. Since it is a shareable image, you cannot run SHARE.EXE directly. Instead, you have to link your main image against it, then point the logical SHARE to your copy of SHARE.EXE, and then run the main image:

```
$ FORTRAN/DEBUG/NOOPT MAIN
$ LINK/DEBUG MAIN,SYS$INPUT/OPTION
SHARE.EXE/SHARE CTRL/Z
$ DEFINE SHARE SYS$DISK:[]SHARE.EXE
$ RUN MAIN

      VAX DEBUG Version 4.4

%DEBUG-I-INITIAL, ...
DBG>
```

Now, suppose you want to set a breakpoint on SHARED_ROUTINE, the routine in your shareable image. To do that, you "set" the shareable image SHARE to bring in its symbol table. The symbol SHARED_ROUTINE is then available to the debugger, and you can set a breakpoint on it:

```
DBG> SET BREAK SHARED_ROUTINE
%DEBUG-W-NOSYMBOL, symbol 'SHARED_ROUTINE' not found
DBG> SHOW IMAGE

 image name                     set    base address    end address

*MAIN                           yes    00000200        000009FF
 SHARE                          no     00001000        00001FFF
```

```
 total images: 2                    remaining size: 32856
DBG> SET IMAGE SHARE
DBG> SHOW IMAGE

 image name                         set    base address    end address

 MAIN                               yes    00000200        000009FF
*SHARE                              yes    00001000        00001FFF

 total images: 2                    remaining size: 32856
DBG> SET MODULE/ALL
DBG> SET BREAK SHARED_ROUTINE
DBG> GO
break at routine SHARED_ROUTINE
10:     subroutine shared_routine(a,b)
DBG>
```

Now that you have set image SHARE and all its modules and have reached the breakpoint at SHARED_ROUTINE, you can debug it in the normal fashion (for example, step through the routine, examine variables, and so on).

5 Referencing Program Locations

In debugging, you make references to program locations in order to stop program execution, to examine and deposit values, and to set breakpoints, watchpoints, and tracepoints.

All references to program locations are called address expressions. An address expression may be a simple address or an expression consisting of one or more simple addresses, operators, and delimiters.

Generally, when the debugger interprets an address expression, the results are a program location and a type that is associated with the contents of that location.

The first part of this chapter contains a discussion of type; the second part, a description of various simple addresses; and the third part, a discussion of the use of operators and delimiters to form expressions.

5.1 Type

When a symbol is declared in the source language, the language compiler associates a type with the entity that the symbol represents. Thereafter, the entity is interpreted in that language-dependent (or compiler-generated) type. In general, the debugger understands language-dependent types that are associated with entities by declaration in the source language.

However, because all references are not symbolic, an entity may not have a compiler-generated type. In this case, the entity is interpreted using a debugger default type. The default type is used only when the entity has no type associated with it already.

The following are the debugger types:

ASCII:n	Designates the ASCII character string type (length n bytes)
ASCIC	Designates the counted ASCII character string type
ASCID	Designates a string descriptor pointing to an ASCII string
ASCIW	Designates a varying ASCII character string
ASCIZ	Designates a zero-terminated ASCII character string
BYTE	Designates the byte integer type (length 1 byte)
DATE_TIME	Designates the 64-bit VMS representation of date and time.
D_FLOAT	Designates the D_floating type (length 8 bytes), whose value range is the same as the F_floating type but with approximately 16 decimal digits precision
FLOAT	Designates the F_floating type (length 4 bytes), whose values may range from $.29 \cdot 10^{--38}$ to $1.7 \cdot 10^{38}$ with approximately 7 decimal digits precision
G_FLOAT	Designates the G_floating type (length 8 bytes), whose values may range from $.56 \cdot 10^{--308}$ to $.9 \cdot 10^{308}$ with approximately 15 decimal digits precision

H_FLOAT	Designates the H_floating type (length 16 bytes), whose values may range from $.84*10^{--4932}$ to $.59*10^{4932}$ with approximately 33 decimal digits precision
INSTRUCTION	Designates the VAX instruction type (variable length)
LONGWORD	Designates the longword integer type (length 4 bytes)
OCTAWORD	Designates the octaword integer type (length 16 bytes), whose values are signed integers in the range -2^{127} to $2^{127}-1$
PACKED:n	Designates a packed decimal type (length n digits).
QUADWORD	Designates the quadword integer type (length 8 bytes), whose values are signed integers in the range -2^{63} to $2^{63}-1$
WORD	Designates the word integer type (length 2 bytes)

To change the default type, specify the desired type as a parameter in the SET TYPE command.

For example, the following command changes the default type to 6-byte ASCII string:

```
DBG>SET TYPE ASCII:6
```

As a result of this command, the debugger interprets any entity without a compiler-generated type as a 6-byte ASCII string.

To see what the current default type is, issue the SHOW TYPE command.

If you want all entities, even those with compiler-generated types, to be interpreted in one of the above types, you establish an override type by specifying the desired type as a parameter in the SET TYPE/OVERRIDE command.

For example, the following command directs the debugger to interpret all entities, even those with compiler-generated types, as VAX instructions:

```
DBG>SET TYPE/OVERRIDE INSTRUCTION
```

To see what override type is in effect, issue the SHOW TYPE/OVERRIDE command.

To cancel an override type, issue the CANCEL TYPE/OVERRIDE command.

If you want the debugger to interpret an entity in one of the above types, but only for the duration of the command that mentions the entity, specify the desired type as a command qualifier. A type specified as a command qualifier is a command override type. For example, the following command directs the debugger to interpret the entity RADIUS as a byte integer:

```
DBG>EXAMINE/BYTE RADIUS
```

As a result of this command, any associated compiler-generated type or current override type established by the SET TYPE/OVERRIDE command is overridden for the duration of the command.

To sum up, you can control the type used by the debugger to interpret an entity in three ways. In increasing order of power, these are

- Setting the default type by the SET TYPE command
- Setting an override type by the SET TYPE/OVERRIDE command
- Specifying a command override type by using a type command qualifier

Thus, a command override type overrides any compiler-generated type, default type, or override type; an override type overrides any compiler-generated type or default type; a compiler-generated type overrides the default type; and the default type, being the weakest type, overrides no other type.

See Chapter 6 for examples of how these types are used in debugging.

5.1.1 The Type Associated with Address Expressions

The type associated with an address expression depends on the current default type, any override types currently in effect, and on the address expression itself.

The debugger associates a type with an address expression in the following way:

1. If an address expression is found in a command that contains a type command qualifier, then the type specified by the qualifier is associated with the address expression.
2. If an address expression is found in a command that does not contain a type command qualifier, then any override type established by the SET TYPE/OVERRIDE command is associated with the address expression.
3. If a type is not associated with the address expression in one of the above ways, then the type associated with the address expression depends on the address expression itself.
   - An address expression that consists of a single, symbolic reference has the type associated with that reference by the source language (the compiler-generated type).
   - The debugger symbols for current entity, logical predecessor, and logical successor have the type associated with the address expression for which they are an abbreviation.
4. If a type is not associated with the address expression in one of the above ways, then the address expression is given the default type.

5.2 Simple Addresses

You can make references to program locations using symbolic notation present in your program, as well as other notation generated by the language compiler and notation peculiar to the debugger itself. Taken as a whole, these forms of notation are called simple addresses.

This section presents each form of simple address and explains how the debugger interprets it.

5.2.1 Symbolic References

Symbolic references are program location references that use program symbols, with or without path-name prefixes.

A symbolic reference may be a source-language symbol or a symbol created with the debugger DEFINE command.

In the following examples, the symbolic references DAT1 and ROUTINE\BLOCK\DAT1 are simple addresses in the debugger EXAMINE command:

```
DBG>EXAMINE DAT1

DBG>EXAMINE ROUTINE\BLOCK\DAT1
```

Both commands direct the debugger to display the value of DAT1 in its compiler-generated type.

5.2.2 Line Numbers

Some language compilers generate line numbers so that references to program locations can be found more easily.

In a debugging session, you can use these line numbers to refer to lines of code in your program. When the debugger encounters the %LINE symbol, it interprets the number that follows as a compiler-generated line number and uses it as a simple address to reference that location.

The following example shows how the %LINE symbol, together with a line number, is used as a simple address:

```
DBG>SET BREAK %LINE 232
```

If you want to use line numbers as simple addresses, you might find that having a compiler listing is helpful because it lists, among other things, each line of code with its corresponding line number. For more information on obtaining a compiler listing, refer to the user's guide for the language you are using.

You may also use line numbers with path-name qualification as simple addresses. The path name precedes the entire line number notation, as follows:

```
MODULE\%LINE 30
```

The %LINE symbol can be used to identify anonymous blocks. See Section 4.5.1 for an example of the %LINE symbol used in this way and for more information about the %LINE symbol.

5.2.3 Statement Numbers

In those languages that allow more than one statement on a line, statement numbers are used to differentiate among statements on the same line.

A statement number consists of a line number, followed by a period and a number indicating the statement. The format follows:

```
%LINE xxx.yy
```

In this format, xxx is the line number and yy is a number specifying the statement. The number one (1) represents the first statement that begins on the line; the number two (2), the second; and so on.

For example, in VAX BASIC, the second statement on line 500 is expressed as follows:

```
%LINE 500.2
```

You may also use statement numbers with path-name qualification as simple addresses. The path name precedes the entire statement number, as follows:

```
MODULE\%LINE 30.3
```

5.2.4 Numeric Labels

In some languages, numeric labels are used to label lines of source code in a program. You can use these labels to reference lines of code in a debugging session by prefixing them with the %LABEL symbol.

The format of a reference to a numeric label in the debugger syntax is

```
%LABEL xxx
```

Here, xxx is a user-defined numeric label.

For example, in VAX FORTRAN, a numeric label is a number you place in a designated column of your program at an important program location so that you can reference that location. In debugging your program, you must prefix a reference to a numeric label with the %LABEL symbol.

Example 5–11 shows part of a compiler listing of a VAX FORTRAN program. The numbers in the leftmost column are line numbers; you refer to them in a debugging session using the %LINE prefix. The numbers in the next column to the right are numeric labels; you refer to them in a debugging session using the %LABEL prefix.

The following debugger command halts program execution at numeric label 20 in the VAX FORTRAN program shown in Example 5–11:

```
DBG>SET BREAK %LABEL 20
```

Note that program execution is stopped at the same program location if the following command is issued:

```
DBG>SET BREAK %LINE 12
```

You can use a numeric label with path-name qualification as a simple address. The path name precedes the entire line numeric label, as follows:

```
CIRCLE\%LABEL 10
```

Example 5–11 Line Numbers and Numeric Labels

```
        C     PROGRAM TO FIND THE AREA
        C     OF A CIRCLE

0001            PROGRAM CIRCLE
0002    1       TYPE 5
0003    5       FORMAT (' ENTER RADIUS VALUE ')
0004            ACCEPT 10,RADIUS
0005    10      FORMAT (F6.2)
0006                  IF (RADIUS .LE.0) GO TO 20
0007            PI = 3.1415927
0008            AREA = PI*RADIUS**2
0009            TYPE 15,AREA
0010    15      FORMAT (' AREA OF CIRCLE EQUALS ',F10.3)
0011                  GO TO 30
0012    20            TYPE 25
0013    25            FORMAT (' PLEASE TYPE A POSITIVE NUMBER')
0014                  GO TO 1
0015    30            STOP
0016            END
```

5.2.5 Numeric Literals

A *numeric literal* is any character string in the source language that is a constant but not a symbol. The debugger can interpret numeric literals in a variety of ways.

If you use a numeric literal in a simple address, the debugger interprets it as a virtual memory address. On the other hand, a literal in an address expression is treated as an offset from a program location. Positive and negative numbers indicate offset direction.

In the following example, the debugger interprets the numeric literal 5032 as a virtual memory address:

```
DBG>EXAMINE 5032
```

In the example below, the literal 4 is treated as a 4-byte offset from the memory address represented by OPT. This debugger command deposits the value of X, which is interpreted as a byte integer, in the program location specified by the address expression OPT+4.

```
DBG>DEPOSIT/BYTE  OPT+4 = X
```

In the evaluation of an address expression containing a literal of a type other than integer (such as floating point, bit string, or complex), the debugger attempts to convert that literal into integer form in the semantics of the source language. If the source language supports such conversion, the debugger uses the resulting integer value. However, if the conversion is not supported, the debugger issues an error message.

In addition, you can use radix operators with any legal, source-language numeric literal. They cause the debugger to interpret your input in the radix you have specified. Legal radix operators are listed below.

%BIN	Causes the debugger to interpret the number in binary radix
%DEC	Causes the debugger to interpret the number in decimal radix
%HEX	Causes the debugger to interpret the number in hexadecimal radix
%OCT	Causes the debugger to interpret the number in octal radix

5.2.6 Current Entity Symbol

The two current entity symbols are the period and the debugger permanent symbol **%CURLOC**. You may use them to make a reference to the program location last referenced by an EXAMINE or DEPOSIT command.

In other words, whenever either of the commands below is executed, the value of the current entity is set equal to the value of the address-expression in that command.

- EXAMINE address-expression
- DEPOSIT address-expression = value

Further, the debugger interprets the value of the current entity in the type associated with the address-expression in the EXAMINE or DEPOSIT command that set the current entity.

In Example 5–12, EXAMINE RADIUS sets the current entity. The command "EXAMINE ." results in a display of the name of the current entity (CIRCLE\RADIUS) and the value of the current entity in the floating point type.

Example 5–12 Examining the Current Entity

```
DBG>EXAMINE RADIUS                !Sets current entity
CIRCLE\RADIUS:  0.0000000E+00      !symbol.

DBG>EXAMINE .                     !Current entity is
CIRCLE\RADIUS:  0.0000000E+00      !RADIUS.  Display type
                                   !used is the type
                                   !of RADIUS.

DBG>DEPOSIT PI = 3.141593         !Sets current entity.
                                   !symbol.

DBG>EXAMINE %CURLOC
CIRCLE\PI:       3.141593          !Current entity is PI.
```

5.2.7 Logical Predecessor Symbols

The two logical predecessor symbols are the circumflex (^) and the debugger permanent symbol **%PREVLOC**. When the current entity symbol refers to an entity in an aggregate such as an array, you may use either of the logical predecessor symbols to refer to that entity in the aggregate that is logically prior to the current entity.

The logical predecessor of an entity may not be its physical predecessor; that is, the logical predecessor may not occupy the region of physical storage directly preceding that of the current entity. Such is the case, for example, when the current entity refers to the cell of a disconnected array A(2) whose logical predecessor A(1) is not stored physically adjacent to A(2) in memory. On the other hand, the logical predecessor of an entity may also be its physical predecessor. For instance, in a connected array, A(1) is both the logical and physical predecessor of A(2).

In some cases the current entity may not have a logical predecessor. For instance, the first cell of an array A(1) has no logical predecessor, and hence its physical predecessor is logically unrelated to it.

As with the current entity, the debugger uses type and symbolic information associated with the logical predecessor.

Example 5–13 demonstrates how to use the logical predecessor symbol to examine the cells of an array.

Example 5–13 Using the Logical Predecessor Symbol

```
DBG>DEPOSIT CHAR(1) = '1234567890'   !Sets current entity
                                      !symbol.

DBG>DEPOSIT CHAR(2) = 'ABCDEFGHIJ'   !Sets current entity
                                      !symbol.

DBG>DEPOSIT CHAR(3) = 'abcdefghij'   !Sets current entity
                                      !symbol.

DBG>EXAMINE .                        !Examines current entity
MOD\CHAR(3):   abcdefghij             !which is CHAR(3).

DBG>EXAMINE ^                        !Logical predecessor is
MOD\CHAR(2):   ABCDEFGHIJ             !previous array member
                                      !CHAR(2).

DBG>EXAMINE %PREVLOC                 !Logical predecessor is
MOD\CHAR(1):   1234567890             !previous array member
                                      !CHAR(1).
```

5.2.8 Logical Successor Symbols

The logical successor symbols are the RETURN key and the debugger permanent symbol **%NEXTLOC**. You may use them to make a reference to the program location that logically follows the current entity. To examine the logical successor of the current entity with the RETURN key, for instance, you enter the EXAMINE command without an operand and then press RETURN.

The logical successor of an entity may not be its physical successor; that is, the logical successor may not occupy the region of physical storage directly following that of the current entity. Such is the case, for example, when the current entity refers to the cell of a disconnected array A(2) whose logical successor A(3) is not stored physically adjacent to A(2) in memory. On the other hand, the logical successor of an entity may also be its physical successor. For instance, in a connected array, A(3) is both the logical and physical successor of A(2).

In some cases the current entity may not have a logical successor. For instance, the last cell of an array has no logical successor, and hence its physical successor is logically unrelated to it.

As with both the current entity and logical predecessor, the debugger uses type and symbolic information associated with the logical successor.

Example 5–14 demonstrates how the logical successor symbol may be used to examine succeeding cells in an array.

Example 5–14 Using the Logical Successor Symbol

```
DBG>DEPOSIT CHAR(1) = '1234567890'

DBG>DEPOSIT CHAR(2) = 'ABCDEFGHIJ'

DBG>DEPOSIT CHAR(3) = 'abcdefghij'

DBG>EXAMINE CHAR(1)              !Sets current entity
MOD\CHAR(1): 1234567890           !symbol.

DBG>EXAMINE RET                 !Logical successor is
MOD\CHAR(2): ABCDEFGHIJ           !CHAR(2).

DBG>EXAMINE %NEXTLOC             !Next logical successor
MOD\CHAR(3): abcdefghij           !is CHAR(3).
```

5.3 Address Expressions

An address expression specifies a (possibly) typed program location. It may consist of a single operand (a simple address) or of many operands together with operators and delimiters.

The debugger evaluates an address expression according to its own rules, which are similar to those used to evaluate an expression in a programming language. Both an expression and an address expression may include operators and delimiters, and both are evaluated according to rules of precedence. The result of the evaluation of an address expression is a 32-bit longword integer that represents a program location.

In an address expression, an operand may be one of the following:

- A simple address
- A unary operator with an operand
- A binary operator with two operands
- An address expression surrounded by parentheses

5.3.1 Operands

Operands in address expressions may be simple addresses, as defined in Section 5.2, or subexpressions, where a subexpression is an expression within an expression. The operands in a subexpression may be simple addresses or other subexpressions.

In the following example, the simple address %LINE 40 is used as an operand in an address expression that also contains the literal 2 as an operand, a multiplication operator, and delimiters.

```
DBG>EXAMINE (%LINE 40)*2
```

This command displays the value at the program location whose address is twice that of %LINE 40.

In the next example, the address expression (X+4)*2 consists of two operands: the subexpression (X+4) and the literal 2.

```
DBG>EVALUATE/ADDRESS (X+4)*2
```

This command calculates the virtual memory address denoted by the address expression (X+4)*2.

In the next example, the address expression ((X–8)*2)+4 contains two subexpressions. The larger subexpression ((X–8)*2) is delimited so as to be one operand for the addition operator, the other operand being the literal 4. The smaller subexpression (X–8) is contained within the larger subexpression and is one of the operands for the multiplication operator, the other operand being the literal 2.

```
DBG>DEPOSIT  ((X-8)*2)+4 = Q
```

This command deposits the value Q in the location represented by the address expression ((X–8)*2)+4.

5.3.2 Operators

An operator with one operand is called a unary operator; an operator with two operands is called a binary operator.

Legal operators are listed below.

Plus sign (+)	Can be either a unary or binary operator. As a unary operator, the plus sign indicates the unchanged value of its operand. As a binary operator, the plus sign adds the preceding operand and succeeding operand together.
Minus sign (–)	Can be either a unary or binary operator. As a unary operator, the minus sign indicates the negation of the value of its operand. As a binary operator, the minus sign subtracts the succeeding operand from the preceding operand.
Multiplication sign (*)	Is a binary operator. It multiplies the preceding operand by the succeeding operand.
Division sign (/)	Is a binary operator. It divides the preceding operand by the succeeding operand.

At sign (@)
Period (.)
	Are both unary operators. The at sign (@) and the period function as "contents" (of) operators. The "contents of" operator causes its operand to be interpreted as a virtual address and thus requests the "contents of" (or value residing at) that address.
	
	For example, assume that the value of pointer variable PTR is 7FF00000 hexadecimal, the virtual address of an entity that you want to examine. Assume further that the value of this entity is 3FF00000 hexadecimal. The following command demonstrates how the "contents of" operator is used to examine the entity:
	
	`DBG> EXAMINE/LONG .PTR`
	`7FF00000: 3FF00000`

Bit field <p,s,e>

Is a unary operator. You can apply bit field selection to an address-expression. To select a bit field, you must supply a bit offset (P), a bit length (S), and a sign extension bit (E), which is optional. For example, to examine the address-expression X_NAME starting at bit 3 with a length of 4 bits and no sign extension, you would issue the command as follows:

```
DBG> EXAMINE X_NAME <3,4,0>
```

5.3.3 Precedence

Rules of precedence determine the sequence in which the operations of an expression are carried out. Although the debugger evaluates expressions according to its own rules of precedence, which are language independent, its rules are identical to those of most programming languages.

The order in which the operations within an address expression are carried out is determined by the following three factors, listed in decreasing order of precedence (first listed have higher precedence):

1. The use of delimiters (usually parentheses or brackets) to group operands with particular operators
2. The assignment of relative priority to each operator
3. Left-to-right priority of operators

The following are the legal debugger operators, listed in decreasing order of precedence.

1. Unary operators ((.), (@), (+), (-))
2. Multiplication and division operators ((*), (/))
3. Addition and subtraction operators ((+), (-))

For example, in the evaluation of the following address expression, the debugger first adds the operands within parentheses, then divides the result by 4, then subtracts the result from 5.

```
5-(T+5)/4
```

5.3.4 The EVALUATE/ADDRESS Command

The debugger interprets the operand of an EVALUATE/ADDRESS command as a language-independent address expression, evaluates the address expression using its own rules for expression evaluation (which are similar to those of most languages), and displays the value of the address expression as a virtual address.

The format of the EVALUATE/ADDRESS *command is*

```
EVALUATE/ADDRESS address-expression [,address-expression...]
```

If you specify a radix mode command qualifier, the debugger displays the value of the address expression as a virtual address in that radix.

You can evaluate more than one address expression in a single EVALUATE /ADDRESS command by separating address expressions with a comma.

Any address expression, as discussed in this section, may be used with the EVALUATE/ADDRESS command.

The EVALUATE/ADDRESS command is useful in determining the virtual addresses of symbols and of expressions that either contain or do not contain symbols. Example 5–15 shows how this command is used.

Example 5–15 Using the EVALUATE/ADDRESS Command

```
DBG>EVALUATE/ADDRESS RADIUS       !For determining the
1024                               !address of a symbol.

DBG>EVALUATE/ADDRESS 1024 + 40    !For performing address
1064                               !arithmetic.

DBG>EVALUATE/ADDRESS (RADIUS + 4)/2
514
```

6 Examining and Depositing Data

This chapter explains how to use the EXAMINE, DEPOSIT, and EVALUATE commands, and how to control the modes and types used by the debugger. The first part of this chapter discusses the modes you can use to control the interpretation of address expressions and language expressions. For information on types, see Section 5.1.

6.1 Modes

The SET MODE command and SET RADIX command let you control how the debugger interprets information you enter in debugger commands, as well as how it displays the results of command execution. A command summary follows. Note that the SET MODE command is also used to control functions that are not related to the interpretation of data or addresses.

D_floating G_floating	The SET MODE [NO]G_FLOATING command specifies that all double-precision constants you enter in expressions should be interpreted as G_floating constants or D_floating constants.
Radix	Radix influences the interpretation of numeric literals and, under certain circumstances, the display of data. Radix may be established by three methods: • A command qualifier specifying the radix • The SET RADIX/OVERRIDE command • The SET RADIX command A SET RADIX/OVERRIDE command overrides the effect of a SET RADIX command, and both commands can be overridden by a radix qualifier on the EXAMINE and EVALUATE commands. Keywords are BINARY, DECIMAL, HEXADECIMAL, and OCTAL. They can be used either as parameters to the SET RADIX/OVERRIDE and SET RADIX commands or as command qualifiers on certain commands.
Symbolic Nonsymbolic	The SET MODE [NO]SYMBOL command influences the amount of symbolization the debugger performs.

To display the current modes, issue the SHOW MODE command.

The following sections discuss radix mode and symbolic mode in more detail.

6.1.1 Radix Modes

You can direct the debugger to interpret and display numbers in any one of four radixes: BINARY, DECIMAL, HEXADECIMAL, and OCTAL. DECIMAL is the default radix for most languages. For BLISS and MACRO, however, the default radix is HEXADECIMAL.

You can set radix in one of the following three ways:

A Command Qualifier

For certain commands, you can override the current radix mode for the duration of a command by using a radix mode command qualifier. The radix mode that you specify overrides all other radix modes. The qualifiers are /BINARY, /DECIMAL, /HEXADECIMAL, and /OCTAL. For example, if the radix mode is set to decimal (the default) but you want to have the value of a program location STATSBLK displayed in binary, you specify the command qualifier as follows:

```
DBG>EXAMINE/BIN STATSBLK
```

This command causes the debugger to display STATSBLK as a binary integer regardless of any other radix information the debugger currently has.

A radix mode command qualifier controls output radix only; it does not control the input radix. You can, however, set the input radix for an expression with four radix operators:

%BIN	Indicates that the debugger should treat the input radix as binary
%DEC	Indicates that the debugger should treat the input radix as decimal
%HEX	Indicates that the debugger should treat the input radix as hexadecimal
%OCT	Indicates that the debugger should treat the input radix as octal

The SET RADIX/OVERRIDE Command

The next most powerful way to establish radix mode is with the SET RADIX /OVERRIDE command. Keywords are BINARY, DECIMAL, HEXADECIMAL, and OCTAL. For example, the command sequence

```
DBG>SET RADIX/OVERRIDE OCT
DBG>EXAMINE STATSBLK
```

displays STATSBLK as an octal integer, overriding any type information the debugger currently has about STATSBLK. To cancel the radix override, issue the CANCEL RADIX/OVERRIDE command.

The SET RADIX Command

The weakest way to establish radix mode is with the SET RADIX command. It causes the debugger to display any integer values (including addresses) using the radix you specified on the SET RADIX command. However, other values (such as floating or enumeration type values) are to be displayed as they normally would. For example, the command sequence

```
DBG>SET RADIX HEX
DBG>EXAMINE 1000
DBG>EXAMINE STATSBLK
```

displays the integer value 1000 as a hexadecimal integer. However, the variable STATSBLK is displayed according to its type.

When the debugger evaluates a source-language expression or an address expression, it interprets numeric literals within that expression in the current input radix mode.

When the debugger evaluates an address expression or a source-language expression, it displays the result in the current output radix mode unless a radix mode command qualifier is specified. In that case, the debugger displays the result in the mode specified by the command qualifier.

Note that when you enter a hexadecimal value that begins with a letter, you must prefix that value with a zero; otherwise the debugger attempts to interpret the entry as a symbol.

The radix mode specified by a command qualifier overrides both the other ways to establish radix mode. Likewise, the radix mode specified with the SET RADIX/OVERRIDE command overrides any radix mode specified with the SET RADIX command.

Although the SET RADIX command has the same effect as a SET MODE command, the SET RADIX command is the preferred way to specify radix information.

To cancel modes established by the SET MODE command, issue the CANCEL MODE command.

After the debugger executes the CANCEL MODE command, default modes are in effect.

Example 6–16 shows how radix modes are used in EXAMINE, EVALUATE, and EVALUATE/ADDRESS commands.

6.1.1.1 Radix Operators

By using a radix operator, you can direct the debugger to interpret a numeric literal in binary (%BIN), decimal (%DEC), octal (%OCT), or hexadecimal (%HEX) radix, provided that the numeric literal is legal for that radix.

Radix operators are useful when you want to enter several numeric literals of different radixes in a single debugger command. By using radix operators, you can specify the radix for each individual numeric literal.

Radix operators are also useful when you want to convert a number in one radix to another radix. For example, if you want to see what a decimal 1000 is in hexadecimal, you can use the command

```
DBG>EV/HEX %DEC 1000
000003E8
```

A radix operator has higher precedence than any other operator. For instance, the expression *%HEX 20 + 33* treats 20 as a hexadecimal integer and 33 as a decimal integer (assuming that decimal is the default radix). However, the expression *%HEX(20 + 33)* treats both 20 and 33 as hexadecimal integers.

You can also nest radix operators. For instance, *%HEX(20 + %OCT 10 + 33)* interprets 20 and 33 hexadecimal radix and 10 in octal radix.

For example, assume you want to deposit a VAX instruction into a memory location denoted by an address expression that is a hexadecimal literal. Further, assume that one operand of the instruction is a decimal literal. You want the debugger to interpret these two numeric literals in their corresponding radixes. In this situation, you cannot use radix command qualifiers since they are not legal qualifiers for the DEPOSIT command. One way to do this operation is to use the %HEX radix operator (to affect the address expression) and a decimal (%DEC) radix operator (to affect the

instruction operand). The following example demonstrates such a situation in VAX MACRO:

```
DBG>DEPOSIT/INSTRUCTION %HEX 5432 = 'MOVL ^O%DEC 222, R1'
```

In this command, the debugger interprets the literal 5432 in hexadecimal radix and the literal 222 in decimal radix.

Example 6–16 Using Radix Mode

```
DBG>SHOW RADIX
input radix: decimal
output radix: decimal
DBG>EVALUATE 10
10                                                    ! "10" is in decimal
                                                      ! for both input and
                                                      ! output

DBG>EVALUATE/HEX 10
0A                                                    ! /HEX affects output only

DBG>SET RADIX/OUTPUT HEX
DBG>SHOW RADIX
input radix: decimal
output radix: hex
DBG>EVALUATE 10
0A                                                    ! Input radix is decimal,
                                                      ! output radix is hex

DBG>SHOW MODE                          !Display default modes.
modes: symbolic, noscreen, nokeypad
input radix : decimal
output radix: decimal

DBG>EXAMINE %LINE 14
TOY$MAIN\%LINE 14: CALLG   W^128(R11),@L^TOY$MAIN+596
                                       !Default type of %LINE
                                       !is instruction.

DBG>E/HEX.                             !Hex representation of
                                                !instruction at
1591:    00 000214FF 0080CBFA          !%LINE 14.  Instruction
                                       !operands are in
                                       !hexadecimal radix.

DBG>E PSL                              !Display PSL in
TOY$MAIN\%PSL:                         !decimal radix.
        CMP TP FPD IS CURMOD PRVMOD IPL DV FU IV T N Z V C
         0   0   0   0  USER   USER    0  0  0  1 0 0 0 0 0

DBG>E/HEX PSL                          !Display PSL in
TOY$MAIN\%PSL:          03C00020   !hexadecimal radix.

DBG>SET RADIX HEX                      !Set default radix mode
                                       !to hexadecimal.

DBG>EXAMINE STATSBLK                   !Examine the array
TOY$MAIN\STATSBLK                      !STATSBLK in hexadecimal
    (1):        00000004               !radix.
    (2):        00000012
    (3):        0000009A
    (4):        00000173
    (5):        00000C93
```

(Continued on next page)

Example 6–16 (Cont.) Using Radix Mode

```
DBG>SHOW RADIX                          !Display the current
input radix : hexadecimal               !default radixes.
output radix: hexadecimal

DBG>EXAMINE/BIN STATSBLK                !Examine STATSBLK
CONV_EX$MAIN\STATSBLK                   !in binary radix.
    (1):        00000000 00000000 00000000 00000100
    (2):        00000000 00000000 00000000 00010010
    (3):        00000000 00000000 00000000 10011010
    (4):        00000000 00000000 00000001 01110011
    (5):        00000000 00000000 00001100 10010011
```

6.1.2 Symbolic and Nonsymbolic Modes

In symbolic mode, which is the default, the debugger attempts to display all addresses symbolically. In nonsymbolic mode, the debugger does not attempt to symbolize addresses. This may increase the speed of command processing. An example follows:

```
DBG>EVALUATE/ADDRESS X
512
DBG>EXAMINE 512
FOO\X:0
DBG>EXAMINE/NOSYMBOL 512
512:0
```

/NOSYMBOL also affects the display of instructions. An example follows:

```
DBG>SHOW MODE                           !Display the new
modes: symbolic, noscreen, nokeypad     !default mode.
input radix : decimal
output radix: decimal

DBG>EXAMINE/INSTRUCTION OUT             Display OUT as an
DRAW\OUT:  CMPB     B^DRAW\COL,#0       !instruction using
                                        !the default symbolic
                                        !mode.  Note that the
                                        !instruction operand
                                        !DRAW\COL is shown
                                        !symbolically.

DBG>EXAMINE/INSTRUCTION/NOSYMBOL OUT    !Specify nonsymbolic
DRAW\OUT:  CMPB     B^3599,#0           !mode.  Note that the
                                        !instruction operand
                                        !DRAW\COL is now
                                        !expressed numeri-
                                        !cally.
```

6.2 The EXAMINE Command

You use the EXAMINE command in the following format to display the value of one or more program entities:

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

If you specify a single address-expression parameter in the EXAMINE command, the debugger displays the value of the entity at the location denoted by the address expression in the type associated with that location.

You can examine more than one entity (a list) in a single EXAMINE command by entering more than one address expression and separating each with a comma.

You can examine a range of entities in a single EXAMINE command by entering the address expression that denotes the first entity in the range, a colon, and the address expression that denotes the last entity in the range. A range is a contiguous sequence of program entities.

You can also examine a list of ranges of entities in a single EXAMINE command by separating each range with a comma.

When you examine the PSL, the debugger displays its contents in a formatted arrangement.

Address expressions may take any of the forms discussed in Chapter 5. See Section 5.1.1 for a complete description of how the debugger associates types with address expressions.

The following sections discuss the use of command qualifiers in the EXAMINE command, the examining of lists, the examining of ranges, and the examining of successive entities.

6.2.1 Command Qualifiers

If you specify a type command qualifier in the EXAMINE command, the entity specified by the address expression is displayed in that type. The type command qualifiers follow:

/ASCIC	/D_FLOAT	/QUADWORD
/ASCID	/DATE_TIME	/OCTAWORD
/ASCII:n	/FLOAT	/PACKED
/ASCIW	/G_FLOAT	/PSL
/ASCIZ	/H_FLOAT	/PSW
/BYTE	/INSTRUCTION	/SOURCE
/CONDITION_VALUE	/LONGWORD	/WORD

You may also specify the radix mode command qualifiers (/BINARY, /DECIMAL, /HEXADECIMAL, and /OCTAL) as well as the symbolic and nonsymbolic mode command qualifiers (/SYMBOL and /NOSYMBOL). See Section 6.1 for further information on the effects of specifying these mode command qualifiers.

You can specify a type qualifier, a radix mode qualifier, and a symbolic /nonsymbolic mode qualifier in a single EXAMINE command.

Example 6–17 demonstrates the use of these mode and type command qualifiers in the EXAMINE command.

Example 6–17 Using Mode and Type Qualifiers with the EXAMINE Command

```
DBG>SET LANGUAGE FORTRAN                    !Set language to FORTRAN.

DBG>SHOW MODE                               !Display the current
modes: nosymbolic, noscreen, nokeypad       !default modes.
input radix : decimal
output radix: decimal

DBG>SHOW TYPE                               !Display default type.
type: long integer

DBG>EXAMINE %LINE 15                        !Display %LINE 15 in default
TOY\%LINE 15 :  MOVL    #1,B^44(R11)                !modes and type.

DBG>EXAMINE/BYTE .                          !Type is byte integer.
TOY\%LINE 15 :  -48

DBG>EXAMINE/WORD .                          !Type is word integer.
TOY\%LINE 15 :  464

DBG>EXAMINE/LONG .                          !Type is long integer.
TOY\%LINE 15 :  749404624

DBG>E/QUAD .                                !Type is quadword integer.
TOY%LINE 15 :  +0130653502894178768

DBG>E/OCTAWORD .                            !Type is octaword integer.
TOY%LINE 15 :      /52446431792802470086860782410464 81

DBG>E/FLOAT .                               !Type is F_floating.
TOY%LINE 15 :    1.9117807E-38

DBG>E/D_FLOAT .                             !Type is D_floating.
TOY%LINE 15 :     1.9117807293306393E-38

DBG>E/G_FLOAT .                             !Type is G_floating.
TOY%LINE 15 :     1.509506018605227E-300

DBG>E/H_FLOAT .                             !Type is H_floating.
TOY%LINE 15 :   2.351187242166315296772081991217048E-4793

DBG>EXAMINE/INSTRUCTION .                   !Type is VAX instruction.
TOY\%LINE 15 :  MOVL    #1,B^44(R11)

DBG>EXAMINE/ASCII .                         !Type is ASCII string.
TOY\%LINE 15 :  ".."

DBG>EXAMINE/OCTAL .                         !Radix mode is octal.
TOY\%LINE 15 :  05452600720                 !Default type is longword.

DBG>EXAMINE/DECIMAL .                       !Radix mode is decimal.
TOY\%LINE 15 : 749404624                    !Default type is longword.

DBG>EXAMINE/BYTE/HEX .                      !Type is byte.  Radix mode is
TOY\%LINE 15 : 0D0                          !hexadecimal.

DBG>EXAMINE/BYTE/OCT .                      !Type is byte.  Radix mode is
TOY\%LINE 15 : 320                          !octal.

DBG>EXAMINE/INSTRUCTION/BYTE .              !When two types are speci-
TOY\%LINE 15 : -48                          !fied, the last overrides the
                                              !first.
```

6.2.2 Examining Instructions

EXAMINE/INSTRUCTION will cause the debugger to display the examined location as an assembly language instruction.

Note that "EXAMINE .PC" will display the instruction you are about to execute. The /INSTRUCTION qualifier is not needed in this case because the debugger knows that ".PC" is of type instruction.

Also note that subsequent EXAMINE commands will display succeeding instructions and subsequent EXAMINE ^ commands will display preceding instructions. The ability to examine backwards in the instruction stream is new with Version 4.0 of VAX/VMS. It is accomplished in a reliable way by backing up to a point we know about, such as the beginning of a line, and decoding forward from there.

You may want to use these features in conjunction with the SET STEP INSTRUCTION command.

6.2.3 Examining Lists

You can examine any number of program locations using a single EXAMINE command by separating parameters in the command string with a comma. The format follows:

```
EXAMINE  address-expression, address-expression, ...
```

The example below shows how you examine a list of three variables with a single EXAMINE command.

```
DBG>EXAMINE I,K,R
TOY\I: 0
TOY\K: 0
TOY\R: 0.0000000E+00
```

6.2.4 Examining Ranges

You can examine a range of successive program locations using a single EXAMINE command by specifying the first and last program locations in the range, in the following format:

```
EXAMINE address-expression1 : address-expression2
```

Address-expression1 must have a smaller virtual memory address than address-expression2. Otherwise, the debugger issues an error message.

As a result of this command, the debugger displays the entity specified by address-expression1, the logical successor of address-expression1, the next logical successor, and so on, until it displays the entity specified by address-expression2.

The debugger associates a type with address-expression1 according to the rules described in Section 5.1.1.

Note that you can use a single EXAMINE command to examine more than one range of program locations (in other words, a list of ranges) by separating ranges with a comma. The following is the format for specifying a list of ranges:

```
EXAMINE address-expression1 : address-expression2, -
        address-expression3 : address-expression4, ...
```

Note that the hyphen, which is the line continuation character, is used to continue the command string on a new line.

Example 6–18 demonstrates the examination of ranges of program locations.

Example 6–18 Examining Ranges of Program Locations

```
DBG>EXAMINE R:I                                  !R has a
%DEBUG-W-EXARANGE, invalid range of addresses    !larger address
                                                 !than I.

DBG>EXAMINE I:R                          !Modes are the defaults
TOY\I:  555                              !symbolic and decimal.
TOY\J:  0                                !Logical successor of I is
TOY\R:  0.0000000E+00                    !J.  Logical successor of J
                                         !is R.  I and J have the
                                         !type longword.  R has the
                                         !floating type.

DBG>EXAMINE/BYTE I:R                     !Byte is the override type.
TOY\I:  43                               !The range of locations is
TOY\I+1:  2                              !displayed as a series of
TOY\I+2:  0                              !bytes.
TOY\I+3:  0
TOY\J:  0
TOY\J+1:  0
TOY\J+2:  0
TOY\J+3:  0
TOY\R:  0

DBG>EXAMINE/BYTE/NOSYMBOL I:R            !Nonsymbolic mode causes
708:  43                                 !the location of each byte
709:  2                                  !to be displayed as a virtual
710:  0                                  !address, not as a
711:  0                                  !symbol.
712:  0
713:  0
714:  0
715:  0
716:  0
```

Example 6–19 Examining Successive Entities

```
DBG>EXAMINE I                    !Default type is longword.
TOY\I:  555

DBG>EXAMINE RET                  !Display logical successor
TOY\J:  0                          !of I.

DBG>EXAMINE %NEXTLOC             !Display the next logical
TOY\R:  0.0000000E+00              !successor of J.

DBG>EXAMINE RET                  !Display the next logical
TOY\K:  0                          !successor of R.

DBG>EXAMINE VECTOR(1)            !Display value of VECTOR(1).
TOY\VECTOR(1):  0.0000000E+00

DBG>E RET                        !Display logical successor.
TOY\VECTOR(2):  0.0000000E+00
```

6.2.5 Examining Successive Entities

You can examine the value of successive entities by using either the logical successor symbol RETURN or the debugger permanent symbol %NEXTLOC. The debugger uses the type and mode associated with the current entity to interpret logical successors.

Example 6–19 demonstrates how to examine successive entities using the logical successor symbols RETURN and %NEXTLOC:

6.2.6 Examining Values in Registers

VAX provides 16 general purpose registers, some of which are used for temporary address and data storage. You can examine the contents of any register by specifying that register in an EXAMINE command, and you can deposit values into any register by specifying that register as the address-expression in a DEPOSIT command. For more information on depositing value in registers, see Section 6.3.5.

The following symbols denote the 16 VAX registers.

- The letter **%R** followed by a numeral from 0 through 11 represents the corresponding VAX general purpose registers, such as %R0, %R1, %R2, %R3, %R4, %R5, . . . %R11. In general, these symbols are debugger permanent symbols.
- **%PC** represents the program counter and is a debugger permanent symbol.
- **%SP** represents the stack pointer and is a debugger permanent symbol.
- **%AP** represents the argument pointer and is a debugger permanent symbol.
- **%FP** represents the frame pointer and is a debugger permanent symbol.

You may abbreviate registers by leaving out the percent sign (for example, R0 instead of %R0). However, if you do not use the percent character, the debugger may interpret these symbols as program variables you have defined, not as debugger permanent symbols. The debugger interprets these symbols as debugger permanent symbols only if your program does not contain variables of the same names.

Example 6–20 demonstrates how to examine values in the VAX registers.

Example 6–20 Examining Values in VAX Registers

```
DBG>SHOW MODE                              !Display the current
modes: nosymbolic, noscreen, nokeypad      !default modes.
input radix : decimal
output radix: decimal

DBG>SHOW TYPE                              !Display current type.
type: long integer

DBG>EXAMINE SP                             !Examine value of the stack
SP:  2147278720                            !pointer.

DBG>DEPOSIT .SP = 33                       !Deposit 33 into .SP.
DBG>EXAMINE .                              !Check the value of .SP.
214727870:  33

DBG>EXAMINE R11                            !Examine contents of R11.
R11:  1024

DBG>DEPOSIT R11 = 444                      !Deposit new value into R11.

DBG>EXAMINE R11                            !Check the value of R11.
R11:  444
```

6.2.6.1 The Processor Status Longword

The PSL is a 32-bit VAX register whose value represents a number of processor state variables. The first 16 bits of the PSL (referred to separately as the processor status word, or PSW) contains unprivileged information about the current processor state; the values of these bits may be controlled by a user program. The latter 16 bits of the PSL, bits 16 through 31, contain privileged information and should not be altered by the user process.

To examine the contents of the PSL, issue the command

```
DBG>EXAMINE PSL
```

The debugger displays the 32-bit PSL in the following format:

```
PSL:  CMP TP FPD IS CURMOD PRVMOD IPL DV FU IV T N Z V C
       n  n  n  n  mode   mode   1v  n  n  n n n n n n
```

In this display, "n" may be 0 or 1; "mode" may be either KERN, EXEC, SUPR, or USER; and "lv," the interrupt priority level, may be a hexadecimal number from 0 through 1F.

You can also say EXAMINE/PSL to display any location in PSL format. This is useful for examining saved PSLs on the stack.

For more information about the PSL, refer to Section 6.3.5.1.

Example 6–21 demonstrates how to examine and modify the PSL.

Example 6–21 Examining and Modifying the PSL

```
DBG>SHOW MODE                              !Display the current
modes: nosymbolic, noscreen, nokeypad      !default modes.
input radix : decimal
output radix: decimal

DBG>SHOW TYPE                              !Display current type.
type: long integer

DBG>DEPOSIT/WORD PSL = 0                   !Disable all conditions in
                                           !PSL.  In other words,
                                           !clear bits 0 thru 15.

DBG>EXAMINE PSL
PSL:  CMP TP FPD IS CURMOD PRVMOD IPL DV FU IV TN Z V C
       0  0  0  0  USER   USER   0  0  0  1 0 0 0 0 0

                                           !Display formatted PSL.
                                           !All bits are cleared.
```

6.3 The DEPOSIT Command

You use the DEPOSIT command to deposit values in program locations. In the command format below, the expression designates the value to be deposited, and the address expression designates the program location into which the value is to be deposited.

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

The expression is evaluated in the syntax of the source language and in the default radix mode to yield a value. The address expression is evaluated to yield a program location with an associated type.

When the debugger executes the DEPOSIT command, it converts the value of the expression to the type associated with the address expression and deposits the value at the location designated by the address expression.

You may specify a type command qualifier in the DEPOSIT command to associate a type with the program location denoted by the address expression.

The following subsections discuss the depositing of numeric data, ASCII strings, and VAX instructions, as well as the use of radix and mode qualifiers in the DEPOSIT command.

6.3.1 Depositing ASCII Strings

You can deposit an ASCII string into a program location only if that location is denoted by an address expression with an associated ASCII type.

Delimiters may be either apostrophes or quotation marks. However, the delimiter used cannot appear within the string itself.

In all cases, if the length of the string to be deposited exceeds the length of the program location into which it is deposited, the debugger truncates the string to the length of the program location. On the other hand, if the length of the string is less than the length of the program location, the debugger inserts ASCII blanks in the remaining bytes of memory to the right of the last character in the string.

The following sections discuss the depositing of ASCII strings into program locations denoted by address expressions that have an ASCII type, have a non-ASCII type, and have no type.

Depositing with an ASCII type

To deposit an ASCII string into a program location denoted by an address expression of ASCII type, issue the DEPOSIT command in the following format:

```
DEPOSIT address-expression = "ASCII string"
```

Depositing with a Non-ASCII type

If the program location is denoted by an address expression of a type other than ASCII, you can override that type by using the command SET TYPE /OVERRIDE ASCII:n. This procedure is especially useful when you intend to issue several DEPOSIT commands.

You can also use the /ASCII:n command qualifier with the DEPOSIT command to override any associated type only for the duration of the command. This procedure is especially useful when you are issuing a single DEPOSIT command.

The following command format demonstrates the use of the /ASCII:n command qualifier with the DEPOSIT command:

```
DEPOSIT/ASCII:n address-expression = "ASCII string of length n"
```

Note that the value n used in the command qualifier should normally correspond to the number of ASCII characters to be deposited, that is, to the length of the ASCII string.

Example 6–22 demonstrates how to deposit ASCII strings.

Example 6–22 Depositing ASCII Strings

```
DBG>DEPOSIT I = "GOOD"
%DEBUG-W-INVNUMBER, invalid numeric string 'GOOD'

                                !Debugger is expecting a
                                !number because I has an
                                !associated numeric type.
                                !Override that type.

DBG>DEPOSIT/ASCII I = "GOOD"    !Use ASCII override type.
                                !Debugger deposits the string.

DBG>EXAMINE .
TOY\I:  1146048327              !Display in default type.

DBG>EXAMINE/ASCII .             !Specify ASCII display.
TOY\I:  "GOOD"

DBG>SET TYPE/OVERRIDE ASCII     !Specify ASCII override
                                !type.

DBG>DEPOSIT I = "GOOD"

DBG>EXAMINE I
TOY\I:  "GOOD"                   !Display in ASCII override
                                !type.

DBG>CANCEL TYPE/OVERRIDE        !Cancel ASCII override type.

DBG>EXAMINE I
TOY\I:  1146048327              !Compiler-generated type
                                !overrides the default ASCII
                                !type.
```

6.3.2 Depositing Numeric Data

Example 6–23 demonstrates how you can deposit three integer values into three memory locations. It also shows how to use the byte, word, and longword type command qualifiers on the DEPOSIT command.

6.3.3 Depositing and Replacing VAX Instructions

To deposit VAX instructions, the following command format may be used:

```
DEPOSIT/INSTRUCTION address-expression = "VAX instruction"
```

You must enclose the instruction in either apostrophes or quotation marks. Either delimiter may be used so long as it does not appear within the instruction string. The first and last delimiter must be the same.

You must tell the debugger to interpret the delimited string as an instruction and not as an ASCII string. You do this either with the DEPOSIT/INSTRUCTION command or with the SET TYPE/OVERRIDE INSTRUCTION followed by any number of DEPOSIT commands (to execute the command repeatedly).

Depositing VAX instructions is simple when you are depositing into successive memory locations because you can use the logical successor symbol to locate the address where the next instruction is to be deposited. Example 6–24 demonstrates this technique.

Example 6–23 Depositing Numeric Data

```
DBG>SHOW MODE                          !Display the current
modes: nosymbolic, noscreen, nokeypad  !default modes.
input radix : decimal
output radix: decimal

DBG>DEPOSIT J = 333                    !Deposit data in J, R, K.
DBG>DEPOSIT R = 444
DBG>DEPOSIT K = 555

DBG>EXAMINE .                          !Current entity is location
TOY\K:  555                            !last deposited into.

DBG>EXAMINE J                          !The value 333 is deposited
TOY\J:  333                            !in J.

DBG>EXAMINE RET                        !The value 444 is deposited
TOY\R:    444.0000                     !into the logical successor
                                       !of J, which is R.  The type
                                       !associated with R is used
                                       !in display, not the default
                                       !type.

DBG>EXAMINE RET                        !The value 555 is deposited
TOY\K:  555                            !into the logical successor
                                       !of R, which is K.

DBG>SHOW TYPE                          !Display the default type.
type:  long integer

DBG>EVALU/ADDR .                       !Current location is 724.
724

DBG>DEPOSIT/BYTE . = 1                 !Deposit the value 1 into
                                       !the byte of memory whose
                                       !address is 724.

DBG>E .                                !Because the default type is
724:  1280461057                       !long integer, 4 bytes are
                                       !examined.

DBG>E/BYTE .                           !Examine one byte only.
724:  1

DBG>DEPOSIT/WORD . = 2                 !Deposit the value 2 into
                                       !the first two bytes (word)
                                       !of the current entity.

DBG>E/WORD .                           !Examine a word of the
724:  2                                 !current entity.
DBG>DEPOSIT/LONG 724 = 9999            !Deposit the value 9999 into
                                        !4 bytes (a longword) begin-
                                        !ning at virtual address 724.

DBG>E/LONG 724                         !Examine 4 bytes (longword)
724:  9999                              !beginning at virtual
                                        !address 724.
```

Replacing one or more VAX instructions with new ones involves keeping track of the length of each instruction, which varies depending on the type of instruction and the number of operands.

When you replace an instruction, you must ensure that the new instruction is the same length in bytes as the old instruction.

If the new instruction is longer than the old instruction, you cannot deposit it without overwriting, and thereby destroying, the subsequent instruction.

Example 6–24 Depositing VAX Instructions

```
DBG>SET TYPE INSTRUCTION              !Set default type to
                                       !instruction.

DBG>DEPOSIT 730 = "MOVB #77, R1"      !Deposit the instruction
                                       !beginning at virtual
                                       !address 730.

DBG>EXAMINE .                         !Examine current entity.
730:  MOVB  #77,R1

DBG>EXAMINE RET                      !Make current entity the
734:  HALT                             !logical successor of
                                       !virtual address 730.

DBG>DEPOSIT . = "MOVB #66, R2"        !Deposit next instruction.

DBG>EXAMINE .                         !Examine current entity.
734:  MOVB  #66,R2

DBG>EXAMINE RET                      !Make current entity the
738:  HALT                             !logical successor of
                                       !virtual address 734.

DBG>DEPOSIT . = "MOVB #55, R3"        !Deposit next instruction.

DBG>EXAMINE .                         !Examine current entity.
738:  MOVB  #55,R3
```

If the new instruction occupies fewer bytes of memory than the old instruction, it can be deposited without overwriting previous or subsequent instructions; however, you must deposit NOP instructions (instructions that cause "no operation") in bytes of memory left unoccupied after the replacement.

The debugger does not warn you if an instruction you are depositing will overwrite a subsequent instruction, nor does it remind you to fill in vacant bytes of memory with NOPs. Therefore, careful calculation of instruction length is required in replacing instructions.

Example 6–25 demonstrates the replacing of an instruction with an instruction of equal length.

6.3.4 Depositing in Different Radixes

Literals in source-language expressions are interpreted by the debugger in the current radix mode or in the radix specified by a radix operator. Example 6–26 demonstrates how to deposit literals in binary, decimal, hexadecimal, or octal radix.

6.3.5 Depositing Values in Registers

The rules for referring to registers are the same for both the DEPOSIT command and the EXAMINE command. See Section 6.2.6 for detailed information.

Example 6–27 demonstrates how to examine and deposit values into the VAX registers.

Example 6–25 Replacing VAX Instructions

```
DBG>SET STEP INSTRUCTION                !Set step unit to instruc-
                                         !tion.

DBG>STEP                                !Step by instruction.
stepped to 1584: PUSHAL  (R11)

DBG>STEP                                !Step by instruction.
stepped to 1586: CALLS   #1,L^2224      !Replace this instruction.

DBG>EXAMINE .PC
1586: CALLS  #1.L^2224

DBG>EXAMINE RET
1593:  CALLS   #0,L^2216                !Subsequent instruction
                                        !begins at 1593.

DBG>DEPOSIT/INSTRUCTION 1586 = "CALLS  #2,L^2224"
                                        !Deposit new instruction.

DBG>EXAMINE .
1586:  CALLS   #2,L^2224                !New instruction is
                                        !deposited.

DBG>EXAMINE RET
1593:  CALLS   #0,L^2216                !Subsequent instruction is
                                        !unchanged.
```

Example 6–26 Depositing in Different Radixes

```
DBG>SHOW MODE                           !Display the current
modes: nosymbolic, noscreen, nokeypad    !default modes.
input radix : decimal
output radix: decimal

DBG>SHOW TYPE                           !Display default type.
type: long integer

DBG>EXAMINE J                           !Display J in default
TOY\J:  234567890                        !modes and type.

DBG>DEPOSIT J = %OCT 7777777            !Deposit an octal value.

DBG>EXAMINE .                           !Display in compiler-generated
TOY\J:  2097151                          !type and default radix
                                         !(decimal).

DBG>EXAMINE/OCTAL .                     !Display in compiler-generated
TOY\J:  00007777777                      !type and octal radix.

DBG>DEPOSIT J = %HEX 7777777            !Deposit a hexadecimal
                                         !value.

DBG>EXAMINE .                           !Display in type
TOY\J:  125269879                        !associated with J and in
                                         !default radix (decimal).

DBG>EXAMINE/HEX .                       !Display in hexadecimal.
TOY\J:  07777777
```

6.3.5.1 The Processor Status Longword

When you deposit into the PSL, your purpose is to enable or disable certain processor state conditions. Table 6–2 in Section 6.2.6.1 contains a list of the processor state conditions that you can manipulate. For more information on the PSL, refer to Section 6.2.6.1.

Example 6–27 Examining and Depositing Values in VAX Registers

```
DBG>SHOW MODE                              !Display the current
modes: nosymbolic, noscreen, nokeypad       !default modes.
input radix : decimal
output radix: decimal

DBG>SHOW TYPE                              !Display current type.
type: long integer

DBG>EXAMINE SP                             !Examine value of the stack
SP:  2147278720                             !pointer.

DBG>DEPOSIT SP = 33                        !Deposit 33 into SP.

DBG>EXAMINE .                              !Check the value of SP.
SP:  33

DBG>EXAMINE R11                            !Examine contents of R11.
R11:  1024

DBG>DEPOSIT R11 = 444                      !Deposit new value into R11.

DBG>EXAMINE R11                            !Check the value of R11.
R11:  444
```

Table 6–2 PSL Modification 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

To deposit a value into the PSL, determine which bits you want set; add their corresponding key numbers together; and use the sum as the "expression" in the command below:

```
DBG>DEPOSIT/WORD PSL = %HEX (expression)
```

Note: If you deposit into the high word of the PSL, bits 16 through 31, a reserved operand fault is generated when control is returned to your program by a STEP or GO command.

Example 6–28 demonstrates how to examine and modify the PSL:

Example 6–28 Examining and Modifying the PSL

```
DBG>SHOW MODE                                !Display the current
modes: nosymbolic, noscreen, nokeypad         !default modes.
input radix : decimal
output radix: decimal

DBG>SHOW TYPE                                !Display current type.
type: long integer

DBG>DEPOSIT/WORD PSL = 0                     !Disable all conditions in
                                              !PSL.  In other words,
                                              !clear bits 0 thru 15.

DBG>EXAMINE PSL
PSL:  CMP TP FPD IS CURMOD PRVMOD IPL DV FU IV TN Z V C
       0   0  0   0  USER   USER   0  0  0  1  0 0 0 0 0

                                              !Display formatted PSL.
                                              !All bits are cleared.

DBG>DEPOSIT/WORD PSL = %HEX 80               !Key-number 80 from
                                              !Table 6--2 enables the
                                              !decimal overflow trap;
                                              !this is key "DV" in
                                              !the formatted display.

DBG>EXAMINE PSL
PSL:  CMP TP FPD IS CURMOD PRVMOD IPL DV FU IV TN Z V C
       0   0  0   0  USER   USER   0  1  0  1  0 0 0 0 0

                                              !Verify that the
                                              !DV bit is set.
```

6.4 The EVALUATE Command

The debugger interprets the operand of an EVALUATE command as a source-language expression, evaluates it in the semantics of the source language, and displays its value as a literal in the source language.

The format of the EVALUATE command is

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

See Appendix A for detailed information about supported operators in language expressions for various languages. You can also obtain information by typing "HELP languages."

If you specify a radix mode command qualifier, the debugger interprets any integer literals in the expression (or expressions) in the current radix. However, it displays the value of the expression (or expressions) in radix you specified with the command qualifier.

If you specify a radix operator, the debugger interprets integer literals in the expression (or expressions) in that radix. However, it displays the value of the expression (or expressions) in the current radix.

If an expression contains symbols with different compiler-generated types, the debugger uses the type-conversion rules of the current language to evaluate the expression.

You can evaluate more than one expression in a single EVALUATE command by separating expressions with a comma.

Using the EVALUATE command, you can perform arithmetic calculations that may or may not be related to your program. You may also perform radix conversions by using radix mode command qualifiers or radix operators.

Example 6–29 demonstrates how to use the EVALUATE command (language is set to FORTRAN).

Example 6–29 Using the EVALUATE Command

```
DBG>DEPOSIT R = 5.35E3              !R is of type floating point.

DBG>EVALUATE R                      !Gives the value of R.
   5350.000

DBG>EXAMINE R                       !The EXAMINE command is
TOY\R:  5350.000                     !similar.  Note that symbolic
                                     !address of R is displayed.

DBG>EVALUATE R*50                   !Perform arithmetic
   267500.0                          !calculation.

DBG>EVALUATE (R*50)/(540-40)        !Perform arithmetic
   535.0000                          !calculation.

DBG>DEPOSIT I = 22222               !Deposit value in I.

DBG>EVALUATE I                      !Note that I is of type
   22222                             !integer.

DBG>EVALUATE R/I                    !Expression is evaluated
   0.2407524                         !in floating point.

DBG>EVALUATE 2.5*35                 !Perform an arithmetic
   87.50000                          !calculation.

DBG>EVALUATE/OCTAL 17/2             !Octal qualifier affects
00000000010                          !output, not input.

DBG>EV %OCT 17/2                    !Octal operator affects
00000000007                          !input, not output.

DBG>EVALUATE/HEX 17/2               !Hex qualifier affects
00000008                             !output, not input.

DBG>EV %HEX 17/2                    !Hex operator affects
DBG>11                              !input, not output.

DBG>EVALUATE/HEX %HEX 17/2          !Both input and output
0000000B                             !are interpreted in hex.
```

7 Displaying Source Code

This chapter describes a source-language display feature that makes it possible (in some languages only) to display programming statements in the language in which the program was written. Such programming statements are referred to as source code.

The debugger identifies a line of source code (a source line) by the line number assigned to it by the compiler. The compiler assigns a line number to each line of source code in the program in sequential order from the first to the last line. Line numbers appear on a compiler listing, which the compiler generates when the /LIST command qualifier is specified at compile time.

The display of source lines is independent of program execution; that is, you may display source lines from any region of your program without such display affecting the value of the program counter (PC) or processor status longword (PSL).

You can display source code by specifying any of the following forms of input parameter:

- A compiler-assigned line number
- An address expression that denotes a program location that has a corresponding line number
- A source code string that identifies a source line by its appearance on that line

It is also possible to display source lines along with other debugger display such as that resulting from the execution of a STEP command, a breakpoint hit, or a watchpoint hit.

Section 7.1 describes how the debugger locates source files; it also describes how to direct the debugger to locate source files that have been moved to a different directory since being compiled. Sections 7.2 through 7.5 describe how to display source lines in each of the ways mentioned above. Section 7.6 describes how to establish parameters that govern the display of source lines.

7.1 Location of Source Files

A source file is a file containing statements in a programming language. A compiler processes source files to generate object modules. If the /DEBUG qualifier is specified at compile time, the compiler generates source-line correlation records for inclusion in the debug symbol table (DST).

Source-line correlation records contain the full file specification (that is, device name, directory name, file name, file type, and version number) of each source file that contributes to each object module, and they specify which source record in which source file corresponds to which line number in the object module. Source-line correlation records are passed to the linker and made available to the debugger at run time.

During a debugging session, therefore, the debugger need only refer to its DST to locate, for any object module, the source file containing the source lines that you want to display.

However, if a source file has been moved to another directory since being compiled, the full file specification of the source file as listed in the DST record will no longer be correct, and the debugger will not be able to locate the source file using the DST record. In this case, you must direct the debugger to the current location of the source file by means of the SET SOURCE command.

The availability of long filenames in Version 4.0 of VAX/VMS creates a problem with the "Declare Source File" command in the source correlation DST record. Both the DST record and the command within the record have byte fields to hold the length. However, a file specification can be as long as 252 characters, and together with the 20 bytes of other information in the DST record, this exceeds what can be stored in a byte length. In order to obtain source display, restrict the full file specification of the source file to 231 characters. You can work around this problem by using the SET SOURCE command. For example, if you define a logical name "X" to expand to your long file specification, the "SET SOURCE X" will show the source display.

7.1.1 SET, SHOW, and CANCEL SOURCE Commands

If you have moved a source file to another directory, issue the SET SOURCE command in the following format:

```
SET SOURCE[/MODULE=modname] dirname[,dirname...]
```

If you specify the optional /MODULE=modname command qualifier, the debugger looks in the specified directory or directories only when it is locating the source file(s) for the specified module.

If you issue a SET SOURCE command without the /MODULE=modname command qualifier, the debugger looks in the specified directory or directories when it is locating the source file(s) for any module not mentioned in a previous SET SOURCE/MODULE=modname command.

In sum, the SET SOURCE/MODULE command tells the debugger where to find source files for a particular module, whereas the SET SOURCE command tells the debugger where to find source files for modules that were not mentioned explicitly in SET SOURCE/MODULE commands.

The dirname parameter may consist of one, several, or all fields in a full file specification, though usually it is only a directory name. The following is the format of a full file specification:

```
node::device:[directory]file-name.file-type;version-number
```

When specifying any of these fields, you must include the punctuation for that field, as shown in the above format. For example, to specify the relocation of source files to the directory NEWDIR on the disk NEWDISK, issue the following command:

```
DBG>SET SOURCE NEWDISK:[NEWDIR]
```

The debugger processes the dirname parameter by inserting the specified fields in the file specification of the source file as it appears in the DST. In this way, the debugger creates a new file specification, which it then uses to locate the source file.

When a source file is moved to another directory, the version number of the source file may change. Hence, to locate the correct version of the source file in the event that a version number was not specified in the dirname parameter, the debugger inserts the match-all wildcard character (*) in the version number field of the new file specification. As a result, all versions of the moved source file are searched until the correct version is located. The correct version of the source file is the version that has the same revision date and time, the same file size, the same record format, and the same file organization as the original compile-time source file.

You can specify more than one dirname parameter in a single SET SOURCE command by separating each dirname parameter with a comma. In this case, the debugger constructs a new file specification for each dirname parameter specified and uses this list of file specifications to locate source files for the object module. Since the dirname parameter is most often a directory name, when you specify more than one dirname parameter in a single SET SOURCE command, you establish a source directory search list.

When a source directory search list has been established for a module or modules, the debugger locates the source files for the designated modules by searching the first directory on the list, then the next, and so on, until it either locates the source file or exhausts the list. Using a source directory search list is particularly helpful when relocated source files are in several directories.

The SHOW SOURCE command displays the source directory search lists currently in effect. The format of the SHOW SOURCE command is SHOW SOURCE.

If a source directory search list has been established for all modules, the SHOW SOURCE command indicates the name(s) of each directory specified and indicates that the list applies to all modules.

If a source directory search list has been established for one or more modules, the SHOW SOURCE command displays the name(s) of each directory specified and displays the name(s) of the module(s) to which the directory search list applies.

If no source directory search list has been established, the SHOW SOURCE command indicates that no such list is currently in effect.

The CANCEL SOURCE command cancels the effects of previous SET SOURCE commands. The format of the CANCEL SOURCE command is

```
CANCEL SOURCE [/MODULE=modname]
```

The CANCEL SOURCE command, without the /MODULE=modname command qualifier, cancels the effect of a previous SET SOURCE command, but does not cancel the effect of any previous SET SOURCE /MODULE=modname command.

The CANCEL SOURCE/MODULE=modname command cancels the effect of a previous SET SOURCE/MODULE=modname command in which the same module name was specified; it does not cancel the effect of a previous SET SOURCE command or of a SET SOURCE/MODULE=modname command in which a different module name was specified.

In all cases, when a source directory search list has been canceled, the debugger again expects the source files corresponding to the designated modules to be in the same directories they were in at compile time.

Assume that you write a VAX COBOL program with the file name TEST.COB;8 and that this program contains two modules MODA and MODB. Assume further that your default disk is DB4 and that, when you compile this program, your default is set to [ME.COBPROG], a subdirectory in which you always keep COBOL programs. At compile time, then, your source file has the full file specification

```
DB4:[ME.COBPROG]TEST.COB;8
```

Assume that you link your program with a VAX FORTRAN program, consisting of a single module CHECK, and that the full file specification of this program at compile time is

```
DB4:[ME.FORPROG]CHECK.FOR;2
```

When you run the program and want to display source lines, the debugger looks for the source lines of modules MODA and MODB in DB4:[ME.COBPROG]TEST.COB;8 and for the source lines of module CHECK in DB4:[ME.FORPROG]CHECK.FOR;2.

Assume that you move TEST.COB to a new subdirectory [ME.TEST]. Now when you run the program and request the debugger to display source lines from MODA, the debugger looks in DB4:[ME.COBPROG]TEST.COB;8 but does not find the source file.

To direct the debugger to the correct directories, you can establish a source directory search list by issuing the command

```
DBG>SET SOURCE [ME.TEST],[ME.FORPROG]
```

The debugger processes this command by substituting the directory name [ME.TEST] in the compile-time file specifications of the source files corresponding to MODA, MODB, and CHECK. As a result, the debugger would successfully locate the source files for modules MODA and MODB (because they were moved to this directory) but would not locate the source file for module CHECK (because it was not moved from its compile-time directory [ME.FORPROG]). However, after failing to find the source file for module CHECK in the directory [ME.TEST], the debugger would substitute the next directory in the source directory search list ([ME.FORPROG]) and would then successfully locate the source file there.

Another way to direct the debugger to the correct directories is to issue the following two commands:

```
DBG>SET SOURCE/MODULE=MODA [ME.TEST]
DBG>SET SOURCE/MODULE=MODB [ME.TEST]
```

The debugger processes these commands by inserting the directory name [ME.TEST] in the compile-time file specifications of the source files corresponding to MODA and MODB. Now whenever you request a display of source lines from MODA or MODB, the debugger locates the source file in DB4:[ME.TEST]TEST.COB.

In this example, it is necessary to issue two SET SOURCE /MODULE=modname commands, rather than a single SET SOURCE command, because the program contains another module CHECK whose corresponding source file was not moved to the directory [ME.TEST]. As a result, the debugger continues to look in the original compile-time directory DB4:[ME.FORPROG]CHECK.FOR;2 when displaying source lines for module CHECK.

7.2 Display by Line Number

The TYPE command allows you to display one or more source lines by specifying one or more compiler-assigned line numbers, where each line number designates a line of source code.

The following is the format of the TYPE command:

```
TYPE [ [modname\][line-number[:line-number] -
     [,[modname\]line-number[:line-number]...] ]
```

If you specify a single line number, the debugger displays the source line corresponding to that line number.

If you specify a list of line numbers, separating each with a comma, the debugger displays the source line corresponding to each of the line numbers.

If you specify a range of line numbers, separating the starting and ending line numbers in the range with a colon, the debugger displays the source lines corresponding to that range of line numbers.

You can read through all the source language statements in your program by specifying a range of line numbers starting from 1 and ending at a number equal to or greater than the largest line number in the program listing.

After displaying a source line, you can display the next line by issuing a TYPE command without a line number, that is, by issuing a TYPE command and then pressing the RETURN key. You can then display the next line and successive lines by repeating this sequence, in this way reading through your source program one line at a time.

You can specify a module name with the line numbers to indicate that the lines are located in that module. In this case, you enter the module name, a backslash ( ), and the line numbers, without intervening spaces.

If you do not specify a module name with the line numbers, the debugger uses the current scope setting to determine which module to use. In this case, the current scope is either the first module designated in a SET SCOPE command or, if a SET SCOPE command was not issued, the module containing the current PC.

Example 7–30 demonstrates how to use the TYPE command.

7.3 Display by Address Expression

By specifying the /SOURCE qualifier in the EXAMINE command, you can display the source line(s) corresponding to the location(s) designated by one or more address expressions.

The debugger evaluates each address expression to derive a virtual address, determines which compiler-assigned line number corresponds to each virtual address, and then displays the source line(s) designated by the line number(s).

The format of the EXAMINE command is

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

The format of the EXAMINE command allows you to specify

- A single address-expression parameter

Example 7–30 Using the TYPE Command

```
DBG>TYPE 160                          !Display the source line
module COBOLTEST                       !designated by line
   160: START-IT-PARA.                 !number 160.

DBG>TYPE RET                         !Display the source line
module COBOLTEST                       !following the last
   161:          MOVE SC1 TO ES0.      !displayed source line.

DBG>T 160:163                         !Display the range of source
module COBOLTEST                       !lines corresponding to the
   160: START-IT-PARA.                 !range of specified line
   161:          MOVE SC1 TO ES0.      !numbers.
   162:          DISPLAY ES0.
   163:          MOVE SC1 TO ES1.

DBG>TYPE COBOLTEST\160,22:24          !Display a single line
module COBOLTEST                       !and a range of lines of
   160: START-IT-PARA.                 !source code in a specified
module COBOLTEST                       !module.
    22: 02       SC2V2    PIC S99V99       COMP VALUE  22.33.
    23: 02       SC2V2N   PIC S99V99       COMP VALUE -22.33.
    24: 02       CPP2     PIC PP99         COMP VALUE  0.0012.
```

- A list of address-expression parameters
- A range of address-expression parameters
- A list of ranges of address-expression parameters

In each case, when you specify the /SOURCE qualifier, the debugger evaluates any specified address expression to determine its corresponding source line.

If you specify a single address expression, the debugger evaluates the address expression to derive a virtual address, determines what line number corresponds to that virtual address, and then (as in the TYPE command) displays the source line designated by that line number.

If you specify more than one address expression (that is, a list), with a comma separating each address expression, the debugger evaluates each address expression as described above and displays, for each specified address expression, a corresponding source line.

If you specify two address expressions, separated by a colon, the debugger evaluates each address expression to derive a range of virtual addresses, determines which line numbers correspond to the range of addresses, and then displays the source lines designated by the line numbers. In this case, both addresses in the range must correspond to line numbers in the same module, and the first of the two line numbers must be less than or equal to the second.

Example 7–31 demonstrates how to use the EXAMINE/SOURCE command.

Example 7–31 Using the EXAMINE/SOURCE Command

```
DBG>EX/SOURCE %PC
!%DEBUG-W-NOSRCLIN, no source line for address 7FFF005C
                                    !The address expression %PC
                                    !designates the address at
                                    !which the contents of the PC
                                    !is stored.  There is no
                                    !source line corresponding to
                                    !this address.  To get the
                                    !desired result, you must
                                    !specify the "contents of"
                                    !operator, the period (.),
                                    !as shown in the following
                                    !command.

DBG>EX/SOURCE .%PC                  !This command displays the
module COBOLTEST                     !source line corresponding
   162:          DISPLAY ES0.        !to the line containing
                                     !the PC.  In effect, it
                                     !displays the source line
                                     !currently being executed.

DBG>EX/SOU 2150                     !This command displays
module COBOLTEST                     !the source line corresponding
   165:          MOVE SC1 TO ES2.    !to the virtual address 2150.

DBG>EX/SOURCE 2150:2200             !This command displays the
module COBOLTEST                     !range of lines of source
   165:          MOVE SC1 TO ES2.    !code that correspond
   166:          DISPLAY ES2.        !to the range of specified
   167:          MOVE SC2V2 TO ES0.  !virtual addresses.
   168:          DISPLAY ES0.
   169:          MOVE SC2V2 TO ES1.

DBG>EX/SOURCE 2100:2120,2150:2175 !This command displays the
module COBOLTEST                     !list of ranges of source
   161:          MOVE SC1 TO ES0.    !lines corresponding to
   162:          DISPLAY ES0.        !the list of ranges of
   163:          MOVE SC1 TO ES1.    !specified virtual addresses.
module COBOLTEST
   165:          MOVE SC1 TO ES2.
   166:          DISPLAY ES2.
   167:          MOVE SC2V2 TO ES0.
```

7.4 Display During Program Execution

When you specify either the SET STEP SOURCE command (the default) or the STEP/SOURCE command, the debugger displays source lines when any one of the following events occur:

- You issue a STEP command and do not specify the /NOSOURCE qualifier. In this case, the debugger displays the source line following the last line or instruction executed.
- A breakpoint or tracepoint is activated; in this case, the debugger displays the source line at the location of the eventpoint.
- A watchpoint is activated; in this case, the debugger displays the source line corresponding to the instruction that caused the watchpoint activation.

When you have given the SET STEP NOSOURCE command, the debugger does not display source lines when STEP commands are executed or when eventpoints are activated.

When you specify the /SOURCE or /NOSOURCE qualifiers on the STEP command, they override, for the duration of that STEP command, any specified or default source parameter currently in effect. However, as command qualifiers, /SOURCE and /NOSOURCE do not affect whether or not the debugger displays source lines when an eventpoint is being evaluated; they only affect whether or not the debugger displays source lines when STEP commands are executed.

Thus, when the SOURCE parameter is in effect, specifying the /NOSOURCE command qualifier on the STEP command suppresses the display of source line(s) for the duration of that STEP command. The debugger continues to display source lines when eventpoints are activated or when subsequent STEP commands are executed (unless the /NOSOURCE command qualifier is again specified).

Also, when the NOSOURCE parameter is in effect (whether by default or by an explicit SET STEP NOSOURCE command), specifying the /SOURCE command qualifier on the STEP command causes the debugger to display source line(s) for the duration of that STEP command. The debugger does not display source lines when eventpoints are activated or when subsequent STEP commands are executed (unless the /SOURCE command qualifier is again specified).

However, you can specify the /SOURCE or /NOSOURCE command qualifiers on the SET BREAK, SET TRACE, and SET WATCH commands. These qualifiers override the current step setting while the eventpoint is evaluated; they do not permanently change the step setting.

Example 7–32 demonstrates how to display lines of source code during program execution.

Example 7–32 Displaying Source Code During Program Execution

```
DBG>STEP                                !Execute a line of code.
stepped to COBOLTEST\START-IT           !Since the default SOURCE
   161:          MOVE SC1 TO ESO.       !parameter is in effect, the
                                        !debugger displays the source
                                        !line.

DBG>SET BREAK %LINE 163                 !Set a breakpoint at
                                        !line number 163.

DBG>GO
break at COBOLTEST\START-IT\START-IT-PARA\%LINE 163
   163:          MOVE SC1 TO ES1.
                                        !Begin execution at the
                                        !current location.  Source
                                        !line display occurs when
                                        !the breakpoint is reached.

DBG>SET BREAK/NOSOURCE %LINE 164        !Set a breakpoint at
                                        !line number 164.  Specify
                                        !that the source lines are
                                        !not to be displayed.

DBG>GO
break at COBOLTEST\START-IT\START-IT-PARA\%LINE 164
                                        !Begin execution at the
                                        !current location.  The
                                        !source code is not dis-
                                        !played although SOURCE
                                        !is the current step setting.

DBG>SET BREAK %LINE 165                 !Set a breakpoint at
                                        !line number 165.

DBG>SET STEP NOSOURCE                   !Specify that source lines
                                        !not be displayed when STEP
                                        !commands are executed or
                                        !when breakpoints or
                                        !watchpoints are activated.

DBG>GO
break at COBOLTEST\START-IT\START-IT-PARA\%LINE 167
                                        !Begin execution at the
                                        !current location.  Note that
                                        !a source line is not
                                        !displayed at breakpoint
                                        !activation because NOSOURCE
                                        !is in effect.

DBG>STEP/SOURCE                         !Execute a line of code and
stepped to COBOLTEST\%LINE 166          !display the source line
   166:          DISPLAY ES2.           !following the line executed.
                                        !The NOSOURCE parameter is
                                        !overriden for the duration
                                        !of the STEP command.

DBG>STEP                                !Execute a line of code.
stepped to COBOLTEST\%LINE 167          !Note that no source code
                                        !is displayed because the
                                        !SET STEP NOSOURCE command
                                        !was issued and the /SOURCE
                                        !qualifier was not specified
                                        !in the STEP command.
```

7.5 Display by Search String

The SEARCH command directs the debugger to search the source code for a specified string and to display the source line or lines containing an occurrence of the string.

The format of the SEARCH command is

```
SEARCH [/qualifier,...] range string
```

The range parameter limits the debugger's search for occurrences of the string to specified regions of the source code. These regions may be specified in any of the following formats:

MODNAME	Indicates a search of the specified module from line number 0 to the end of the module.
MODNAME\LINE-NUM	Indicates a search of the specified module from the specified line number to the end of the module.
MODNAME\LINE-NUM:LINE-NUM	Indicates a search of the specified module beginning at the line number specified to the left of the colon and ending at the line number specified to the right of the colon.
LINE-NUM	Indicates a search of the module designated by the current scope setting from the specified line number to the end of the module.
LINE-NUM:LINE-NUM	Indicates a search of the module designated by the current scope setting beginning at the line number specified to the left of the colon and ending at the line number to the right of the colon.
NULL	Indicates a search of the same module as that from which a source line was most recently displayed (as a result of a SEARCH, TYPE, or EXAMINE/SOURCE command), beginning at the first line following the line most recently displayed and continuing to the end of the module.

The string parameter specifies the source code characters for which to search. If the string parameter is not specified, the debugger uses the last specified search string, that is, the string parameter specified in the last SEARCH command. If there was no previous search string, an error message results. The string parameter may be enclosed in quotation marks or in apostrophes or may be specified without delimiters.

If the string contains a quotation mark and you want to delimit the string with quotation marks, use double quotation marks to indicate the quotation mark that is part of the string. Likewise, if the string contains an apostrophe and you want to delimit the string with apostrophes, use double apostrophes to indicate the apostrophe that is part of the string.

A delimited string may contain spaces, tabs, or special characters. Specifying an undelimited string is more convenient because it saves keystrokes, but an undelimited string is subject to certain restrictions:

- It must have no leading or trailing blanks or tabs.
- It must have no embedded semicolon (;).
- The range parameter must not be null; that is, an explicit range must be specified.

The /ALL command qualifier directs the debugger to search for all occurrences of the string in the specified range and display every line containing an occurrence of the string.

The /NEXT command qualifier directs the debugger to search for the first occurrence of the string in the specified range and to display only the line containing this occurrence. Subsequent lines in the specified range that contain the search string are not displayed. /NEXT is the default.

The /IDENTIFIER command qualifier directs the debugger to search for an occurrence of the string in the specified range but to display the string only if it is bounded on either side by a character that cannot be part of an identifier in the current language. (An identifier is the name associated with a data or program entity.)

Specifying /IDENTIFIER is useful when you are searching for an identifier but want to eliminate extraneous occurrences of the character(s) composing that identifier. For example, suppose you want to display all occurrences of the identifier X, but your program also contains identifiers XT and EXP. Obviously, you do not want every occurrence of XT and EXP to be displayed just because it contains the character X. So you specify the /IDENTIFIER command qualifier to direct the debugger to disregard occurrences of XT and EXP. The debugger disregards them because the X in both XT and EXP is bounded on at least one side by a character (a letter of the alphabet) that can be part of an identifier in the current language.

The /STRING command qualifier directs the debugger to search for and display the string as specified, and not to interpret the context surrounding an occurrence of the string, as it does in the case of /IDENTIFIER. /STRING is the default.

The SET SEARCH command establishes current SEARCH parameters for the debugger to use whenever a SEARCH command qualifier is not specified in a SEARCH command. The following is the format of the SET SEARCH command:

```
SET SEARCH parameter[,parameter]
```

SEARCH parameters determine whether the debugger searches for all occurrences (ALL) of the string or only the next occurrence (NEXT) of the string, and whether the debugger displays any occurrence of the string (STRING) or only those occurrences in which the string is not bounded on either side by a character that can be part of an identifier in the current language (IDENTIFIER).

If you do not specify SEARCH parameters with the SET SEARCH command, the debugger uses the default values NEXT and STRING.

You can override current SEARCH parameters for the duration of a single SEARCH command by specifying other SEARCH parameters as command qualifiers in the SEARCH command.

You can specify more than one SEARCH parameter in a single SET SEARCH by separating parameters with a comma.

The SHOW SEARCH command displays the current SEARCH parameters.

Example 7–33 demonstrates how to use the SEARCH, SET SEARCH, and SHOW SEARCH commands:

Example 7–33 Using the SEARCH and SET/SHOW SEARCH Commands

```
DBG>SHOW SEARCH
search settings: search for next occurrence, as a string
                                    !Display current SEARCH
                                    !parameters.

DBG>SEARCH/STRING/ALL 40:50 D
module COBOLTEST
    40: 02      D2N     COMP-2 VALUE -234560000000.
    41: 02      D       COMP-2 VALUE  222222.33.
    42: 02      DN      COMP-2 VALUE -222222.333333.
    47: 02      DR0     COMP-2 VALUE  0.1.
    48: 02      DR5     COMP-2 VALUE  0.000001.
    49: 02      DR10    COMP-2 VALUE  0.00000000001.
    50: 02      DR15    COMP-2 VALUE  0.0000000000000001.
                                    !Display all source lines
                                    !in the range of line numbers
                                    !40 to 50 that contain an
                                    !occurrence of the letter D.

DBG>SEARCH/IDENTIFIER/ALL 40:50 D
module COBOLTEST
    41: 02      D       COMP-2 VALUE  222222.33.
                                    !Display a line containing D
                                    !only if D is bounded on
                                    !both sides by a character
                                    !that cannot be part of an
                                    !identifier in the current
                                    !language.

DBG>SEARCH/NEXT 40:50 D
module COBOLTEST
    40: 02      D2N     COMP-2 VALUE -234560000000.
                                    !Display the first occurrence
                                    !of D in the range.

DBG>SEARCH/NEXT
module COBOLTEST
    41: 02      D       COMP-2 VALUE  222222.33.
                                    !Display the first occurrence
                                    !of D (the previous search)
                                    !string in the range begin-
                                    !ning at the line following
                                    !the last line displayed
                                    !(line 40 in the above com-)
                                    !mand and ending at the end
                                    !of the current module.
```

(Continued on next page)

Example 7–33 (Cont.) Using the SEARCH and SET/SHOW SEARCH Commands

```
DBG>SEA 80:90
module COBOLTEST
    80: 02      LS10    PIC S9(10)     LEADING SEPARATE VALUE
     -: 1234567890.
                                    !Display all occurrences of D
                                    !(the previous search string)
                                    !in the range 80:90.  Note
                                    !that the line wraps.

DBG>SEA COBOLTEST\170 'D'
module COBOLTEST
   170:         DISPLAY ES1.
                                    !Display the first occurrence
                                    !of D in the module COBOLTEST
                                    !in the range beginning at
                                    !line 170 and ending at the
                                    !end of the module.

DBG>SEA COBOLTEST\150 ' E'
module COBOLTEST
   161:         MOVE SC1 TO ES0.
                                    !Display the first occurrence
                                    !of the delimited string in
                                    !the module COBOLTEST in the
                                    !range beginning at line 150
                                    !and ending at the end of
                                    !the module.

DBG>SEARCH/NEXT 1 ". "
module COBOLTEST
     2: PROGRAM-ID. COBOLTEST.
                                    !Display the first occurrence
                                    !of the delimited string in
                                    !the module COBOLTEST in the
                                    !range beginning at line 1
                                    !and ending at the end of
                                    !the module.

DBG>SET SEARCH IDENT               !Set the current SEARCH
                                    !parameter to IDENTIFIER.

DBG>SHOW SEARCH
search settings: search for next occurrence, as an identifier
                                    !Display current SEARCH
                                    !parameters

DBG>SET SE AL                      !Set the current SEARCH
                                    !parameter to ALL.

DBG>SH SEA
search settings: search for all occurences, as an identifier
                                    !Display current SEARCH
                                    !parameters.
```

7.6 Source Display Parameters

This section describes parameters to set margins on the display of source code and to limit the number of source files that the debugger may keep open at any one time.

7.6.1 Margin Parameters

The SET MARGINS command specifies the leftmost source-line character position at which to begin display of a source line (the left margin) and/or the rightmost character position at which to end display of a source line.

By default, the debugger displays a source line beginning at character position 1 of the source line. Source-line character position 1 is actually character position 9 on your terminal screen. The first eight character positions on the terminal screen are occupied by the line number and cannot be manipulated by the SET MARGINS command.

Increasing the left margin setting (from its default value of 1) is particularly useful when the source code is deeply indented; by eliminating the display of empty space, more space is available on the terminal line for the display of source code.

Decreasing the right margin setting from its default value of 255 prevents wrapping of long lines by truncating them. Since a wrapped line of source code requires two lines on the terminal, eliminating wrapping allows more lines of source code to be displayed on the terminal screen.

The SET MARGINS command affects only the display of source lines, that is, the display resulting from commands such as TYPE and EXAMINE/SOURCE. The SET MARGINS command does not affect the display resulting from commands (such as EXAMINE, EVALUATE, SHOW MODE, and so on) that do not display source code. If a command displays source code together with other information—as, for example, STEP/SOURCE does—the display of source code is affected by the current margin settings but the other information is not.

The following is the format of the SET MARGINS command:

```
                rm
SET MARGINS     lm:rm
                lm:
                  :rm
```

If you specify a single number, the debugger sets the left margin to 1 and the right margin to the number specified.

If you specify two numbers, separated with a colon, the debugger sets the left margin to the number specified to the left of the colon and the right margin to the number specified to the right of the colon.

If you specify a single number followed by a colon, the debugger sets the left margin to the number specified and leaves the right margin unchanged.

If you specify a colon followed by a single number, the debugger sets the right margin to the number specified and leaves the left margin unchanged.

The SHOW MARGINS command displays the current margin settings for the display of source lines.

Example 7–34 demonstrates how to use the SET MARGINS and SHOW MARGINS commands.

Example 7–34 Using the SET/SHOW MARGINS Commands

```
DBG>SHOW MARGINS
left margin: 1 , right margin: 255
                                    !Display current margin
                                    !settings.

DBG>TYPE 116
module COBOLTEST
   116: 02      RA26    PIC A(26)       JUST RIGHT VALUE
     -: "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
                                    !Display source line 116.
                                    !Note that the line wraps,
                                    !thus requiring two terminal
                                    !lines for its display.

DBG>SET MARGINS 50                 !Set the left margin to 1
                                    !and the right margin to 50.

DBG>SHOW MARGINS
left margin: 1 , right margin: 50
                                    !Display current margin
                                    !settings.

DBG>TY 116
module COBOLTEST
   116: 02      RA26    PIC A(26)       JUST RIGHT VALUE
                                    !With a right margin of 50,
                                    !characters on source line 116
                                    !at positions greater than 50
                                    !are not displayed.  As a
                                    !result, line 116 does not
                                    !wrap.

DBG>SET MARGINS 10:60              !Set left margin to 10 and
                                    !right margin to 60.

DBG>TY 116
module COBOLTEST
   116: A26    PIC A(26)       JUST RIGHT VALUE  "ABCDEFGHI
                                    !Characters on source line 116
                                    !at positions less than 10 or
                                    !greater than 60 are not
                                    !displayed.

DBG>SET MARGINS :100               !Set right margin to 100 and
                                    !leave left margin unchanged.
```

(Continued on next page)

Example 7–34 (Cont.) Using the SET/SHOW MARGINS Commands

```
DBG>SHOW MARGINS
left margin: 10 , right margin: 100
                                    !Display current margin
                                    !settings.

DBG>SET MARGINS 5:                 !Set left margin to 5 and
                                    !leave right margin unchanged.

DBG>SHOW MARGINS
left margin: 5 , right margin: 100
                                    !Display current margin
                                    !settings.
```

7.6.2 Maximum Source Files Parameter

The SET MAX_SOURCE_FILES command specifies the maximum number of source files that the debugger may keep open at any one time.

The format of the SET MAX_SOURCE_FILES command is

```
SET MAX_SOURCE_FILES n
```

The parameter n is a decimal integer whose value designates the maximum number of source files that the debugger may keep open at any one time. The value of n may not exceed 20. The default value is 5.

Opening a source file requires the use of an I/O channel, which is a limited system resource. Both the program and the debugger use I/O channels. To ensure that the debugger does not use all available I/O channels and thus cause the user program to fail (for lack of an available I/O channel), you can issue the SET MAX_SOURCE_FILES command to specify the maximum number of source files (and thus source file I/O channels) that the debugger may use at any one time.

Note that the value of MAX_SOURCE_FILES does not limit the number of source files that the debugger can open; rather, it limits the number that may be kept open at any one time. Thus, if the debugger reaches this limit, it must close a file in order to open another one.

Note too that setting MAX_SOURCE_FILES to a very small number can make the debugger's use of source files inefficient.

The SHOW MAX_SOURCE_FILES command displays the number of source files that the debugger may keep open at any one time.

Example 7–35 demonstrates how to use the SET MAX_SOURCE_FILES and SHOW MAX_SOURCE_FILES commands.

Example 7–35 Using the SET/SHOW MAX_SOURCE_FILES Commands

```
DBG>SHOW MAX_SOURCE_FILES          !Display the number of source
max_source_files: 5                 !files that the debugger may
                                    !keep open at the present
                                    !time.

DBG>SET MAX_SOURCE_FILES 8         !Set the number of source
                                    !files that the debugger may
                                    !keep open at any one time
                                    !to 8.

DBG>SHOW MAX_SOURCE_FILES          !Verify the change.
max_source_files: 8
```

7.7 Differences Between Source and Object Code Due to Optimization

When debugging with source code, you should keep in mind that, in general, there is no precise one-to-one correspondence between your source code and the compiler-generated object code.

Most compilers optimize the object code they produce so that the program will execute faster. One method of optimizing object code is to perform operations in a sequence different from the sequence specified in the source code.

As a result, the source code displayed by the debugger will not correspond exactly to the actual object code being executed. This is important to keep in mind, especially when using the SET STEP SOURCE, STEP/SOURCE, and EXAMINE/SOURCE commands.

To illustrate, the following example depicts a segment of source code from a FORTRAN program as it would appear on a compiler listing. This code segment sets the first ten elements of array A to the value 1/X.

```
line          source code
----          -----------
  5            DO 100 I=1,10
  6            A(I) = 1/X
  7         100 CONTINUE
```

As the compiler processes the source program, it determines that the reciprocal of X need only be computed once, not ten times as the source code specifies, since the value of X never changes in the DO-loop. The compiler thus generates optimized object code equivalent to the following code segment:

```
line          object code equivalent
----          ----------------------
  5            T = 1/X
               DO 100 I=1,10
  6            A(I) = T
  7         100 CONTINUE
```

In the optimized object code, the value of 1/X is computed once, saved in a temporary location, and then assigned to each A(I). The object code now executes faster, but it no longer corresponds exactly to the source code.

In this example, if you step to line 5 by issuing STEP/SOURCE (or SET STEP SOURCE followed by STEP), the debugger displays the source line as it appears in the source file, not the optimized object code equivalent that it is actually executing.

```
stepped to PROG_\%LINE 5
      5:        DO 100 I=1,10
```

At this point, if you issue another STEP command to execute line 5, the debugger executes line 5 of the optimized object code, not line 5 of the displayed source code. Thus, the program computes the reciprocal of X and sets up the DO loop, whereas the source display indicates only that the DO loop is set up.

This discrepancy is not obvious from looking at the displayed source line. Furthermore, if the computation of 1/X were to fail because X is zero, it would appear from inspecting the source display that a division by zero had occurred on a source line that contains no division at all.

This kind of apparent mismatch between source code and object code should be expected from time to time when debugging optimized programs. It can be caused not only by "code motions" out of loops, as in the above example, but by a number of other optimization methods as well. Optimization can cause segments of object code to move long distances from their original source code locations.

If you should encounter a program segment where source and object code do not seem to match, you can inspect the object code itself by using the EXAMINE/INSTRUCTION or STEP/INSTRUCTION commands. Alternatively, you can inspect a compiler-generated machine code listing. Using either of these methods, you should be able to determine what is happening at the object code level and thereby resolve the discrepancy between source line display and program behavior.

In addition, most languages allow you to specify the /NOOPTIMIZE command qualifier at compile time. Specifying this qualifier inhibits compiler optimization, thereby eliminating discrepancies between source code and object code caused by optimization.

8 Screen Mode

Screen mode lets you see more information more conveniently than the default, line-oriented, display mode. In screen mode, you display different types of data in separate areas of the screen. You might, for example, display your source code in the top left half of the screen, the contents of the VAX registers in the top right half, debugger output in the middle, and diagnostic messages at the bottom, near your interactive input.

To enable screen mode, press keypad key PF3 (or type the command SET MODE SCREEN). To return to line-oriented debugging, press GOLD-PF3 (or type the command SET MODE NOSCREEN).

Screen mode output is best displayed on VT100 or VT200 series terminals and MicroVAX workstations. The ability to have many displays on the screen for different purposes is particularly useful with the larger screen of MicroVAX workstations. You can use screen mode with VT52 terminals, but they are less suited to the formatted screen displays since they do not support the scrolling regions used in screen mode.

This chapter covers the following topics:

- Section 8.1 introduces screen mode concepts and terminology used throughout the chapter.
- Section 8.2 describes the predefined displays SRC, OUT, PROMPT, INST, and REG, which are automatically available when you enter screen mode.
- Section 8.3 describes how to scroll, hide, delete, move, and resize a display.
- Section 8.4 explains how to create a new display.
- Section 8.5 explains how to specify a display window.
- Section 8.6 describes the different kinds of displays and how to use them.
- Section 8.7 explains how to direct various types of debugger output to different displays by assigning display attributes.
- Section 8.8 gives an example of a display configuration to illustrate a possible use of screen mode.
- Section 8.9 explains how to save the current state of your screen displays.
- Section 8.10 explains how to change your terminal screen's height and width during a debugging session and describes how display windows behave when you change the height or width.

Many screen mode commands are bound to keypad keys. See Appendix B for key definitions. Also, Appendix C contains screen mode information in summary reference format.

8.1 Concepts and Terminology

A *display* is a group of text lines. The text may be lines from a source file, assembly language instructions, the values contained in registers, your input to the debugger, various types of debugger output, or program input and output.

You view a display through its *window*, which may occupy any rectangular area of the screen. Because a display's window is typically smaller than the display, you can scroll the window up, down, right, and left across the display text to view any part of the display.

The following example of screen mode shows three displays. The name of each display (SRC, OUT, and PROMPT) appears at the top left corner of its window. It serves both as a tag on the display itself and as a name for future reference in commands.

- Display SRC is a source code display (displaying FORTRAN code in this example). SRC's current window is the upper half of the screen. Like other display windows, SRC's window may be changed to accommodate different display layouts. The name of the module whose source code is displayed, FORSQUARE$MAIN, is to the right of the display name.
- Display OUT, located in a window directly below SRC, shows the output of debugger commands.
- Display PROMPT, at the bottom of the screen, shows the debugger prompt and debugger input.

SRC, OUT, and PROMPT are three of the five *predefined displays* that the debugger provides by default when you enter screen mode. You can create additional displays.

```
|- SRC: module FORSQUARE$MAIN -scroll-source-------------------------------------|
|     7: C     ---Square all non-zero elements and store in output array         |
|     8:       K = 0                                                             |
|     9:       DO 10 I = 1, N                                                    |
|    10:       IF(INARR(I) .NE. 0) THEN                                          |
|->  11:           OUTARR(K) = INARR(I)**2                                       |
|    12:       ENDIF                                                             |
|    13:    10 CONTINUE                                                          |
|    14: C                                                                       |
|    15: C     ---Print the squared output values.  Then stop.                   |
|- OUT -output-------------------------------------------------------------------|
|stepped to FORSQUARE$MAIN\%LINE 9                                               |
|     9:       DO 10 I = 1, N                                                    |
|FORSQUARE$MAIN\N:       9                                                       |
|FORSQUARE$MAIN\K:       0                                                       |
|stepped to FORSQUARE$MAIN\%LINE 11                                              |
|                                                                                |
|                                                                                |
|                                                                                |
|- PROMPT -error-program-prompt--------------------------------------------------|
|DBG> EXAM N, K                                                                  |
|DBG> STEP 2                                                                     |
|DBG>                                                                            |
|--------------------------------------------------------------------------------|
```

Every display has a memory buffer, whose size is independent of the window size and may be adjusted. Displays that hold source code or assembly language instructions let you see all of the lines of source code of the associated module or all of the instructions of the associated routine, regardless of the size of the memory buffer. This is because the necessary information is paged into the buffer as needed. For other displays, such as

display OUT, the buffer size defines how much text the display will hold. If you add more text to the display, the oldest text lines are discarded to make room for the new text.

Conceptually, displays are placed on the screen as on a *pasteboard*. The display that is most recently referenced in a command is put on top of the pasteboard, by default. Therefore, depending on the window locations, the displays that you have referenced recently may overlay or hide other displays (as on a pasteboard).

The debugger maintains a *display list*, which is the pasting order of displays. Several keypad key definitions use the display list to cycle through the displays currently on the pasteboard.

Every display belongs to a *display kind*. The display kind determines what type of information the display can capture and/or display; for example, source code, assembly language instructions, debugger output of various types. The display kind also determines how the contents of the display are generated.

In general, there are two ways in which the contents of a display are generated. Some displays are automatically updated. Their definition includes a command list that is executed whenever the debugger gains control from the program. The output of the command list forms the contents of those displays. Display SRC belongs to that category: the source display is automatically updated so that an arrow centered in the window shows the current location of the program counter.

Other displays, for example display OUT, derive their contents from commands you issue interactively. If you create a display of this general category, you must first select it (with the SELECT command) as the target display for one or more types of output before anything can be written to it. This is also known as assigning one or more *attributes* to a display.

The names of any attributes assigned to a display appear to the right of the display name, in lowercase letters. In the preceding example, SRC has the source and scroll attributes (SRC is the *current source display* and the *current scrolling display*), OUT has the output attribute (it is the *current output display*), and so on. Note that, although SRC is automatically updated by its own built-in command, it can also receive the output of certain interactive commands (such as EXAMINE/SOURCE) because it has the source attribute.

The concepts introduced in this section are developed in more detail in the rest of this chapter.

8.2 The Predefined Displays

The debugger provides the following predefined displays by default:

- A source display named "SRC"
- An output display named "OUT"
- A prompt display named "PROMPT"
- An assembly-language instruction display named "INST"
- A register display named "REG"

When you enter screen mode and the language is set to any language except MACRO, the debugger puts SRC in the top half of the screen, PROMPT in the bottom sixth, and OUT between SRC and PROMPT, as illustrated in Section 8.1.

Each of the predefined displays is discussed in the next sections.

8.2.1 The Predefined Source Display SRC

Note: The debugger does not provide source line display for MACRO. If the language is set to MACRO, SRC is marked as *removed* (see Section 8.3.2) from the display pasteboard and is not visible. The INST display is put in its place.

The predefined source display SRC displays the source code of the module being debugged, if that source code is available. The arrow in the leftmost column indicates the current PC location. Each time the debugger gains control from your program, the arrow position is updated, and the source text scrolls as needed so that the display is centered around the PC location.

By default, SRC has the source attribute and, therefore, also shows the output of a TYPE or EXAMINE/SOURCE command (the source text is scrolled as needed to reveal the source line output).

If source lines are not available for the currently executing routine (because, for example, the routine is a run-time library routine), the debugger attempts to display source lines in the caller of that routine (scope 1). If source lines are also not available at that level, the debugger tries scope 2, and so on. When displaying source lines that are not associated with the executing module, the debugger displays a message to that effect:

```
%DEBUG-I-SOURCESCOPE, Source lines not available for .0\%PC.
       Displaying source in a caller of the current routine.
```

The following example illustrates this feature. In the source display, the arrow indicates that the PC is at a call of routine TYPE. TYPE corresponds to a FORTRAN run-time library procedure. No source lines are available for that code, so the debugger displays lines in the caller of that routine. The output of a SHOW CALLS command, shown in the output display, identifies the currently executing module and the call sequence.

```
|- SRC: module TEST -scroll-source-------------------------------------------|
|%DEBUG-I-SOURCESCOPE, Source lines not available for .0\%PC                  |
|        Displaying source in a caller of the current routine                 |
|     3:          CHARACTER*(*) ARRAYX                                        |
|->   4:          TYPE *, ARRAYX                                              |
|     5:          RETURN                                                      |
|     6:          END                                                         |
|                                                                             |
|                                                                             |
|                                                                             |
|- OUT --output--------------------------------------------------------------|
|stepped to SHARE$FORRTL+810                                                  |
| module name     routine name                    line       rel PC    abs PC |
| SHARE$FORRTL    SHARE$FORRTL                               0000032A  00000B2A |
|*TEST            TEST                               4       0000001E  00000436 |
|*A               A                                  3       00000011  00000411 |
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
|- PROMPT -error-program-prompt----------------------------------------------|
|DBG> STEP                                                                    |
|DBG> SHOW CALLS                                                              |
|DBG>                                                                         |
|-----------------------------------------------------------------------------|
```

8.2.2 The Predefined Output Display OUT

By default, the predefined display OUT has the output attribute and therefore displays any debugger output that is not directed to the source display SRC or the instruction display INST. By default, OUT does not display debugger diagnostic messages (these appear in the PROMPT display).

The preceding examples illustrate some typical debugger output in display OUT. You can assign attributes to OUT so it will capture debugger input and diagnostics as well as normal output (see Section 8.7).

8.2.3 The Predefined Prompt Display PROMPT

The predefined display PROMPT is where the debugger prompts for input. The preceding examples show PROMPT in its default location, the bottom sixth of the screen.

By default, PROMPT has the program and error attributes, in addition to the prompt attribute. Therefore, the debugger forces program output to PROMPT and prints diagnostic messages to that display.

PROMPT has different properties and restrictions than other displays. This is to eliminate possible confusion when manipulating that display:

- The debugger always keeps PROMPT on top of the display pasteboard so it cannot be hidden by another display. You cannot hide PROMPT (with the DISPLAY/HIDE command), or remove PROMPT from the pasteboard (with the DISPLAY/REMOVE command), or permanently delete PROMPT (with the CANCEL DISPLAY command).
- PROMPT can have the scroll attribute, so that it can be made the default target display for the MOVE and EXPAND commands. But you cannot scroll PROMPT.

- You can move PROMPT anywhere on the screen, expand it to fill the full screen height, and contract it down to two lines. But PROMPT must always occupy the full width of the screen. Therefore, you cannot move, expand, or contract PROMPT horizontally.

The debugger alerts you if you try to move or expand a display where it will be hidden by PROMPT.

8.2.4 The Predefined Instruction Display INST

The predefined assembly-language instruction display INST displays the instruction stream of the routine being debugged (see the example that follows). The instructions displayed are decoded from the image being debugged. The numbers to the left of the instructions are line numbers. The arrow points to the instruction at your current PC. Each time the debugger gains control from your program, the arrow position is updated and the display is centered around the PC location.

If the language is set to a language other than MACRO, INST is marked as *removed* (see Section 8.3.2) from the display pasteboard and is not visible. You need to use the DISPLAY command (or keypad keys 7 or GOLD-7) to show the INST display in such cases. If the language is set to MACRO, the INST display takes the place of the SRC display and is shown by default.

If you assign the instruction attribute to INST with the command SELECT /INSTRUCTION INST, then the output of an EXAMINE/INSTRUCTION command will be directed to the instruction display.

```
|- INST: routine FORSQUARE$MAIN ------------------------------------------------|
|          : BLEQ    FORSQUARE$MAIN\%LINE 16                                    |
|Line    10: MOVL    B^4(R11),R0                                                |
|          : TSTL    W^-164(R11)[R0]                                            |
|          : BEQL    FORSQUARE$MAIN\%LINE 13                                    |
|->ne    11: MOVL    B^12(R11),R1                                               |
|          : MOVL    B^4(R11),R0                                                |
|          : MULL3   W^-164(R11)[R0],W^-164(R11)[R0],B^-84(R11)[R1]             |
|Line    13: AOBLEQ  B^16(R11),B^4(R11),FORSQUARE$MAIN\%LINE 10                 |
|Line    16: PUSHAL  L^525                                                      |
|- OUT -output------------------------------------------------------------------ |
|stepped to FORSQUARE$MAIN\%LINE 9                                              |
|     9:         DO 10 I = 1, N                                                 |
|FORSQUARE$MAIN\N:       3                                                      |
|FORSQUARE$MAIN\K:       0                                                      |
|stepped to FORSQUARE$MAIN\%LINE 11                                             |
|FORSQUARE$MAIN\I:       1                                                      |
|FORSQUARE$MAIN\K:       0                                                      |
|                                                                               |
|                                                                               |
|- PROMPT -error-program-prompt------------------------------------------------|
|DBG> STEP                                                                      |
|DG> EXAMINE I,K                                                                |
|DBG>                                                                           |
|-------------------------------------------------------------------------------|
```

8.2.5 The Predefined Register Display REG

The predefined register display REG automatically shows the current values of all VAX machine registers (see the example that follows). REG also shows the four condition code bits (C,V, Z, and N) of the processor status longword (PSL), plus the top several values on the stack and on the current argument list. The values in this display are highlighted when they change as you execute the program.

REG is initially marked as *removed* (see Section 8.3.2) from the display pasteboard and is not visible. You need to use the DISPLAY command (or the keypad key sequence GOLD-7) to show the REG display.

```
|- SRC: module FORSQUARE$MAIN -scroll-sou+- REG --------------------------------|
|     3: C       ---Read the input array |R0:00000000  R10:7FFEDDD4  @SP:00000000|
|     4:         OPEN(UNIT=8, FILE='DATAF|R1:00000008  R11:000004A0   +4:08000000|
|     5:         READ(8,*) N, (INARR(I), |R2:00000000  AP :7FF4A1CC   +8:7FF4A1CC|
|     6: C                               |R3:7FF4A194  FP :7FF4A180  +12:7FF4A1B8|
|     7: C       ---Square all non-zero e|R4:00000000  SP :7FF4A180  +16:000196C8|
|->   8:         K = 0                   |R5:00000000  PC :0000064D  +20:7FFE33DC|
|     9:         DO 10 I = 1, N          |R6:7FF49E49  @AP:00000006  +24:000009FF|
|    10:         IF(INARR(I) .NE. 0) THEN|R7:8001E4DD   +4:7FFE6440  +28:00000005|
|    11:                 K = K + 1       |R8:7FFED052   +8:7FF9F4EB  +32:00000600|
|    12:                 OUTARR(K) = INAR|R9:7FFED25A  +12:7FFE640C  +36:00000000|
|    13:         ENDIF                   |N:0    Z:0    V:0    C:0   +40:00000001|
|- OUT -output-------------------------+---------------------------------------|
|stepped to FORSQUARE$MAIN\%LINE 4                                              |
|stepped to FORSQUARE$MAIN\%LINE 5                                              |
|stepped to FORSQUARE$MAIN\%LINE 8                                              |
|FORSQUARE$MAIN\I:       5                                                      |
|FORSQUARE$MAIN\K:       0                                                      |
|FORSQUARE$MAIN\N:       4                                                      |
|                                                                               |
| - PROMPT -error-program-prompt-------------------------------------------------|
|DBG> STEP                                                                      |
|DBG> EXAMINE I,K,N                                                             |
|DBG>                                                                           |
|-------------------------------------------------------------------------------|
```

8.3 Manipulating Existing Displays

This section explains how to

- Use the SELECT and SCROLL commands to scroll a display (Section 8.3.1).
- Use the DISPLAY command to show, hide, or remove a display; the CANCEL DISPLAY command to permanently delete a display; and the SHOW DISPLAY command to identify the displays that currently exist and their order in the display list (Section 8.3.2).
- Use the MOVE command to move a display across the screen (Section 8.3.3).
- Use the EXPAND command to expand or contract a display (Section 8.3.4).

8.3.1 Scrolling a Display

A display usually has more lines of text (and possibly longer lines) than can be seen through its window. The SCROLL command lets you view text that is hidden beyond a window's border. You can scroll through all displays except for the PROMPT display.

The easiest way to scroll displays is with keypad keys, as described later in this section. First, use of the relevant commands will be explained.

You can specify a display explicitly with the SCROLL command. Typically, however, you first use the SELECT/SCROLL command to select the *current scrolling display*. This display then has the scroll attribute and is the default display for the SCROLL command. You can then use the SCROLL command with no parameter to scroll that display up or down by a specified number of lines, or to the right or left by a specified number of columns. The direction and distance scrolled are specified with the command qualifiers (/UP:n, /RIGHT:n, and so on).

In the following example, the SELECT command selects display OUT as the current scrolling display (/SCROLL may be omitted because it is the default qualifier); the SCROLL command then scrolls OUT to reveal text 18 lines down:

```
DBG> SELECT OUT
DBG> SCROLL/DOWN:18
```

Several useful SELECT and SCROLL command lines are assigned to keypad keys (see Appendix B for the keypad diagram):

- Pressing key 3 assigns the scroll attribute to the next display in the display list after the current scrolling dipslay. So, to select a display as the current scrolling display, press key 3 repeatedly until the word "scroll" appears on the top line of that display.
- Press key 8, 2, 6, or 4 to scroll up, down, right, or left, respectively. The amount of scroll depends on which key state you use (DEFAULT, GOLD, or BLUE).

8.3.2 Showing, Hiding, Removing, and Canceling a Display

The DISPLAY command is the most versatile command for manipulating existing displays. The basic syntax is:

```
DISPLAY display-name [AT window-specification] [display-kind]
```

The display name must be that of an existing display. The other parameters let you change the window specification and the display kind (see Sections 8.5 and 8.6 for information on how to specify windows and display kinds).

When used without a qualifier or any parameters, the DISPLAY command simply puts a display on top of the pasteboard where it appears through its current window. For example, the following command shows the display INST through its current window:

```
DBG> DISPLAY INST
```

Keypad key 9, which is bound to the command DISPLAY %NEXTDISP, lets you achieve this effect for each display in the display list. The built-in function %NEXTDISP signifies the next display in the list (Appendix D identifies all screen-related built-in functions). Each time you press key 9, the next display is put on top of the pasteboard, in its current window. Note that,

by default, the top line of display OUT (which identifies the display) coincides with the bottom line of display SRC. If SRC is on top of the pasteboard, its bottom line will hide the top line of OUT (keep this in mind when using the DISPLAY command and associated keypad keys to put various displays on top of the pasteboard).

To *hide* a display at the bottom of the pasteboard, use the DISPLAY/HIDE command. This command simply changes the order of that display in the display list.

To *remove* a display from the pasteboard so that it is no longer seen (yet is not permanently deleted), use the DISPLAY/REMOVE command. To put a removed display back on the pasteboard, use the DISPLAY command.

To *delete* a display permanently, use the CANCEL DISPLAY command. To recreate the display, use the SET DISPLAY command, which is described in Section 8.4.

Note that you cannot hide, remove, or delete the PROMPT display.

To identify the displays that currently exist, use the SHOW DISPLAY command. They are listed according to their order on the display list. The display that is on top of the pasteboard is listed last.

See the Command Dictionary for information on the various options provided by the DISPLAY command qualifiers.

8.3.3 Moving a Display Across the Screen

The MOVE command lets you move a display across the screen. The qualifiers /UP:n, /DOWN:n, /RIGHT:n, and /LEFT:n specify the direction and the number of lines or columns by which to move the display. If you do not specify a display, the current scrolling display is moved.

The easiest way to move a display is by using keypad keys:

- Press key 3 repeatedly as needed to select the current scrolling display.
- Put the keypad in the MOVE state, then use keys 8, 2, 4, or 6 to move the display up, down, left, or right, respectively (see Appendix B).

8.3.4 Expanding or Contracting a Display

The EXPAND command lets you expand or contract a display. The qualifiers /UP:n, /DOWN:n, /RIGHT:n, and /LEFT:n specify the direction and the number of lines or columns by which to expand or contract the display (to contract a display, specify negative integer values with these qualifiers). If you do not specify a display, the current scrolling display is expanded or contracted.

The easiest way to expand or contract a display is by using keypad keys.

- Press key 3 repeatedly as needed to select the current scrolling display.
- Put the keypad in the EXPAND or CONTRACT state, then use keys 8, 2, 4, or 6 to expand or contract the display vertically or horizontally (see Appendix B).

Note that the PROMPT display cannot be contracted (or expanded) horizontally. Also, it cannot be contracted vertically to less than two lines.

8.4 Creating a New Display

To create a new screen display, use the SET DISPLAY command. The basic syntax is:

```
SET DISPLAY display-name [AT window-specification] [display-kind]
```

The display name can be any name that is not already used to name a display. When you create a new display, it is placed on top of the pasteboard, on top of any existing displays (except for the predefined PROMPT display, which cannot be hidden). The display name appears at the top left corner of the display window.

Section 8.5 explains the options you have for specifying windows. If you do not provide a window specification, the display is positioned in the upper or lower half of the screen, alternating between these locations as you create new displays.

Section 8.6 explains the options you have for specifying display kinds. If you do not specify a display kind, an *output* display is created.

For example, the following command creates a new output display named OUT2. The window associated with OUT2 is either the top or bottom half of the screen.

```
DBG> SET DISPLAY OUT2
```

The following command creates a new *"DO"* display named EXAM_XY that is located in the right third quarter (RQ3) of the screen. This display shows the current value of variables X and Y and is updated whenever the debugger gains control from the program.

```
DBG> SET DISPLAY EXAM_XY AT RQ3 DO (EXAMINE X,Y)
```

See the Command Dictionary for information on the various options provided by the SET DISPLAY command qualifiers.

8.5 Specifying a Display Window

Display windows may occupy any rectangular portion of the screen.

You can specify a display window when creating a display with the SET DISPLAY command. You can also change the window currently associated with a display by specifying a new window with the DISPLAY command. You have the following options:

- Specify a window in terms of lines and columns.
- Use the name of a predefined window, such as H1.
- Use the name of a window definition previously established with the SET WINDOW command.

Each of these techniques is described in the next sections. When specifying windows, keep in mind that the PROMPT display always remains on top of the display pasteboard and, therefore, will occlude any part of another display that shares the same region of the screen.

Display windows, regardless of how specified, are *dynamic*. This means that, if you use a SET TERMINAL command to change the screen height or width, the window associated with a display will expand or contract in proportion to the new screen height or width.

8.5.1 Specifying a Window in Terms of Lines and Columns

The general form of a window specification is *(start-line,line-count[,start-column,column-count])*. For example, the following command creates the output display CALLS and specifies that its window be 7 lines deep starting at line 10, and 30 columns wide starting at column 50:

```
DBG> SET DISPLAY CALLS AT (10,7,50,30)
```

If you do not specify *start-column* or *column-count*, the window occupies the full width of the screen.

8.5.2 The Predefined Windows

The debugger provides many predefined windows. These have short symbolic names that you can use in the SET DISPLAY and DISPLAY commands instead of having to specify lines and columns. For example, the following command creates the output display ZIP and specifies that its window be RH1 (the top right half of the screen).

```
DBG> SET DISPLAY ZIP AT RH1
```

The predefined windows are all identified in Appendix C. The SHOW WINDOW command also displays these definitions.

8.5.3 Creating a New Window Definition

Although the predefined windows should be adequate for most situations, you can create a new window definition with the SET WINDOW command. This command, which has the following form, associates a window name with a window specification:

```
SET WINDOW window-name AT (start-line,line-count[,start-col,col-count])
```

After creating a window definition, you can simply use its name (like that of a predefined window) in a SET DISPLAY or DISPLAY command. In the following example, the window definition MIDDLE is established. That definition is then used to display OUT through the window MIDDLE.

```
DBG> SET WINDOW MIDDLE AT (9,4,30,20)
DBG> DISPLAY OUT AT MIDDLE
```

To identify all current window definitions, use the SHOW WINDOW command. To delete a window definition, use the CANCEL WINDOW command.

8.6 Specifying the Display Kind

Every display has a *display kind*. The display kind determines the type of information a display contains and how that information is generated.

Typically, you specify a display kind when using the SET DISPLAY command to create a new display (if you do not specify a display kind, an *output* display is created). You can also specify a display kind when using the DISPLAY command to change a display kind. The keywords and associated parameters with which you specify a display kind are listed below. Each of

these options is explained in the sections that follow (refer also to the displays illustrated in Section 8.2).

DO (command-list)
INSTRUCTION
INSTRUCTION (command)
OUTPUT
REGISTER
SOURCE
SOURCE (command)

The contents of a *register* display are generated and updated automatically by the debugger. The contents of other kinds of displays are generated by commands, and these display kinds fall into two general groups.

A display that belongs to one of the following display kinds has its contents updated automatically according to the command or command list you supply when defining that display.

DO (command-list)
INSTRUCTION (command)
SOURCE (command)

The command list specified is executed each time the debugger gains control from your program, provided the display is not marked as removed. The output of the commands forms the new contents of the display. If the display is marked as removed, the debugger does not execute the command list until you view that display (marking that display as unremoved).

A display that belongs to one of the following display kinds derives its contents from commands that you issue interactively:

INSTRUCTION
OUTPUT
SOURCE

To direct debugger output to a specific display in this group, you must first select it with the SELECT command. The technique is explained in the next sections and, in further detail, in Section 8.7. Once a display is selected for a certain type of output, the output from your commands forms the contents of the display.

The default size of the memory buffer associated with any newly created display is 64 lines. For source and instruction displays, the size of the buffer only affects performance. In the case of a source display, source files will be paged in as necessary as you scroll through the module. In the case of an instruction display, the instructions will be decoded from the image as necessary as you scroll through the routine.

For output and DO displays, the buffer size defines how many lines of text the display will hold. If you add more text to the display, the oldest lines are discarded to make room for the new text. You can use the /SIZE qualifier on the SET DISPLAY and DISPLAY commands to change the buffer size.

8.6.1 DO (command-list) Display Kind

A "DO" display is an automatically updated display. The commands in the command list are executed each time the debugger gains control from your program. Their output forms the content of the display, erasing any previous content.

For example, the following command creates the DO display CALLS at window Q3. Each time the debugger gains control from the program, the SHOW CALLS command is executed and the output is displayed in CALLS, replacing any previous contents.

```
DBG> SET DISPLAY CALLS AT Q3 DO (SHOW CALLS)
```

8.6.2 INSTRUCTION Display Kind

An instruction display shows the output of an EXAMINE/INSTRUCTION command within the assembly-language instruction code of the routine being debugged (the instructions displayed are decoded from the image being debugged). One line is devoted to each instruction. Source line numbers corresponding to the instructions are displayed in the left column. The instruction at the location being examined is centered in the display and is marked by an arrow in the left column.

Note that, before anything can be written to an instruction display, you must select it as the *current instruction display* with the SELECT/INSTRUCTION command.

In the following example, the SET DISPLAY command creates the instruction display INST2 at RH1. The SELECT/INSTRUCTION command then selects INST2 as the current instruction display. When the EXAMINE/INSTRUCTION X command is executed, window RH1 fills with the instruction code surrounding the location X. The arrow points to the instruction at location X, which is centered in the display.

```
DBG> SET DISPLAY INST2 AT RH1 INSTRUCTION
DBG> SELECT/INSTRUCTION INST2
DBG> EXAMINE/INSTRUCTION X
```

Each subsequent EXAMINE/INSTRUCTION command will update the display.

8.6.3 INSTRUCTION (command) Display Kind

This is an instruction display that is automatically updated with the output of the command specified. That command, which must be an EXAMINE/INSTRUCTION command, is executed each time the debugger gains control from your program.

For example, the following command creates the instruction display INST3 at window RS45. Each time the debugger gains control, the command EXAMINE/INSTRUCTION .0\%PC is executed (it displays the instruction at the current PC location), updating the display.

```
DBG> SET DISPLAY INST3 AT RS45 INSTRUCTION (EX/INST .0\%PC)
```

This command creates a display that functions like the predefined display INST.

If an automatically updated instruction display is selected as the current instruction display, it will be updated like a simple instruction display by an interactive EXAMINE/INSTRUCTION command (in addition to being updated by its built-in command).

8.6.4 OUTPUT Display Kind

An output display shows any debugger output that is not directed to some other display. New output is appended to the previous contents of the display.

Note that, before anything can be written to an output display, it must be selected as the *current output display* with the SELECT/OUTPUT command, or as the *current error display* with the SELECT/ERROR command, or as the *current input display* with the SELECT/INPUT command. See Section 8.7 for more information on using the SELECT command with output displays.

In the following example, the SET DISPLAY command creates the output display OUT2 at window T2 (the display kind OUTPUT could have been omitted from this example, since it is the default kind). The SELECT /OUTPUT command then selects OUT2 as the current output display. These two commands create a display that functions like the predefined display OUT.

```
DBG> SET DISPLAY OUT2 AT T2 OUTPUT
DBG> SELECT/OUTPUT OUT2
```

OUT2 will now collect any debugger output that is not directed to another display. For example:

- The output of a SHOW CALLS command will go to OUT2.
- If no instruction display has been selected as the current instruction display, the output of an EXAMINE/INSTRUCTION command will go to OUT2.
- By default, debugger diagnostic messages are directed to the PROMPT display. They may be directed to OUT2 with the SELECT/ERROR command.

8.6.5 REGISTER Display Kind

A register display is an automatically updated display that shows the current contents of all VAX machine registers, the four condition code bits (C, V, Z, and N) of the processor status longword (PSL), and the top several values on the stack and on the current argument list. The display is updated each time the debugger gains control from your program. Any values that have changed are highlighted.

As opposed to other display kinds, a register display is not dynamic by default. It does not change its window dimensions in proportion if you change the terminal screen height or width.

8.6.6 SOURCE Display Kind

A source display shows the output of a TYPE or EXAMINE/SOURCE command within the source code of the module being debugged, *if that* source code is available. Source line numbers are displayed in the left column. The source line that is the output of the command is centered in the display and is marked by an arrow in the left column. If a range of lines is specified with the TYPE command, the lines are centered in the display, but no arrow is shown.

Note that, before anything can be written to a source display, you must select it as the *current source display* with the SELECT/SOURCE command.

In the following example, the SET DISPLAY command creates the source display SRC2 at Q2. The SELECT/SOURCE command then selects SRC2 as the current source display. When the TYPE 34 command is executed, window RH1 fills with the source code surrounding line 34 of the module being debugged. The arrow points to line 34, which is centered in the display.

```
DBG> SET DISPLAY SRC2 AT Q2 SOURCE
DBG> SELECT/SOURCE SRC2
DBG> TYPE 34
```

Each subsequent TYPE or EXAMINE/SOURCE command will update the display.

8.6.7 SOURCE (command) Display Kind

This is a source display that is automatically updated with the output of the command specified. That command, which must be an EXAMINE/SOURCE or TYPE command, is executed each time the debugger gains control from your program.

For example, the following command creates a source display SRC3 at window RS45. Each time the debugger gains control, the command EXAMINE/SOURCE .%SOURCE_SCOPE\%PC is executed, updating the display.

```
DBG> SET DISPLAY SRC3 AT RS45 SOURCE (EX/SOURCE .%SOURCE_SCOPE\%PC)
```

This command creates a display that functions like the predefined display SRC. %SOURCE_SCOPE is a built-in scope that signifies scope 0 when source lines are available for scope 0. Otherwise, it signifies scope N, where N is the first level down the call stack for which source lines are available.

If an automatically updated source display is selected as the current source display, it will be updated like a simple source display by an interactive EXAMINE/SOURCE or TYPE command (in addition to being updated by its built-in command).

8.6.8 PROGRAM Display Kind

The PROMPT display belongs to the special display kind "program." Note that PROMPT is the only display of that kind. You cannot specify that display kind in a SET DISPLAY or DISPLAY command.

To avoid possible confusion, the PROMPT display has several restrictions (see Section 8.2.3).

8.7 Assigning Display Attributes

In screen mode, the output from commands you issue interactively is directed to various displays according to the type of output and the attributes assigned to these displays. For example, debugger diagnostic messages go to the display that has the *error* attribute (the *current error display*). By assigning one or more attributes to a display, you can mix or isolate different kinds of information.

The attributes have the following names: error, input, instruction, output, program, prompt, scroll, and source. When a display is assigned an attribute, the name of that attribute appears in lowercase letters on the top border of its window, to the right of the display name. Note that the scroll attribute does not affect debugger output but is used to control the default display for the SCROLL, MOVE, and EXPAND commands.

By default, attributes are assigned to the predefined displays as follows:

- For all languages except MACRO, SRC has the source and scroll attributes.
- For MACRO, INST has the instruction and scroll attributes.
- OUT has the output attribute.
- PROMPT has the prompt, program, and error attributes.

To assign an attribute to a display, use the SELECT command with the qualifier of the same name as the attribute. In the following example, the SET DISPLAY command creates the output display ZIP. The SELECT/OUTPUT command then selects ZIP as the *current output display*—the display that has the output attribute. After this command is executed, the word "output" will disappear from the top border of the predefined output display OUT and will appear instead on display ZIP, and all debugger output formerly directed to OUT will now be directed to ZIP.

```
DBG> SET DISPLAY ZIP OUTPUT
DBG> SELECT/OUTPUT ZIP
```

Specific attributes may be assigned only to certain display kinds. The following list identifies each of the SELECT command qualifiers, its effect, and the display kinds to which you can assign that attribute.

SELECT Qualifier	Description
/ERROR	Selects the specified display as the current error display. Directs any subsequent debugger diagnostic message to that display. It must be either an output display or the PROMPT display. If no display is specified, selects the PROMPT display as the current error display.
/INPUT	Selects the specified display as the current input display. Echoes any subsequent debugger input in that display. It must be an output display. If no display is specified, unselects the current input display: debugger input is not echoed to any display.

SELECT Qualifier	Description
/INSTRUCTION	Selects the specified display as the current instruction display. Directs the output of any subsequent EXAMINE /INSTRUCTION command to that display. It must be an instruction display. Keypad key sequence BLUE-COMMA selects the next instruction display in the display list as the current instruction display. If no display is specified, unselects the current instruction display: no display has the instruction attribute.
/OUTPUT	Selects the specified display as the current output display. Directs any subsequent debugger output to that display, except where a particular type of output is being directed to another display (such as diagnostic messages going to the current error display). The specified display must be either an output display or the PROMPT display. Keypad key sequence GOLD-3 selects the next output display in the display list as the current output display. If no display is specified, selects the PROMPT display as the current output display.
/PROGRAM	Selects the specified display as the current program display. Tries to force any subsequent program input or output to that display. Currently, only the PROMPT display may be specified. If no display is specified, unselects the current program display: program output is no longer forced to the PROMPT display.
/PROMPT	Selects the specified display as the current prompt display, where the debugger prompts for input. Currently, only the PROMPT display may be specified. You cannot unselect the PROMPT display.
/SCROLL	Selects the specified display as the current scrolling display. Makes that display the default display for any subsequent SCROLL, MOVE, or EXPAND command. You can specify any display (however, note that the PROMPT display cannot be scrolled). /SCROLL is the default if you do not specify a qualifier with the SELECT command. Key 3 selects as the current scrolling display the next display in the display list after the current scrolling display. If no display is specified, unselects the current scrolling display: no display has the scroll attribute.
/SOURCE	Selects the specified display as the current source display. Directs the output of any subsequent TYPE or EXAMINE /SOURCE command to that display. It must be a source display. Keypad key sequence BLUE-3 selects the next source display in the display list as the current source display. If no display is specified, unselects the current source display: no display has the source attribute.

Subject to the restrictions listed, a display can have several attributes. In the preceding example, ZIP was selected as the current output display. In the next example, ZIP is further selected as the current input, error, and scrolling display. After these commands are executed, debugger input, output, and diagnostics will be logged in ZIP in the proper sequence as they occur, and ZIP will be the current scrolling display.

```
DBG> SELECT/INPUT/ERROR/SCROLL ZIP
```

To identify the displays currently selected for each of the display attributes, use the SHOW SELECT command.

If you use the SELECT command with a particular qualifier but without specifying a display name, the effect is typically to de-assign that attribute (to "unselect" the display that had the attribute). The exact effect depends on the attribute, as described in the preceding list.

8.8 A Sample Display Configuration

How to best use screen mode depends on your personal style and on what type of bug you are looking for. You may be satisfied to simply use the predefined displays. On the other hand, especially if you have access to a larger screen, you may want to create additional displays for various purposes. The following example may give you some ideas.

Assume you are debugging in a high-level language and are interested in tracing the execution of your program through several routine calls.

First set up the default screen configuration—that is, SRC in H1, OUT in S45, and PROMPT in S6 (the keypad key sequence BLUE-MINUS gives this configuration). SRC will show the source code of the currently executing module.

The next command creates a source display named SRC2 in RH1 that shows the PC location at scope 1 (one level down the call stack, at the call to the currently executing module):

```
DBG> SET DISPLAY SRC2 AT RH1 SOURCE (EXAMINE/SOURCE .1\%PC)
```

Thus the left half of your screen shows the currently executing routine, whereas the right half shows the caller of that routine.

The next command creates a DO display named CALLS at S4 that executes the SHOW CALLS command each time the debugger gains control from the program:

```
DBG> SET DISPLAY CALLS AT S4 DO (SHOW CALLS)
```

Since the top half of OUT is now hidden by CALLS, make OUT's window smaller:

```
DBG> DISPLAY OUT AT S5
```

You can create a similar display configuration with instruction displays instead of source displays.

8.9 Saving Displays and the Screen State

The SAVE command lets you make a "snapshot" of an existing display and save that copy as a new display. This is useful if, for example, you later want to refer to the current contents of an automatically updated display (such as a DO display).

In the following example, the SAVE command saves the current contents of display CALLS into display CALLS4, which is created by the command:

```
DBG> SAVE CALLS AS CALLS4
```

The new display is removed from the pasteboard. So, to view its contents use the DISPLAY command:

```
DBG> DISPLAY CALLS4
```

The EXTRACT command has two uses. First, it lets you save the contents of a display in a text file. For example, the following command extracts the contents of display CALLS, appending the resulting text to the file COB34.TXT:

```
DBG> EXTRACT/APPEND CALLS COB34
```

Second, the EXTRACT/SCREEN_LAYOUT command lets you create a command procedure that may later be invoked during a debugging session to re-create the previous state of the screen. In the following example, the EXTRACT/SCREEN_LAYOUT command creates a command procedure with the default specification SYS$DISK:[ ]DBGSCREEN.COM. The file contains all the commands needed to recreate the current state of the screen.

```
DBG> EXTRACT/SCREEN_LAYOUT
   .
   .
   .
DBG> @DBGSCREEN
```

Note that you cannot save the PROMPT display as another display, or extract it into a file.

8.10 Changing the Screen Height and Width

During a debugging session, you may want to change the height or width of your terminal screen. One reason may be to accommodate long lines that would wrap if displayed across 80 columns. Or, if you are using a MicroVAX workstation, you may want to reformat your debugger window relative to other windows.

To change the screen height or width, use the SET TERMINAL command. The general effect of the command is the same whether you are at a VT-series terminal or at a MicroVAX workstation.

In this example, assume you are using a workstation in its default emulated VT100-screen mode, with a screen size of 24 lines by 80 columns. You have invoked the debugger and are using it in screen mode. You now want to take advantage of the larger screen. The following command increases the screen height and width of the debugger window to 35 lines and 110 columns respectively:

```
DBG> SET TERMINAL/PAGE:35/WIDTH:110
```

By default, all displays except for register displays are "dynamic." A dynamic display automatically adjusts its window dimensions in proportion when a SET TERMINAL command changes the screen height or width. This means that, when using the SET TERMINAL command, you preserve the relative positions of your displays. The /[NO]DYNAMIC qualifier on the DISPLAY and SET DISPLAY commands lets you control whether or not a display, including a register display, is dynamic. If a display is not dynamic, it does not change its window coordinates after you issue a SET TERMINAL command (you can then use the DISPLAY, MOVE, or EXPAND commands, or various keypad key combinations, to move or resize a display).

To see the current terminal width and height being used by the debugger, use the SHOW TERMINAL command.

Note that the debugger's SET TERMINAL command does not affect the terminal screen size at DCL level. When you exit the debugger, the original screen size is maintained.

9 Tailoring the Debugger

The debugger has certain specialized features that allow you to tailor your debugging sessions to the design of your particular program as well as your own individual needs. These features allow you to

- Allocate additional memory
- Use control structures
- Declare parameters to command procedures
- Define and undefine commands
- Define and undefine keys

9.1 Allocating Additional Memory

The debugger has an internal memory pool from which it provides both temporary and permanent memory. The data structures the debugger builds when it processes a command need only temporary memory. Since these data structures are discarded as soon as the command is processed, the memory they occupy can be reused for other processing. On the other hand, data structures (such as breakpoints) which remain during the debugging session need permanent memory.

Most of the memory pool is used to store the run-time symbol table (RST) for the modules that you have set with the SET MODULE command. Typically, the debugger memory pool allows you to set your six largest modules.

However, your program may have many large modules that you want to set all at the same time, and the normal memory allocation may not be adequate to include all these symbols in the RST. To solve this problem, you can allocate additional memory with either the ALLOCATE or the SET MODULE/ALLOCATE command.

9.1.1 The ALLOCATE Command

The ALLOCATE command gives you explicit control over the size of the debugger memory pool. It allows you to increase the virtual address space allotted to the debugger. This is the space that the debugger uses to store symbol table and eventpoint information, for instance. You can use this command if you get the "NOFREE" error message indicating that the debugger has run out of free virtual memory. The command has the form

```
ALLOCATE byte-count
```

The byte-count parameter specifies the number of bytes you want to add to the debugger memory pool. You must request at least 1000 bytes. You can find out the size of the memory pool by giving the command SHOW MODULE. The size is shown in the "remaining size:" line. This number represents the number of free bytes that are still in the pool.

The SHOW MODULE command also tells how many bytes are needed to set each module in your program. You can use these size requirements as a guideline for deciding how many bytes to allocate.

You should allocate extra memory only at the beginning of your debugging session. You can, however, allocate additional memory in the middle of a debugging session, but memory allocation within your program may be affected if your program calls LIB$GET_VM. Since the debugger also uses LIB$GET_VM (to increase the size of memory pool), the memory your program allocates may be interleaved with that allocated by the debugger. This is usually not a problem, but in some cases, the behavior of your program may be affected.

For more information, see the ALLOCATE command in the Command Dictionary.

9.1.2 The SET MODULE/ALLOCATE Command

The /ALLOCATE qualifier on SET MODULE tells the debugger to allocate extra storage if necessary in order to complete the SET MODULE command.

Normally the debugger tries to allocate all its storage at the beginning of the debugging session, before your program has run. So, any SET MODULE command that is issued before you execute a STEP or GO command will allocate storage if necessary, whether or not you specify /ALLOCATE.

After your program has run, the debugger will not allocate memory because the debugger memory may be interleaved with your program's memory and this may affect the behavior of the program. Therefore, after your program has run, if you need extra memory allocation during a SET MODULE, you must explicitly ask for it by using SET MODULE/ALLOCATE.

The following example shows how to use the ALLOCATE command, the SET MODULE/ALLOCATE and the SHOW MODULE command.

Example 9–36 Using the ALLOCATE, the SET MODULE/ALLOCATE, the SET MODULE/ALL, and the SHOW MODULE Commands

```
DBG>SHOW MODULE                 !Display module names and sizes.

module name                      symbols   language   size

DISTANCE$MAIN                    yes       FORTRAN   15600
DIST                             no        FORTRAN    5420
SUB1                             no        FORTRAN   10000
SUB2                             no        FORTRAN    5260
SHARE$FORRTL                     no        Image         0
SHARE$LIBRTL                     no        Image         0
SHARE$MTHRTL                     no        Image         0
SHARE$DEBUG                      no        Image         0
SHARE$LBRSHR                     no        Image         0
SHARE$PLIRTL                     no        Image         0
SHARE$SCRSHR                     no        Image         0
SHARE$SMGSHR                     no        Image         0

total modules: 12.               remaining size: 46260.

DBG>SET MODULE/ALL              !Set all modules at the beginning
                                !of the debugging session.

DBG>SHOW MODULE                 !Display all set modules.  Note that
                                !no shareable image is set.

module name                      symbols   language   size

DISTANCE$MAIN                    yes       FORTRAN   15600
DIST                             yes       FORTRAN    5420
SUB1                             yes       FORTRAN   10000
SUB2                             yes       FORTRAN    5260
SHARE$FORRTL                     no        Image         0
SHARE$LIBRTL                     no        Image         0
SHARE$MTHRTL                     no        Image         0
SHARE$DEBUG                      no        Image         0
SHARE$LBRSHR                     no        Image         0
SHARE$PLIRTL                     no        Image         0
SHARE$SCRSHR                     no        Image         0
SHARE$SMGSHR                     no        Image         0

total modules: 12.               remaining size: 20000.
```

(Continued on next page)

Example 9–36 (Cont.) Using the ALLOCATE, the SET MODULE/ALLOCATE, the SET MODULE/ALL, and the SHOW MODULE Commands

```
DBG>SET MODULE/ALLOCATE SHARE$FORRTL
                              !Explicitly request that the
                              !shareable image be set, regardless
                              !of size.
DBG>SHOW MODULE               !Display set modules.  Note that
                              !SHARE$FORRTL is now set.

module name                    symbols   language   size

DISTANCE$MAIN                  yes       FORTRAN   15600
DIST                           yes       FORTRAN    5420
SUB1                           yes       FORTRAN   10000
SUB2                           yes       FORTRAN    5260
SHARE$FORRTL                   yes       Image     11600
SHARE$LIBRTL                   no        Image         0
SHARE$MTHRTL                   no        Image         0
SHARE$DEBUG                    no        Image         0
SHARE$LBRSHR                   no        Image         0
SHARE$PLIRTL                   no        Image         0
SHARE$SCRSHR                   no        Image         0
SHARE$SMGSHR                   no        Image         0

total modules: 10.             remaining size:  8400.

DBG>SET MODULE SHARE$LIBRTL    !Try to set another module.
%DEBUG-E-NOFREE, no free storage available

DBG>ALLOCATE 15000             !Explicitly expand the memory pool
                               !by 15000 bytes.  Alternately, you
                               !could have used the command
                               !SET MODULE/ALLOCATE.

DBG>SET MODULE SHARE$LIBRTL    !Try to set the module again.

DBG>SHOW MODULE                !Display all modules.

module name                    symbols   language   size

DISTANCE$MAIN                  yes       FORTRAN   15600
DIST                           yes       FORTRAN    5420
SUB1                           yes       FORTRAN   10000
SUB2                           yes       FORTRAN    5260
SHARE$FORRTL                   yes       Image     11600
SHARE$LIBRTL                   yes       Image     19652
SHARE$MTHRTL                   no        Image         0
SHARE$DEBUG                    no        Image         0
SHARE$LBRSHR                   no        Image         0
SHARE$PLIRTL                   no        Image         0
SHARE$SCRSHR                   no        Image         0
SHARE$SMGSHR                   no        Image         0

total modules: 12.             remaining size:  4200.
```

For more information, see the SET MODULE command in the Command Dictionary.

9.2 Using Control Structures

The debugger's command language is a specialized programming language that consists primarily of primitive operations, or simple commands, such as EXAMINE and SET BREAK. However, as in other programming languages, the effectiveness and flexibility of these simple commands are enhanced by conditional or looping constructs. The debugger presently includes three such control structures: the FOR, IF, REPEAT, and WHILE commands.

9.2.1 The FOR Command

The FOR command allows a debugger command list to be executed iteratively for a specified number of times. It has the form

```
FOR name = expression1 TO expression2 [BY expression3] DO (command-list)
```

The behavior of the FOR command depends on the values of the expression3 parameter. If expression3 is positive, name is incremented from the value of expression1 by the value of expression3 until it is greater than the value of expression2.

If expression3 is negative, name is decremented from the value of expression1 by the value of expression3 until it is less than the value of expression2.

If expression3 is zero, the debugger returns an error message.

If expression3 is left out entirely, the debugger assumes it to have the value +1.

Example 9–37 shows how to use the FOR command (FORTRAN example).

Example 9–37 Using the FOR command

```
DBG>FOR COUNT = 1 TO 4 DO (STEP)                !Set up a loop that
DBG>stepped to DISTANCE$MAIN\%LINE 5            !steps to the next
     5:         STATUS=LIB$GET_LUN(LOG_UNIT)    !four lines.
stepped to DISTANCE$MAIN\%LINE 6
     6:         OPEN (UNIT=LOG_UNIT,FILE='FOR.DAT',STATUS='OLD')
stepped to DISTANCE$MAIN\%LINE 7
     7:         DO WHILE (X1 .NE. -9.9)
stepped to DISTANCE$MAIN\%LINE 8
     8:         READ (UNIT=LOG_UNIT,FMT=100) X1,X2,Y1,Y2

DBG>SHOW SYMBOL COUNT                           !Display symbol COUNT
defined COUNT

DBG>EVALUATE COUNT                              !Display current value
4                                                    !of COUNT

DBG>REPEAT 4 DO (STEP)                          !Another way of iterating
stepped to DISTANCE$MAIN\%LINE 10                    !a debugger command

    10:         D = DIST(X1,X2,Y1,Y2)
stepped to DISTANCE$MAIN\%LINE 16
    16:         TYPE 200,X1
stepped to DISTANCE$MAIN\%LINE 17
    17:         TYPE 201,X2
stepped to DISTANCE$MAIN\%LINE 18
    18:         TYPE 202,Y1
```

For more information, see the FOR command in the Command Dictionary.

9.2.2 The IF Command

The IF command allows a debugger command list to be executed conditionally if a language expression is evaluated as TRUE. It has the form

```
IF boolean-expression THEN (command-list) [ELSE (command-list)]
```

The IF command evaluates a language-specific boolean-expression. If the value is TRUE (as defined in the current language), the debugger command list in the THEN clause is executed. If the expression is FALSE, the command list in the ELSE clause is executed (if it is present).

Example 9–38 shows how to use the IF command (FORTRAN example).

Example 9–38 Using the IF Command

```
DBG>IF X1 .NE. -9.9 THEN (EXAMINE X2) ELSE (EX Y1)
DISTANCE$MAIN\X2:        0.0000000
                                        !Examine the variable
                                        !X2 if X1 is not equal
                                        !-9.9; otherwise,
                                        !examine the variable
                                        !Y1.

DBG>FOR COUNT = 1 TO 4 DO (IF X1 .NE. -9.9 THEN (STEP))
stepped to DISTANCE$MAIN\%LINE 19       !Set up a control
    19:          TYPE 203,Y2            !structure to step
stepped to DISTANCE$MAIN\%LINE 20       !four lines as long
    20:          TYPE 204,D             !as X1 does not equal
stepped to DISTANCE$MAIN\%LINE 21       !-9.9
    21:          END DO
stepped to DISTANCE$MAIN\%LINE 7
     7:          DO WHILE (X1 .NE. -9.9)
```

For more information, see the IF command in the Command Dictionary.

9.2.3 The WHILE Command

The WHILE command allows the debugger command lists to be executed iteratively until the language expression you have specified evaluates as FALSE. It has the form

```
WHILE boolean-expression DO (command-list)
```

If the value is TRUE (as defined in the current language), the debugger command list in the DO clause is executed. The command then repeats the sequence specified by the DO command-list, reevaluating the boolean-expression and executing the command-list until the expression is evaluated as FALSE.

If the boolean-expression is FALSE, the WHILE command terminates.

Example 9–39 shows how to use the WHILE command (FORTRAN example).

For more information, see the WHILE command in the Command Dictionary.

Example 9–39 Using the WHILE Command

```
DBG>WHILE X2 .LE. X1 DO (EX X2;STEP)
DISTANCE$MAIN\X2:       5.500000
stepped to DISTANCE$MAIN\%LINE 8
     8:         READ (UNIT=LOG_UNIT,FMT=100) X1,X2,Y1,Y2
DISTANCE$MAIN\X2:       5.500000        !Examine the variable
stepped to DISTANCE$MAIN\%LINE 10       !X2 and then step
    10:         D = DIST(X1,X2,Y1,Y2)   !to the next line
DISTANCE$MAIN\X2:       5.000000        !while it is less
stepped to DISTANCE$MAIN\%LINE 16       !than the variable
    16:         TYPE 200,X1             !X1.
DISTANCE$MAIN\X2:       5.000000
stepped to DISTANCE$MAIN\%LINE 17
    17:         TYPE 201,X2
DISTANCE$MAIN\X2:       5.000000
```

9.3 Declaring Parameters to Command Procedures

Another special feature of the debugger, the DECLARE command, allows you to pass parameters to command procedures. You can use this command only in a debugger command procedure.

Each parameter declaration has the same effect as a DEFINE command: it binds a name (the formal parameter) to a value or other construct (the actual parameter).

The format of the DECLARE command is

```
DECLARE fname:pkind [,fname:pkind [,...]]
```

The fname parameter specifies the formal parameter name. The pkind parameter may take the types below.

ADDRESS	Causes the actual parameter to be interpreted as an address expression. It has the same effect as the command DEFINE /ADDRESS fname = actual-parameter.
VALUE	Causes the actual parameter to be interpreted as a language expression. It has the same effect as the command DEFINE /VALUE fname = actual-parameter.
COMMAND	Causes the actual parameter to be interpreted as a command string. It has the same effect as the command DEFINE/COMMAND fname = actual-parameter.

You can specify more than one fname:pkind pair with a single DECLARE command. Each fname:pkind pair acts like a separate DECLARE command, and each pair binds one parameter. So, for instance, if you want to pass five parameters to a command procedure, you need five corresponding fname:pkind pairs in the command procedure itself. The pairs are always processed in the order in which you specify them.

The %PARCNT symbol specifies the number of actual parameters to the current command procedure. For example, suppose the command file ABC is invoked with the command @ABC 111,222,333. Inside ABC, %PARCNT then has the value 3 because there are three parameters on this particular call to ABC. %PARCNT is used in command procedures that can take a variable number of actual parameters. %PARCNT can only be used inside command files; it is not defined when commands are entered from the terminal.

A sample format follows:

```
EVALUATE %PARCNT
FOR I = 1 TO %PARCNT DO (DECLARE X:VALUE; EVALUATE X)
```

The following example shows a command procedure containing a DECLARE command and how it is used during a debugging session.

Example 9–40 Using the DECLARE Command in a Command Procedure

```
DBG>SET OUTPUT VERIFY                !Turn on VERIFY so the
                                     !command in the command
                                     !procedure will echo.

DBG>@EXAM OPTIONS                    !Execute the procedure.
%DEBUG-I-VERIFYICF, entering indirect command file "EXAM.COM"
 DECLARE P1:ADDRESS                  !OPTIONS is passed to
 EXAMINE P1                          !the procedure as P1.
CONV_EX$MAIN\OPTIONS                 !Examine OPTIONS.
    (1):        18
    (2):        1
    (3):        0
    (4):        1
    (5):        0
    (6):        0
    (7):        1
    (8):        2
    (9):        0
    (10):       0
    (11):       0
    (12):       0
    (13):       0
    (14):       0
    (15):       0
    (16):       0
    (17):       0
    (18):       1
    (19):       0
%DEBUG-I-VERIFYICF, exiting indirect command file "EXAM.COM"
```

9.4 Defining and Undefining Commands

The debugger has yet another special feature that allows you to define your own commands. With the DEFINE/COMMAND command, you can assign character strings to symbolic names. When the symbolic name appears at the start of a subsequent debugger command, the name is replaced by the string it represents. As a result, the symbolic name acts like a new debugger command.

To define complex commands, you may need to use indirect command files with parameters (for example, DEFINE/COMMAND DUMP = "@DUMP.COM"). For more information on declaring parameters to command files, see the Section 9.9.3.

The DEFINE/COMMAND command provides essentially the same capabilities as the DCL command "symbol:=parameter."

Example 9–41 shows how to define your own debugger commands.

Example 9–41 Defining Debugger Commands

```
DBG> DEFINE/COMMAND EXN = "EXAMINE %NEXTLOC"
DBG> SHOW SYM/DEFINED EXN                !Define the command EXN.
defined EXN                              !Display that it has
    bound to: "EXAMINE %NEXTLOC"         !been defined.
    was defined /command

DBG> EX X1                               !Examine a location.
DISTANCE$MAIN\X1:       0.0000000
DBG> EXN                                 !Use the defined symbol EXN
DISTANCE$MAIN\X2:       0.0000000        !to examine the next
DBG> EXN                                 !locations.
DISTANCE$MAIN\Y1:       0.0000000
```

9.5 Defining and Undefining Keys

The debugger allows you to modify the functions of the keypad keys to suit your individual needs. With the DEFINE/KEY, SHOW KEY, and DELETE /KEY (or UNDEFINE/KEY) commands, you can set, show, and delete key definitions, respectively. Before you can use this feature, keypad mode must be enabled with the SET MODE KEYPAD command (keypad mode is enabled by default). Keypad mode also lets you use the predefined functions of the keypad keys.

However, most of the keypad functions are associated with screen mode commands. If you want to use all the predefined functions of the keypad keys, you must first set the mode to screen mode with the SET MODE SCREEN command.

9.5.1 Assigning Key Definitions

The DEFINE/KEY command allows you to assign your own definitions to the numeric keypad keys. When you press the appropriate keypad key, the string associated with that key is inserted into the command line. You can also use qualifiers with the DEFINE/KEY command to associate certain attributes with the performance of the key. You must set mode to keypad to use this command.

The DEFINE/KEY command has the form

```
DEFINE/KEY key-name "equiv-string"
```

The key-name parameter specifies the name of the key that you are defining. In addition, for single words without special symbols, you do not need to enclose the equivalence string in quotation marks.

The standard definable key names are listed below.

Key-name	LK201	VT100-type	VT52-type
PF1	PF1	PF1	Blue
PF2	PF2	PF2	Red
PF3	PF3	PF3	Black
PF4	PF4	PF4	

Key-name	LK201	VT100-type	VT52-type
KP0, KP1, . . . ,KP9	Keypad 0, . . . ,9	Keypad 0, . . . ,9	Keypad 0, . . . ,9
PERIOD	Keypad period (.)	Keypad period (.)	
COMMA	Keypad comma (,)	Keypad comma (,)	
MINUS	Keypad minus (-)	Keypad minus (-)	
ENTER	ENTER	ENTER	ENTER
E1	Find		
E2	Insert Here		
E3	Remove		
E4	Select		
E5	Prev Screen		
E6	Next Screen		
HELP	Help		
DO	Do		
F6, F7, . . . , F20	F6, F7, . . . , F20		

The ability to define keys can save both time and keystrokes. If, for instance, you use a particular debugger command (or set of commands) repetitively, you can define one of the keypad keys as that command. Then when you press the defined key, the key definition is treated as a debugger command.

The DEFINE/KEY command provides the same capabilities as the DCL command DEFINE/KEY.

For additional information on defining keys, refer to the DEFINE/KEY command in the Command Dictionary as well as in the *VAX/VMS DCL Dictionary*.

Example 9–42 shows how to define the keypad keys with debugger commands.

Example 9–42 Using the DEFINE/KEY Command

```
DBG>SET MODE KEYPAD                    !Set the mode the keypad.
DBG>DEFINE/KEY PF1 "SET STEP/INSTRUCTION"
%DEBUG-I-DEFKEY, DEFAULT key PF1 has been defined
                                       !Define the PF1 key.
                                       !The definition is
                                       !echoed because /LOG
                                       !is the default.

DBG>DEFINE/KEY/TERMINATE PF3 "SET OUTPUT LOG"
%DEBUG-I-DEFKEY, DEFAULT key PF3 has been defined
                                       !Define the PF3 key.
                                       !When you press this
                                       !key, the command will
                                       !execute automatically
                                       !because of the
                                       !/TERMINATE qualifier.
```

9.5.2 Using Debugger-Defined Key Definitions

The debugger provides a number of predefined command definitions for the numeric keypad keys. Although most of these definitions apply only to screen mode functions, you can still use some of the keypad definitions while you are using the regular line-oriented debugger.

These keys and their functions are identified in Appendix B.

9.5.3 Showing Key Definitions

You can see the key definitions you have created with the SHOW KEY command.

The command has the form

```
SHOW KEY [key-name]
```

The key-name parameter specifies the name of the key whose definition you want to see. This command provides the same capabilities of the DCL command SHOW KEY.

If you want to see all key definitions, use the command SHOW KEY/ALL.

Example 9–43 shows how to show the definitions associated with each keypad key.

Example 9–43 Using the SHOW KEY Command

```
DBG>SHOW KEY PF1
DEFAULT keypad definitions:
  PF2 = "SET STEP/INSTRUCTION"  (echo,noterminate,nolock)
                                  !Display the command you
                                  !have associated with
                                  !the PF1 key.

DBG>SHOW KEY PF3
DEFAULT keypad definitions:
  PF3 = "SET OUTPUT LOG"  (echo,terminate,nolock)
                                  !Display the command you
                                  !have associated with
                                  !the PF3 key.

DBG>SHOW KEY KP3
DEFAULT keypad definitions:
  KP3 = "Select/Source %Nextsource"  (noecho,terminate,nolock)
                                  !Display the default
                                  !debugger command associated
                                  !with the PF3 key.
```

For additional information on displaying key definitions, refer to the SHOW KEY command in the Command Dictionary as well as in the *VAX/VMS DCL Dictionary*.

9.5.4 Deleting Key Definitions

To delete key definitions you have established with the DEFINE/KEY command, use either the DELETE/KEY or the UNDEFINE/KEY command.

They have the forms

```
DELETE/KEY [key-name]
```

```
UNDEFINE/KEY [key-name]
```

The key-name parameter specifies the name of the key whose definition you want to delete.

Both commands provide the same capabilities of the DCL command DELETE /KEY.

Example 9–44 shows how to use, how to delete or undefine the numeric keypad keys.

Example 9–44 Using the DELETE/KEY and UNDEFINE/KEY Commands

```
DBG>SET MODE KEYPAD              !Set mode to keypad.

DBG>DELETE/KEY PF1               !Delete the PF1 key.
%DEBUG-I-DELKEY, DEFAULT key PF1 has been deleted

DBG>UNDEFINE/KEY/NOLOG PF3
                                 !Undefine the PF3 key.
                                 !The debugger does not
                                 !display a message that
                                 !the key has been
                                 !undefined because of
                                 !the /LOG qualifier.
```

For more information on deleting key definition, see either the DELETE/KEY or the UNDEFINE/KEY commands in the Command Dictionary.

Part II Debugger Command Dictionary

The Debugger Command Dictionary describes each of the debugger commands in detail. The following information is provided for each command:

- Command description
- Command format
- Command parameters
- Command qualifiers
- One or more examples

Refer to the Preface for documentation conventions.

CD.1 Debugger Command Format

A command string is the complete specification of a debugger command. Although you can continue a command on more than one line, the term command string is used to define an entire command that is passed to the debugger.

A debugger command string consists of a verb and, possibly, parameters and qualifiers.

The verb specifies the command to be executed. Some debugger command strings may consist of only a verb or a verb pair. For example:

```
DBG> GO
DBG> SHOW IMAGE
```

A parameter specifies what the verb acts on (for example, a file specification). A qualifier describes or modifies the action taken by the verb. Some command strings may include one or more parameters or qualifiers. In the following examples, COUNT, I, J, and K, and PROG4 are parameters; /SCROLL and /OUTPUT are qualifiers.

```
DBG> SET WATCH COUNT
DBG> EXAMINE I,J,K
DBG> SELECT/SCROLL/OUTPUT PROG4
```

Some commands accept optional WHEN or DO clauses. A WHEN clause consists of the keyword WHEN followed by a conditional expression (within parentheses) that evaluates to TRUE or FALSE in the current language. A DO clause consists of the keyword DO followed by one or more command strings (within parentheses) that are to be executed. Multiple command strings should be separated by semicolons (;). These points are illustrated in the next example.

The following command string sets a breakpoint on routine SWAP that will be triggered whenever the value of J equals 4 during execution. When the breakpoint is triggered, the debugger executes the two command strings SHOW CALLS and EXAMINE I,K, in the order indicated.

```
DBG> SET BREAK SWAP WHEN (J = 4) DO (SHOW CALLS; EXAMINE I,K)
```

DO clauses are also used in some screen display definitions.

CD.2 Entering and Terminating Commands

You can enter debugger commands interactively at the terminal or store them within a command procedure to be invoked later with the @file-spec command. The conventions are described for each mode of operation.

CD.2.1 At the Terminal

When entering a command interactively, you can abbreviate a keyword (verb, qualifier, parameter) to as few characters as are needed to make it unique within the set of all keywords. However, some commonly used commands (for example, EXAMINE, DEPOSIT, GO, STEP) can be abbreviated to their first characters. Also, in some cases, the debugger interprets nonunique abbreviations correctly on the basis of context.

Pressing the RETURN key terminates the current line, causing the debugger to process it. To continue a long command string on another line, type a hyphen (-) before pressing RETURN. The debugger prompt will be prefixed with an underline character (_DBG>), indicating that the command string is still being accepted.

You can enter more than one command string on one line by separating them with a semicolon (;).

The command line editing functions that are available at the DCL prompt are also available at the debugger prompt, including command recall with the up-arrow and down-arrow keys. For example, pressing the left-arrow and right-arrow keys moves the cursor one character to the left and right, respectively; pressing CTRL/H and CTRL/E moves the cursor to the start and end of the line respectively; pressing CTRL/U deletes all the characters to the left of the cursor, and so on.

To interrupt a command that is in progress, press CTRL/Y. This will put you at DCL level. You can then type either CONTINUE or DEBUG to return to the debugging session. (See the description of CTRL/Y in the Command Dictionary.)

CD.2.2 In a Command Procedure

To maximize legibility, it is best to not abbreviate command keywords in a command procedure. In any case, as with DCL commands, do not abbreviate command keywords to less than four significant characters (not counting the negation /NO . . .), to avoid potential conflicts in future releases.

In a command procedure, the start of a new line terminates the previous line. To continue a command string on another line, type a hyphen (-) before starting the new line.

You can enter more than one command string on one line by separating them with a semicolon (;).

A comment is a string that is preceded by an exclamation point (!). Comments do not affect the execution of debugger commands.

ALLOCATE

Expands the debugger memory pool.

FORMAT

ALLOCATE *byte-count*

PARAMETERS

byte-count

Specifies the number of bytes by which you want to expand the debugger memory pool. The byte-count must be at least 1000; if it is not, you will receive an error message. You will also receive an error message if the system is unable to allocate the memory you requested.

QUALIFIERS

None.

DESCRIPTION

Module setting, the process by which symbol records are loaded into the debugger's run-time symbol table, requires memory to be allocated. By default, dynamic module setting is enabled and the debugger automatically allocates memory as it sets modules. If you issue the SET MODE NODYNAMIC command to disable dynamic module setting, you will have to set modules yourself. In that case, memory is not allocated automatically, and you may need to use the ALLOCATE command in order to set more modules. The SHOW MODULE command tells you how much space is required to set each module as well as how much space is available.

When you use the ALLOCATE command, the debugger calls $EXPREG to provide the extra memory. If your program also allocates memory dynamically, the location of the allocated memory in your program may be affected.

Related commands: (SET, CANCEL, SHOW) MODULE, SET MODE [NO]DYNAMIC.

EXAMPLE

```
DBG> ALLOCATE 20000
```

This command increases the size of the memory pool by an additional 20000 bytes.

@file-spec

Executes debugger commands contained in the specified command procedure.

FORMAT

@file-spec [P1 [,P2, . . . ,Pn]]

PARAMETERS

file-spec

Specifies the command procedure to be executed. For any part of the full file specification that is not provided, the debugger uses the file specification established with the last SET ATSIGN command, if any. If the missing part of the file specification was not established by a SET ATSIGN command, the debugger assumes SYS$DISK:[ ]DEBUG.COM as the default file specification.

P

Specifies a parameter that is declared inside the command procedure. Note that, unlike with DCL, parameters must be separated by commas. For more information on declaring parameters to command procedures, see the DECLARE command description.

QUALIFIERS

None.

DESCRIPTION

A command procedure can contain any debugger commands, including another @file-spec command. The debugger executes commands from the command procedure until it reaches an EXIT or QUIT command or the end of the file. At that point, the debugger returns control to the command stream that invoked the command procedure. A command stream can be the terminal, an outer (containing) command procedure, a DO clause in a SET BREAK, SET TRACE, or SET WATCH command, or a DO clause in a screen display definition.

If you enter SET OUTPUT VERIFY, all commands read from a command procedure are echoed on the current output device, as specified by DBG$OUTPUT (the default output device is SYS$OUTPUT, the terminal).

Related commands: (SET, SHOW) ATSIGN.

EXAMPLE

```
DBG> SET ATSIGN USER:[JONES.DEBUG].DBG
DBG> SET OUTPUT VERIFY
DBG> @MAIN
%DEBUG-I-VERIFYICF, entering indirect command file "MAIN"
 SET MODULE/ALL
 SET BREAK SUB1
 GO
break at routine MAIN\SUB1
 EXAMINE/WORD SUB1
SUB1\SUB1:  4800
 EXAMINE/INST
SUB1\SUB1+02:  MOVAL  L^SUB1\Y,R11
 EXIT
%DEBUG-I-VERIFYICF, exiting indirect command file "MAIN"
DBG>
```

The SET ATSIGN command establishes that debugger command procedures reside, by default, in USER:[JONES.DEBUG] and have a file type of DBG. The SET OUTPUT VERIFY command causes all commands to echo at the terminal before they execute. The @MAIN command causes the command procedure USER:[JONES.DEBUG]MAIN.DBG to be executed.

ATTACH

Passes control of your terminal from the current process to another process.

FORMAT

ATTACH *process-name*

PARAMETERS

process-name

Specifies the process to which your terminal is to be attached. The process must already exist before you try to attach to it. If the process name contains non-alphanumeric characters or spaces, you must enclose it in quotation marks ("").

QUALIFIERS

None.

DESCRIPTION

The ATTACH command allows you to go back and forth between a debugging session and your command interpreter, or between two debugging sessions. To do so, you must first spawn a subprocess (see the description of the SPAWN command); you can then attach to it whenever you want. To return to your original process with minimal system overhead, use another ATTACH command.

Related commands: SPAWN.

EXAMPLES

1

```
DBG> SPAWN
$ ATTACH JONES
%DEBUG-I-RETURNED, control returned to process JONES
DBG> ATTACH JONES_1
$
```

This series of commands spawns a subprocess named JONES_1 from the debugger (currently running in the process JONES) and then attaches to that subprocess.

2

```
DBG> ATTACH "Alpha One"
$
```

This example illustrates use of quotation marks to enclose a process name that contains a space.

CALL

Calls a routine that was linked with your program.

FORMAT

CALL *routine-name [(argument[, . . .])]*

PARAMETERS

routine-name

Specifies the name or the virtual address of the routine to be called.

argument

Specifies an argument that is required by the routine. Arguments can be passed by address (the default), by descriptor, by reference, and by value, as described below.

%ADDR Passes the argument by address. This is the default. The format is the following:

```
CALL routine-name (%ADDR address-expression)
```

The debugger evaluates the address expression and passes that address to the routine specified. For simple variables (such as X), the address of X is passed into the routine. This passing mechanism is how FORTRAN implements ROUTINE(X). In other words, for named variables, using %ADDR corresponds to a call by reference in FORTRAN. For other expressions, however, you must use the %REF function to call by reference.

%DESCR Passes the argument by descriptor. The format is the following:

```
CALL routine-name (%DESCR language-expression)
```

The debugger evaluates the language expression and builds a VAX-standard descriptor to describe the value. The descriptor is then passed to the routine you named. You would use this technique to pass strings to a FORTRAN routine.

%REF Passes the argument by reference. The format is the following:

```
CALL routine-name (%REF language-expression)
```

The debugger evaluates the language expression and passes a pointer to the value, into the called routine. This passing mechanism corresponds to the way FORTRAN passes the result of an expression.

%VAL Passes the argument by value. The format is the following:

```
CALL routine-name (%VAL language-expression)
```

The debugger evaluates the language expression and passes the value directly to the called routine.

QUALIFIERS

/[NO]AST

Controls whether aynchronous system traps (ASTs) are enabled or disabled during the execution of the called routine. /AST specifies that ASTs be enabled (can be delivered). /NOAST specifies that ASTs be disabled (cannot be delivered). If you do not specify /AST or /NOAST, ASTs will be enabled in the called routine if, and only if, they were currently enabled at the time of the call.

DESCRIPTION

The CALL command is one of four debugger commands that execute code (the others are GO, STEP, and EXIT). You can use the CALL command to call a routine and debug it independently of the rest of the program. You can also use the command to call user-written routines that dump debugging information.

The CALL command executes a routine whether or not your program actually includes a call to that routine, so long as the routine was linked with your program. You can debug unrelated routines by linking them with a dummy main program that has a transfer address, and then using the CALL command to execute them.

When you issue a CALL command, the debugger takes the following action:

1. Saves the current values of the general registers
2. Constructs an argument list
3. Executes a call to the routine specified in the command and passes any arguments
4. Executes the routine
5. Displays the value returned by the routine in R0
6. Restores the values of the general registers to the values they had just before the CALL command was executed
7. Issues the DBG> prompt

The debugger assumes that the called routine conforms to the VAX/VMS procedure calling standard (see the *VAX Architecture Handbook*). However, note that the debugger does not know about all the argument-passing mechanisms for all supported languages. You therefore may need to specify how to pass parameters—for example, use CALL SUB1(%VAL X) rather than CALL SUB1(X). Also, routines may not be called correctly if you are using complicated arguments such as arrays or records. See your language documentation for complete information on how arguments are passed to routines.

EXAMPLES

```
1   DBG> CALL SUB1(X)
    value returned is 19
```

This command calls the routine SUB1, passing "X" as the required parameter.

```
2   DBG> CALL SUB(%REF 1)
    value returned is 1
```

This command passes a pointer to a memory location containing the numeric literal 1, into the routine SUB.

CANCEL ALL

Cancels all breakpoints, tracepoints, and watchpoints. Restores any modes established with the SET MODE command to their default values. Restores the scope and type to their default values.

FORMAT

CANCEL ALL

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The CANCEL ALL command does the following:

- Cancels all eventpoints (breakpoints, tracepoints, watchpoints). This is equivalent to issuing the CANCEL BREAK, CANCEL TRACE, and CANCEL WATCH commands.
- Restores the scope search list to its default value (0,1,2, . . . ,*n*). This is equivalent to issuing the CANCEL SCOPE command.
- Restores the data type associated with a typed memory location to the compiler generated type. Restores the type associated with untyped memory locations to "longword integer". This is equivalent to issuing the CANCEL TYPE/OVERRIDE and SET TYPE LONGWORD commands.
- Restores the modes established with the SET MODE command to their default values. This is equivalent to issuing the following command:

```
DBG> SET MODE KEYPAD,NOSCREEN,DYNAMIC,LINE,SYMBOLIC,NOG_FLOAT,SCROLL
```

The CANCEL ALL command does not affect the current language setting or modules included in the run-time symbol table (SET MODULE).

Related commands: CANCEL BREAK, CANCEL TRACE, CANCEL WATCH, CANCEL SCOPE, CANCEL TYPE/OVERRIDE, (SET, CANCEL) MODE.

EXAMPLE

```
DBG> CANCEL ALL
```

This command cancels all the eventpoints you have set previously. It also restores scope, modes and types to their default values.

CANCEL BREAK

Cancels breakpoints.

FORMAT

CANCEL BREAK *[address-expression[, . . .]]*

PARAMETERS

address-expression

Specifies a breakpoint to be canceled. Do not use the wildcard character (*). Do not specify an address expression when using any of the qualifiers except for /EVENT.

QUALIFIERS

/ALL

Cancels all breakpoints. Do not use the wildcard character (*). When using /ALL, do not specify an address expression.

/BRANCH

Cancels the effect of a previous SET BREAK/BRANCH command. Do not specify an address expression with /BRANCH.

/CALL

Cancels the effect of a previous SET BREAK/CALL command. Do not specify an address expression with /CALL.

/EVENT=event-name

Note: /EVENT applies only to Ada and SCAN. See the VAX Ada and VAX SCAN documentation for complete information.

Cancels the effect of a previous SET BREAK/EVENT=event-name command. The effect of CANCEL BREAK/EVENT=event-name is symmetrical with the effect of SET BREAK/EVENT=event-name. To cancel a breakpoint, specify the event name and address expression (if any) exactly as you did with the SET BREAK/EVENT command, excluding any WHEN or DO clauses. Event names depend on the run-time facility and are identified in Appendix E for Ada and SCAN. You can also display the event names associated with the current run-time facility by issuing the SHOW EVENT_FACILITY command.

/EXCEPTION

Cancels the effect of a previous SET BREAK/EXCEPTION command. Do not specify an address expression with /EXCEPTION.

/INSTRUCTION

Cancels the effect of a previous SET BREAK/INSTRUCTION command. Do not specify an address expression with /INSTRUCTION.

CANCEL BREAK

/LINE

Cancels the effect of a previous SET BREAK/LINE command. Do not specify an address expression with /LINE.

DESCRIPTION

The effect of the CANCEL BREAK command is symmetrical with the effect of the SET BREAK command.

To cancel a breakpoint that was established at a specific location with the SET BREAK command, specify that same location with the CANCEL BREAK command. To cancel breakpoints that were established on a class of instructions or events by using a qualifier with the SET BREAK command (/CALL, /LINE, /EXCEPTION, /EVENT, and so on), specify that same qualifier with the CANCEL BREAK command.

Generally, you must specify either an address expression or a qualifier, but not both. The only exception is with the /EVENT qualifier, which requires that you specify an event name and lets you also specify an address expression for certain event names.

Note that the command CANCEL ALL also cancels all breakpoints.

Related commands: (SET, SHOW) BREAK, (SET, CANCEL) EXCEPTION BREAK, (SET, SHOW, CANCEL) TRACE, CANCEL ALL.

EXAMPLES

1
```
DBG> CANCEL BREAK MAIN\LOOP+10
```

This command cancels the breakpoint set at the address expression MAIN\LOOP+10.

2
```
DBG> CANCEL BREAK/ALL
```

This command cancels all breakpoints you have set previously.

CANCEL DISPLAY

Permanently deletes a screen display.

FORMAT

CANCEL DISPLAY *[disp-name[, . . .]]*

PARAMETERS

disp-name

Specifies the name of a display to be canceled. Do not specify the PROMPT display, which cannot be canceled. Do not use the wildcard character (*). When using /ALL, do not specify a display name.

QUALIFIERS

/ALL

Cancels all displays, except for the PROMPT display. When using /ALL, do not specify a display name.

DESCRIPTION

When a display is canceled, its contents are permanently lost, it is removed from the display list, and all the memory that was allocated to it is released.

You cannot cancel the PROMPT display.

Related commands: (SET, SHOW) DISPLAY, (SET, SHOW, CANCEL) WINDOW.

EXAMPLE

```
DBG> CANCEL DISPLAY SRC2
```

This command permanently deletes display SRC2.

```
DBG> CANCEL DISPLAY/ALL
```

This command permanently deletes all displays, except for the PROMPT display.

CANCEL EXCEPTION BREAK

Cancels exception breakpoints.

FORMAT CANCEL EXCEPTION BREAK

PARAMETERS *None.*

QUALIFIERS *None.*

DESCRIPTION

The command CANCEL EXCEPTION BREAK cancels the effect of the command SET EXCEPTION BREAK. As a result of the cancellation, exception conditions generated by your program are handled in the following way:

- The debugger fields and resignals the exception.
- Any exception handlers you have defined in your program are executed.
- If you have not defined an exception handler or if an exception handler that you have defined resignals the exception, a diagnostic message is issued and control is returned to the debugger, which then displays its prompt.

Related commands: SET EXCEPTION BREAK, (SET, CANCEL) BREAK /EXCEPTION, CANCEL ALL.

EXAMPLE

```
DBG> CANCEL EXCEPTION BREAK
```

This command cancels the effect of a previous SET EXCEPTION BREAK command.

CANCEL IMAGE

Removes run-time symbol table information for a shareable image.

FORMAT

CANCEL IMAGE *[image-name[, . . .]]*

PARAMETERS

image-name

Specifies a previously set shareable image that is to be canceled. Do not specify the main image, which cannot be canceled. Do not use the wildcard character (*). When using /ALL, do not specify an image name.

QUALIFIERS

/ALL

Specifies that all shareable images except the main image are to be canceled. When using /ALL, do not specify an image name.

DESCRIPTION

The CANCEL IMAGE command deallocates the data structures previously built to debug a shareable image by a SET IMAGE command. The CANCEL IMAGE command can be used when the run-time symbol table is full and debugger performance is slowed down.

If the current image (the image last set with the SET IMAGE command) is canceled, the main image becomes the current image.

Related commands: (SET, SHOW) IMAGE, (SET, SHOW, CANCEL) MODULE.

EXAMPLES

```
1  DBG> CANCEL IMAGE SHARE2,SHARE3
```

The CANCEL IMAGE command cancels shareable images SHARE2 and SHARE3. If either of these was the current image, the main image becomes the current image.

CANCEL MODE

Restores all modes controlled by the SET MODE command to their default values. Also restores the default input/output radix.

FORMAT

CANCEL MODE

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The effect of the CANCEL MODE command is equivalent to issuing the following commands:

```
DBG> SET MODE KEYPAD,NOSCREEN,DYNAMIC,LINE,SYMBOLIC,NOG_FLOAT,SCROLL
DBG> CANCEL RADIX
```

Note that, although the same default modes apply to all languages, the default radix is hexadecimal for BLISS and MACRO and decimal for all other languages.

Related commands: (SET, SHOW) MODE, (SET, SHOW, CANCEL) RADIX.

EXAMPLE

```
DBG> CANCEL MODE
```

This command restores the default radix mode and all default mode values.

CANCEL MODULE

Deletes the symbol records of a module from the run-time symbol table (RST).

FORMAT

CANCEL MODULE *[modname[, . . .]]*

PARAMETERS

modname

Specifies the name of a module whose symbol records are to be deleted from the RST. Do not use the wildcard character (*). When using /ALL, do not specify a module name.

QUALIFIERS

/ALL

Deletes the symbol records of all modules from the RST. When using /ALL, do not specify a module name.

/[NO]RELATED

Note: /[NO]RELATED applies only to Ada programs.

Controls whether the debugger deletes from the RST the symbol records of a module that is related to a specified module through a **with**-clause or subunit relationship.

CANCEL MODULE/RELATED (default) deletes symbol records for related modules as well as for those specified, but not for any module that is also related to another set module. The effect of CANCEL MODULE/RELATED is consistent with Ada's scope and visibility rules and depends on the actual relationship between modules. CANCEL MODULE/NORELATED deletes symbol records only for modules that are specified (no symbol records are deleted for related modules).

DESCRIPTION

Note: The (SET, SHOW, CANCEL) MODULE commands operate on modules in the current image. This is either the main image (by default) or the image established as the current image by a previous SET IMAGE command.

The CANCEL MODULE command is useful under two general conditions:

- If dynamic module setting is enabled (this is the *default* condition) and performance becomes a problem as the debugger sets more and more modules dynamically.
- If dynamic module setting is disabled and you cannot use the SET MODULE command (because the RST is full), but do not want to use the ALLOCATE command to increase the debugger memory pool.

Related commands: (SET, SHOW) MODULE, SET MODE [NO]DYNAMIC, ALLOCATE, (SET, SHOW, CANCEL) IMAGE.

CANCEL MODULE

EXAMPLES

1
```
DBG> CANCEL MODULE SUB1
```

This command removes the symbols of module SUB1 from the run-time symbol table.

2
```
DBG> CANCEL MODULE/ALL
```

This command removes the symbols of all modules from the run-time symbol table (RST).

CANCEL RADIX

Restores the default radix for the entry and display of integers.

FORMAT

CANCEL RADIX

PARAMETERS

None.

QUALIFIERS

/OVERRIDE

Cancels the override radix established by a previous SET RADIX/OVERRIDE command. This sets the current override radix to "none" and restores the output radix mode to the value established with a previous SET RADIX or SET RADIX/OUTPUT command. If you did not change the radix mode with a SET RADIX or SET RADIX/OUTPUT command, the CANCEL RADIX/OVERRIDE command restores the radix mode to its default value (hexadecimal for BLISS and MACRO, decimal for other languages).

DESCRIPTION

The CANCEL RADIX command cancels the effect of any previous SET RADIX and SET RADIX/OVERRIDE commands. It restores the input and output radix to their default value (hexadecimal for BLISS and MACRO, decimal for other languages).

The effect of the CANCEL RADIX/OVERRIDE command is more limited and is explained in the description of the /OVERRIDE qualifier.

Related commands: (SET, SHOW) RADIX.

EXAMPLE

```
DBG> CANCEL RADIX
```

This command restores the default input and output radix.

```
DBG> CANCEL RADIX/OVERRIDE
```

This command cancels any override radix you may have set with the SET RADIX/OVERRIDE command.

CANCEL SCOPE

Restores the default scope for symbol lookup.

FORMAT

CANCEL SCOPE

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The CANCEL SCOPE command cancels the current scope search list established by a previous SET SCOPE command and restores the default scope search list, namely *0,1,2, . . . ,N*, where *N* is the number of calls in the call stack.

The default scope means that, for a symbol without a path-name prefix, a symbol lookup such as "EXAMINE X" first looks for X in the routine that is currently executing (scope 0); if no X is visible there, the debugger looks in the caller of that routine (scope 1), and so on down the call stack; if X is not found in scope N, the debugger searches the rest of the run-time symbol table (RST), then searches the global symbol table (GST), if necessary.

Related commands: (SET, SHOW) SCOPE.

EXAMPLE

```
DBG> CANCEL SCOPE
```

This command cancels the current scope.

CANCEL SOURCE

Cancels a source directory search list established by a previous SET SOURCE command.

FORMAT CANCEL SOURCE

PARAMETERS *None.*

QUALIFIERS */EDIT*

Note: /EDIT applies mainly to Ada programs.

Cancels the effect of a previous SET SOURCE/EDIT command. As a result, when you use the EDIT command, the debugger searches for a source file in the same directory that it was in at compile time. The CANCEL SOURCE /EDIT command does not cancel the effect of a previous SET SOURCE command.

/MODULE=module-name

Cancels the effect of a previous SET SOURCE/MODULE=module-name command in which the same module name was specified. (*module-name* specifies a module for which a source directory search list is to be canceled). As a result, the debugger searches for the source file of the specified module in the same directory that it was in at compile time. The CANCEL SOURCE /MODULE=module-name command does not cancel the effect of a previous SET SOURCE command, or of a SET SOURCE/MODULE=module-name command in which a different module name was specified.

DESCRIPTION

When used without a qualifier, the CANCEL SOURCE command cancels the effect of a previous SET SOURCE command used without a qualifier. CANCEL SOURCE does not cancel the effect of a previous SET SOURCE /EDIT or SET SOURCE/MODULE=module-name commands.

See the qualifier descriptions for an explanation of their effects.

The /EDIT qualifier is needed when the files used for the display of source code are different from the files to be edited by means of the EDIT command. This is the case with Ada programs. For Ada programs, the (SET, SHOW, CANCEL) SOURCE commands affect the search of files used for source display (the "copied" source files in Ada program libraries); the (SET, SHOW, CANCEL) SOURCE/EDIT commands affect the search of the source files that you edit when using the EDIT command. If you use /MODULE with /EDIT, the effect of /EDIT is further qualified by /MODULE.

Related commands: (SET, SHOW) SOURCE, (SET, SHOW) MAX_SOURCE_FILES.

CANCEL SOURCE

EXAMPLE

```
DBG> SHOW SOURCE
source directory search list for COBOLTEST:
        SYSTEM::DEVICE:[PROJD]
        [014,015]
source directory search list for all other modules:
        [PROJA]
        [PROJB]
        [PETER.PROJC]
DBG> CANCEL SOURCE
DBG> SHOW SOURCE
source directory search list for COBOLTEST:
        SYSTEM::DEVICE:[PROJD]
        [014,015]
DBG> CANCEL SOURCE/MODULE=COBOLTEST
DBG> SHOW SOURCE
no directory search list in effect
```

The CANCEL SOURCE command cancels the effect of a previous SET SOURCE command. It does not cancel any source directory search lists for specific modules. But the CANCEL SOURCE/MODULE=module-name (in this case, COBOLTEST) cancels the source directory search list for that module.

CANCEL TRACE

Deletes tracepoints.

FORMAT

CANCEL TRACE *[address-expression[, . . .]]*

PARAMETERS

address-expression

Specifies a tracepoint to be canceled. Do not use the wildcard character (*). Do not specify an address expression when using any of the qualifiers except for /EVENT.

QUALIFIERS

/ALL

Cancels all tracepoints. When using /ALL, do not specify an address expression.

/BRANCH

Cancels the effect of a previous SET TRACE/BRANCH command. Do not specify an address expression with /BRANCH.

/CALL

Cancels the effect of a previous SET TRACE/CALL command. Do not specify an address expression with /CALL.

/EVENT=event-name

Note: /EVENT applies only to Ada and SCAN. See the VAX Ada and VAX SCAN documentation for complete information.

Cancels the effect of a previous SET TRACE/EVENT=event-name command. The effect of CANCEL TRACE/EVENT=event-name is symmetrical with the effect of SET TRACE/EVENT=event-name. To cancel a tracepoint, specify the event name and address expression (if any) exactly as you did with the SET TRACE/EVENT command, excluding any WHEN or DO clauses. Event names depend on the run-time facility and are identified in Appendix E for Ada and SCAN. You can also display the event names associated with the current run-time facility by issuing the SHOW EVENT_FACILITY command.

/EXCEPTION

Cancels the effect of a previous SET TRACE/EXCEPTION command. Do not specify an address expression with /EXCEPTION.

/INSTRUCTION

Cancels the effect of a previous SET TRACE/INSTRUCTION command. Do not specify an address expression with /INSTRUCTION.

CANCEL TRACE

/LINE

Cancels the effect of a previous SET TRACE/LINE command. Do not specify an address expression with /LINE.

DESCRIPTION

The effect of the CANCEL TRACE command is symmetrical with the effect of the SET TRACE command.

To cancel a tracepoint that was established at a specific location with the SET TRACE command, specify that same location with the CANCEL TRACE command. To cancel tracepoints that were established on a class of instructions or events by using a qualifier with the SET TRACE command (/CALL, /LINE, /EXCEPTION, /EVENT, and so on), specify that same qualifier with the CANCEL TRACE command.

Generally, you must specify either an address expression or a qualifier, but not both. The only exception is with the /EVENT qualifier, which requires that you specify an event name and lets you also specify an address expression for certain event names.

Note that the command CANCEL ALL also cancels all tracepoints.

Related commands: (SET, SHOW) TRACE, (SET, SHOW, CANCEL) BREAK, CANCEL ALL.

EXAMPLES

1
```
DBG> CANCEL TRACE MAIN\LOOP+10
```

This command cancels the tracepoint at the location MAIN\LOOP+10.

2
```
DBG> CANCEL TRACE/ALL
```

This command cancels all tracepoints you have set.

CANCEL TYPE/OVERRIDE

Cancels the override type established by a previous SET TYPE /OVERRIDE command.

FORMAT

CANCEL TYPE/OVERRIDE

PARAMETERS

None.

QUALIFIERS

/OVERRIDE

This qualifier must be specified.

DESCRIPTION

The CANCEL TYPE/OVERRIDE command sets the current override type to "none". As a result, a program location associated with a compiler-generated type will be interpreted according to that type.

Related commands: (SET, SHOW) TYPE/OVERRIDE.

EXAMPLE

```
DBG> CANCEL TYPE/OVERR
```

This command cancels the effect of a previous SET TYPE/OVERRIDE command.

CANCEL WATCH

Cancels watchpoints.

FORMAT

CANCEL WATCH *[address-expression[, . . .]]*

PARAMETERS

address-expression

Specifies a watchpoint to be canceled. With high-level languages, this is typically the name of a variable. Do not use the wildcard character (*). When using /ALL, do not specify an address expression.

QUALIFIERS

/ALL

Cancels all watchpoints. When using /ALL, do not specify an address expression.

DESCRIPTION

The effect of the CANCEL WATCH command is symmetrical with the effect of the SET WATCH command. To cancel a watchpoint that was established at a specific location with the SET WATCH command, specify that same location with the CANCEL WATCH command. Thus, to cancel a watchpoint that was set on an entire aggregate, specify the aggregate in the CANCEL WATCH command; to cancel a watchpoint that was set on one element of an aggregate, specify that element in the CANCEL WATCH command.

Generally, you must specify either an address expression or a qualifier, but not both. The only exception is with the /EVENT qualifier, which requires that you specify an event name and lets you also specify an address expression for certain event names.

Note that the CANCEL ALL command also cancels all watchpoints.

Related commands: (SET, SHOW) WATCH, (SET, SHOW, CANCEL) BREAK, (SET, SHOW, CANCEL) TRACE, CANCEL ALL.

EXAMPLES

```
1  DBG> CANCEL WATCH SUB2\TOTAL
```

This command cancels the watchpoint at variable TOTAL in module SUB2.

```
2  DBG> CANCEL WATCH/ALL
```

This command cancels all watchpoints you have set.

CANCEL WINDOW

Permanently deletes a screen window definition.

FORMAT

CANCEL WINDOW *[wname[, . . .]]*

PARAMETERS

wname

Specifies the name of a screen window definition to be canceled. Do not use the wildcard character (*). When using /ALL, do not specify a window definition name.

QUALIFIERS

/ALL

Cancels all predefined and user-defined window definitions. When using /ALL, do not specify a window definition name.

DESCRIPTION

When a window definition is canceled, you may no longer use its name in DISPLAY or SET DISPLAY commands. The command does not affect any displays.

Related commands: (SET, SHOW) WINDOW, (SET, SHOW, CANCEL) DISPLAY.

EXAMPLE

```
DBG> CANCEL WINDOW MIDDLE
```

This command permanently removes the screen window definition MIDDLE.

CTRL/C, CTRL/W, CTRL/Y, CTRL/Z

CTRL/Y interrupts a debugging session. CTRL/C is like CTRL/Y unless the program has a CTRL/C service routine. CTRL/Z ends a debugging session (like EXIT). CTRL/W refreshes the screen in screen mode (like DISPLAY/REFRESH).

FORMAT

CTRL/C

CTRL/W

CTRL/Y

CTRL/Z

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

For an explanation of the CTRL/W and CTRL/Z commands, see the descriptions of the DISPLAY/REFRESH and EXIT commands, respectively.

Unless the system or the user program has defined a CTRL/C service routine, CTRL/Y and CTRL/C have the same effect: the image is interrupted but unchanged, the terminal type-ahead buffer is purged, and the command interpreter receives control.

You can use the CTRL/Y command to (1) interrupt a debugging session or (2) interrupt an executing program in order to then invoke the debugger.

Interrupting a Debugging Session

Pressing CTRL/Y interrupts a debugging session and is useful when the program is executing an infinite loop that does not have a breakpoint, or when you want to interrupt a debugger command that takes a long time to complete. You are then at DCL command level. If you then type the DCL command DEBUG, control passes to the debugger: you return to the debugging session, but execution is suspended and the debugger prompt is displayed. You can then issue debugger commands.

If you type the DCL command CONTINUE after a CTRL/Y interrupt, you simply return to the debugging session at the same point in execution where you interrupted it.

Interrupting an Executing Program

Interrupting program execution with CTRL/Y is useful if your program is running without the debugger and you want to invoke the debugger.

To use this feature, you must, as a minimum, have linked your program with the /TRACE qualifier. To reference your program's symbols, you must have compiled and linked with the /DEBUG qualifier (in that case, you would use the DCL command RUN/NODEBUG to execute the program without the debugger).

Related commands: ($) DEBUG, ($) CONTINUE, EXIT, QUIT, DISPLAY /REFRESH.

EXAMPLE

```
DBG> GO
    .
    .
    .
DBG> CTRL/Y
Interrupt
$ DEBUG
DBG>
```

A debugging session is interrupted with CTRL/Y and resumed with the DCL command DEBUG. The debugger prompt indicates that debugger commands may now be entered.

DECLARE

Lets you pass parameters to command procedures

FORMAT

DECLARE *fname:pkind [,fname:pkind, . . .]*

PARAMETERS

fname

Specifies the formal parameter name. A formal parameter is associated with an actual parameter in the order in which the debugger processes the parameter declaration. If you specify several formal parameters on a single DECLARE command, the leftmost formal parameter is associated with the first actual parameter, the next formal parameter is associated with the second, and so on. If you use a DECLARE command in a loop, the formal parameter is associated with the first actual parameter on the first iteration of the loop; the same formal parameter is associated with the second actual parameter on the next iteration, and so on.

Do not specify a null parameter (represented either by two consecutive commas or by a comma at the end of the command).

pkind

Specifies the parameter type. Valid keywords are:

ADDRESS	Specifies an address expression. It has the same effect as a DEFINE/ADDRESS fname = . . . command.
VALUE	Specifies an expression in the current language. It has the same effect as a DEFINE/VALUE fname = . . . command.
COMMAND	Specifies a quoted character string. It has the same effect as a DEFINE/COMMAND fname = . . . command.

QUALIFIERS

None.

DESCRIPTION

The DECLARE command can be used only within a command procedure. The command associates actual parameters (those on the command line following the "@") with formal parameters (names inside the command procedure).

Each parameter declaration acts like a DEFINE command: it associates a parameter name with a value or other construct. The parameters themselves are consistent with those accepted by the DEFINE command and may in fact be removed from the symbol table with the DELETE command. For more information, see the descriptions of the DEFINE and DELETE commands.

Related commands: @file-spec, DEFINE, DELETE.

EXAMPLES

1

```
$ CREATE DISPLAY.COM
DECLARE P1:ADDRESS
EXAMINE P1:P1 + 100
    .
    .
    .
$ RUN PROG
DBG> @DISPLAY X
X
X+4
X+8
    .
    .
    .
X+100
```

The DECLARE command associates the parameter P1 with an address within the command procedure DISPLAY.COM. When DISPLAY.COM is invoked during the debugging session, address X is examined (as the P1 parameter) as specified by the EXAMINE P1:P1+100 command.

2

```
DBG> FOR I = 1 TO %PARCNT DO (DECLARE X:VALUE; EVALUATE X)
```

The DECLARE command is used in a loop. The built-in symbol %PARCNT specifies the number of actual parameters passed to the command procedure. The DECLARE command associates each parameter with a value and the EVALUATE command obtains that value.

DEFINE

Defines one or more symbols for the duration of the debugging session.

FORMAT

DEFINE *symbol=parameter[,symbol=parameter,...]*

PARAMETERS

symbol

Specifies the name of a symbol to be defined. Symbols can be composed of alphanumeric characters and underscores. The debugger converts lowercase alphabetic characters to uppercase. The first character must not be a number. The symbol must be no more than 31 characters long.

parameter

Depends on the qualifier specified.

QUALIFIERS

/ADDRESS

Specifies that the defined symbol is an abbreviation for an address expression. In this case, *parameter* is an address expression. DEFINE/ADDRESS is the default.

/COMMAND

Specifies that the defined symbol is to be treated as a new debugger command. In this case, *parameter* is a quoted character string. This qualifier provides, in simple cases, essentially the same capability as the DCL command "symbol:=string." To define complex commands, you may need to use command procedures with formal parameters. For more information on declaring parameters to command files (or procedures), see the description of the DECLARE command.

/LOCAL

Specifies that the definition remain local to the command procedure in which the DEFINE command is issued. The defined symbol is not visible at the debugger command level. By default, a symbol defined within a command procedure is visible outside that procedure.

/VALUE

Specifies that the defined symbol is an abbreviation for a value. In this case, *parameter* is a language expression in the current language.

DESCRIPTION

The DEFINE/ADDRESS command lets you define a symbol to refer to an address in your program. For example, you can define a symbol for a nonsymbolic program location or for a symbolic program location having a long path-name prefix. Then, you can refer to that program location by the newly defined symbol. /ADDRESS is the default definition qualifier.

The DEFINE/COMMAND command lets you define abbreviations for debugger commands or even define new commands, either from the debugger command level or from command procedures.

The DEFINE/VALUE command lets you assign a symbolic name to a value (or the result of evaluating a language expression).

Use the /LOCAL qualifier to confine symbol definitions to command procedures. By default, defined symbols are global (visible outside the command procedure).

If you plan to issue several DEFINE commands with the same qualifier, you can first use the SET DEFINE command to establish a new default qualifier (for example, SET DEFINE COMMAND makes the DEFINE command behave like DEFINE/COMMAND). Then you do not have to use that qualifier with the DEFINE command. You can override the current default qualifier for the duration of a single DEFINE command by specifying another qualifier.

In symbol translation, the debugger searches symbols you define during the debugging session first. So if you define a symbol that already exists in your program, the debugger translates the symbol according to its defined definition, unless you specify a path-name prefix.

When a symbol is redefined, the previous definition is canceled, even if different qualifiers were used with the DEFINE command. If you use the SET IMAGE command to establish a new current image, all definitions created with the DEFINE/ADDRESS and DEFINE/VALUE commands are deleted (definitions created with the DEFINE/COMMAND and DEFINE/KEY commands are retained, however).

Use the SHOW SYMBOL/DEFINED * command to determine the equivalence value of a symbol.

Use the DELETE command to cancel a symbol definition.

Related commands: SHOW DEFINE, SHOW SYMBOL/DEFINED, DELETE, SET IMAGE.

EXAMPLES

1
```
DBG> DEFINE CHK=MAIN\LOOP+10
```

This command assigns the symbol CHK to the address MAIN\LOOP+10.

2
```
DBG> DEFINE/VALUE COUNTER=0
DBG> SET TRACE/SILENT R DO (DEFINE/VALUE COUNTER = COUNTER+1)
```

The first command assigns a value of 0 to the symbol COUNTER. The second command tells the debugger to increment the value of the symbol COUNTER by 1 everytime address R is encountered. In other words, this example tells the debugger to count the number of calls to R.

3
```
DBG> DEFINE/COMMAND BRE = "SET BREAK"
```

This command assigns the symbol BRE to the debugger command SET BREAK.

DEFINE/KEY

Assigns a string to a function key.

FORMAT

DEFINE/KEY *key-name "equiv-string"*

PARAMETERS

key-name

Specifies a function key to be assigned a string. Valid key names are the following:

Key-name	LK201	VT100-type	VT52-type
PF1	PF1	PF1	Blue
PF2	PF2	PF2	Red
PF3	PF3	PF3	Black
PF4	PF4	PF4	
KP0, KP1, . . . ,KP9	Keypad 0, . . . ,9	Keypad 0, . . . ,9	Keypad 0, . . . ,9
PERIOD	Keypad period (.)	Keypad period (.)	
COMMA	Keypad comma (,)	Keypad comma (,)	
MINUS	Keypad minus (-)	Keypad minus (-)	
ENTER	ENTER	ENTER	ENTER
E1	Find		
E2	Insert Here		
E3	Remove		
E4	Select		
E5	Prev Screen		
E6	Next Screen		
HELP	Help		
DO	Do		
F6, F7, . . . , F20	F6, F7, . . . , F20		

equiv-string

Specifies the string to be processed when the specified key is pressed. Typically, this is one or more debugger commands. If the string includes any spaces or non-alphanumeric characters (for example, a semicolon separating two commands) enclose the string in quotation marks.

QUALIFIERS

/[NO]ECHO

Controls whether the command line is displayed after the key has been pressed. The default is /ECHO. Do not use /NOECHO with /NOTERMINATE.

/[NO]IF_STATE=(state-name[, . . .])

Specifies one or more states to which a key definition applies. /IF_STATE assigns the key definition to the specified states. You may specify predefined states, such as DEFAULT and GOLD, or user-defined states. A state name can be any appropriate alphanumeric string. /NOIF_STATE (default) assigns the key definition to the current state.

/[NO]LOCK_STATE

Controls how long the state set by /SET_STATE remains in effect after the specified key is pressed. /LOCK_STATE causes the state to remain in effect until it is changed explicitly (for example, with a SET KEY/STATE command). /NOLOCK_STATE (default) causes the state to remain in effect only until the next terminator character is typed, or until the next defined function key is pressed.

/[NO]LOG

Controls whether a message is displayed indicating that the key definition has been successfully created. /LOG (default) displays the message.

/[NO]SET_STATE=state-name

Controls whether pressing the key changes the current key state. /SET_STATE causes the current state to change to the specified state when you press the key. /NOSET_STATE (default) causes the current state to remain in effect.

/[NO]TERMINATE

Controls whether the specified string is to be terminated (processed) when the key is pressed. /TERMINATE causes the string to be terminated when the key is pressed. /NOTERMINATE (default) allows you to press other keys before terminating the string by pressing the RETURN key.

DESCRIPTION

Keypad mode must be enabled (SET MODE KEYPAD) before you can use this command. Keypad mode is enabled by default.

The DEFINE/KEY command lets you assign a string to a function key, overriding any predefined function that was bound to that key (the predefined key functions are listed in Appendix B). When you then press the key, the debugger enters the currently associated string into your command line. The DEFINE/KEY command is like the DCL DEFINE/KEY command.

On VT52 and VT100-series terminals, the function keys you can use include all of the numeric keypad keys. VT-200 series terminals and MicroVAX workstations have the LK201 keyboard. On LK201 keyboards, the function keys you can use include all of the numeric keypad keys, the non-arrow keys of the editing keypad (Find, Insert Here, and so on), and keys F6 through F20 at the top of the keyboard.

A key definition remains in effect until you redefine the key, issue the DELETE/KEY command for that key, or exit the debugger. You can include key definitions in a command procedure, such as your debugger initialization file.

The /IF_STATE qualifier lets you increase the number of key definitions available on your terminal. The same key can be assigned any number of definitions as long as each definition is associated with a different state.

By default, the current key state is the "DEFAULT" state. The current state may be changed with the SET KEY/STATE command, or by pressing a key that causes a state change (a key that was defined with the DEFINE/KEY /LOCK_STATE/STATE qualifier combination).

Related commands: DELETE/KEY, SHOW KEY, SET KEY.

EXAMPLES

1
```
DBG> SET KEY/STATE=GOLD
%DEBUG-I-SETKEY, keypad state has been set to GOLD
DBG> DEFINE/KEY/TERMINATE KP9 "SET RADIX/OVERRIDE HEX"
%DEBUG-I-DEFKEY, GOLD key KP9 has been defined
```

The SET KEY command establishes GOLD as the current key state. The DEFINE/KEY command assigns the SET RADIX/OVERRIDE HEX command to keypad key 9 for the current state (GOLD). The command is processed when key is pressed.

2
```
DBG> DEFINE/KEY/IF_STATE=BLUE KP9 "SET BREAK %LINE "
%DEBUG-I-DEFKEY, BLUE key KP9 has been defined
```

This command assigns the unterminated command string "SET BREAK %LINE " to keypad key 9, for the BLUE state. After pressing keypad key 9, you can enter a line number and then press the RETURN key to process the SET BREAK command.

3
```
DBG> SET KEY/STATE=DEFAULT
%DEBUG-I-SETKEY, keypad state has been set to DEFAULT
DBG> DEFINE/KEY/SET_STATE=RED/LOCK_STATE F12 ""
%DEBUG-I-DEFKEY, DEFAULT key F12 has been defined
```

The SET KEY command establishes DEFAULT as the current state. The DEFINE/KEY command makes key F12 (LK201 keyboard) a state key. Pressing F12 while in the DEFAULT state causes the current state to become RED. The key definition is not terminated and has no other effect (a null string is assigned to F12). After pressing F12, you can issue "RED" commands by pressing keys that have definitions associated with the RED state.

DELETE

Deletes a symbol definition from the DEFINE symbol table.

FORMAT

DELETE *[symbol-name[, . . .]]*

PARAMETERS

symbol-name

Specifies a symbol whose definition is to be deleted from the DEFINE symbol table. Do not use the wildcard character (*). When using /ALL, do not specify a symbol name. If you use /LOCAL, the symbol specified must have been previously defined with the DEFINE/LOCAL command. If you do not specify /LOCAL, the symbol specified must have been previously defined with the DEFINE command without the /LOCAL qualifier.

QUALIFIERS

/ALL

Deletes all global DEFINE definitions. If you also specify /LOCAL, deletes all local DEFINE definitions associated with the current command procedure (but not the global DEFINE definitions). When using /ALL, do not specify a symbol name.

/LOCAL

Deletes the (local) definition of the specified symbol from the current command procedure. The symbol must have been previously defined with the DEFINE/LOCAL command.

DESCRIPTION

The DELETE command deletes either a global DEFINE symbol or a local DEFINE symbol. A global DEFINE symbol is a symbol defined with the DEFINE command without the /LOCAL qualifier. A local DEFINE symbol is a symbol defined in a debugger command procedure with the DEFINE /LOCAL command, so that its definition is confined to that command procedure. The DELETE command is identical to the UNDEFINE command.

Related commands: DEFINE.

EXAMPLE

```
DBG> DEFINE X = INARR, Y = OUTARR
DBG> DELETE X,Y
```

The DEFINE command defines X and Y as global *symbols corresponding to* INARR and OUTARR, respectively. The DELETE command deletes these two symbol definitions from the global symbol table.

```
DBG> DELETE/ALL/LOCAL
```

The DELETE/ALL/LOCAL commmand deletes all local symbol definitions from the current command procedure.

DELETE/KEY

Deletes key definitions that have been established with the DEFINE /KEY command.

FORMAT

DELETE/KEY *[key-name]*

PARAMETERS

key-name

Specifies a key whose definition is to be deleted. Do not use the wildcard character (*). When using /ALL, do not specify a key name. Valid key names are:

Key-name	LK201	VT100-type	VT52-type
PF1	PF1	PF1	Blue
PF2	PF2	PF2	Red
PF3	PF3	PF3	Black
PF4	PF4	PF4	
KP0, KP1, . . . ,KP9	Keypad 0, . . . ,9	Keypad 0, . . . ,9	Keypad 0, . . . ,9
PERIOD	Keypad period (.)	Keypad period (.)	
COMMA	Keypad comma (,)	Keypad comma (,)	
MINUS	Keypad minus (-)	Keypad minus (-)	
ENTER	ENTER	ENTER	ENTER
E1	Find		
E2	Insert Here		
E3	Remove		
E4	Select		
E5	Prev Screen		
E6	Next Screen		
HELP	Help		
DO	Do		
F6, F7, . . . , F20	F6, F7, . . . , F20		

QUALIFIERS

/ALL

Deletes all key definitions in the specified state. When using /ALL, do not specify a key name. If you do not specify a state, all key definitions in the current state are deleted. Use the /STATE qualifier to specify one or more states.

/[NO]LOG

Controls whether a message is displayed indicating that the specified key definitions have been deleted. /LOG (default) displays the message.

/[NO]STATE=(state-name [, . . .])

Selects one or more states for which a key definition is to be deleted. /STATE deletes key definitions for the specified states. You may specify predefined key states, such as DEFAULT and GOLD, or user-defined states. A state name can be any appropriate alphanumeric string. /NOSTATE (default) deletes the key definition for the current state only.

By default, the current key state is the "DEFAULT" state. The current state may be changed with the SET KEY/STATE command, or by pressing a key that causes a state change (a key that was defined with the DEFINE/KEY /LOCK_STATE/STATE qualifier combination).

DESCRIPTION

Keypad mode must be enabled (SET MODE KEYPAD) before you can use this command. Keypad mode is enabled by default.

The DELETE/KEY command is like the DCL DELETE/KEY command and has the same effect as the (debugger) UNDEFINE/KEY command.

Related commands: DEFINE/KEY, SHOW KEY, SET KEY.

EXAMPLES

1
```
DBG> DELETE/KEY KP4
%DEBUG-I-DELKEY, DEFAULT key KP4 has been deleted
```

This command deletes the key definition for keypad key KP4 in the state last set by the SET KEY command (by default, this is the DEFAULT state).

2
```
DBG> DELETE/KEY/STATE=(BLUE,RED) COMMA
%DEBUG-I-DELKEY, BLUE key COMMA has been deleted
%DEBUG-I-DELKEY, RED key COMMA has been deleted
```

This command deletes the key definition for keypad key COMMA in the BLUE and RED states.

DEPOSIT

Changes the value of a variable or the contents of a program location.

FORMAT

DEPOSIT *address-expression = language-expression*

PARAMETERS

address-expression

Specifies the location into which the value of the language expression is to be deposited. With high-level languages, *address-expression* is typically the name of a variable.

language-expression

Specifies the value to be deposited. You may specify any source language expression in the currently set language.

When the DEPOSIT command is executed, the expression is evaluated in the syntax of the source language. The value of the expression is then converted to the type associated with the address expression and placed at the location denoted by the address expression.

If the expression is an ASCII string or a VAX/VMS assembly-language instruction, you must enclose it in quotation marks or apostrophes.

QUALIFIERS

/ASCIC

Deposits a counted ASCII string with a 1-byte count into the specified location. This is an ASCII string preceded by a 1-byte count field that gives the length of the string. /AC is also accepted.

/ASCID

Deposits an ASCII string into the address given by a string descriptor that is at the specified location. The expression on the right-hand side of the equal sign must be a string. The specified location must contain a string descriptor. If the string lengths do not match, the string is either truncated on the right or padded with blanks on the right. /AD is also accepted.

/ASCII:n

Deposits *n* bytes of a string into the specified location. The expression on the right-hand side of the equal sign must be a string. If its length is not *n*, the string is truncated or padded with blanks on the right. If *n* is omitted, the actual length of the data item at the specified location is used.

/ASCIW

Deposits a counted ASCII string with a 1-word count into the specified location. This is an ASCII string preceded by a 2-byte count field that gives the length of the string. /AW is also accepted.

/ASCIZ

Deposits a zero-terminated ASCII string into the specified location. The expression on the right-hand side of the equal sign must be a string. The string is deposited into the specified location followed by a zero byte indicating the end of the string. /AZ is also accepted.

/BYTE

Deposits a 1-byte integer into the specified location.

/D_FLOAT

Converts the expression on the right-hand side of the equal sign to the D_floating type (length 8 bytes) and deposits the result into the specified location. Values of type D_floating may range from $.29*10^{-38}$ to $1.7*10^{38}$ with approximately 16 decimal digits precision.

/DATE_TIME

Converts a string representing a date and time (for example, 31-MAR-1985 21:08:47.15) to the VAX/VMS internal format for date and time and deposits that quadword into the specified location.

/FLOAT

Converts the expression on the right-hand side of the equal sign to the F_floating type (length 4 bytes) and deposits the result into the specified location. Values of type F_floating may range from $.29*10^{-38}$ to $1.7*10^{38}$ with approximately 7 decimal digits precision.

/G_FLOAT

Converts the expression on the right-hand side of the equal sign to the G_floating type (length 8 bytes) and deposits the result into the specified location. Values of type G_floating may range from $.56*10^{-308}$ to $.9*10^{308}$ with approximately 15 decimal digits precision.

/H_FLOAT

Converts the expression on the right-hand side of the equal sign to the H_floating type (length 16 bytes) and deposits the result into the specified location. Values of type H_floating may range from $.84*10^{-4932}$ to $.59*10^{4932}$ with approximately 33 decimal digits precision.

/INSTRUCTION

Deposits a VAX/VMS assembly-language instruction into the specified location. The expression on the right-hand side of the equal sign must be a string representing a VAX/VMS instruction.

/LONGWORD

Deposits a longword integer (length 4 bytes) into the specified location.

/OCTAWORD

Deposits an octaword integer (length 16 bytes) into the specified location.

/PACKED:n
Converts the expression on the right-hand side of the equal sign to a packed decimal representation (length *n* nibbles) and deposits the resulting value into the specified location.

/QUADWORD
Deposits a quadword integer (length 8 bytes) into the specified location.

/TASK

Note: /TASK applies only to Ada programs.

Deposits an Ada task value (a task name, or a task ID such as %TASK 3) into the specified location.

/TYPE=(type-expression)
Converts the expression to be deposited to the type denoted by *type-expression* (the name of a variable or data type), then deposits the resulting value into the specified location.

/WORD
Deposits a word integer (length 2 bytes) into the specified location.

DESCRIPTION

In general, the DEPOSIT command may be used to change the contents of any memory location in your program. Typically, the command is used to change the value of a program variable.

The DEPOSIT command is like an assignment statement in most programming languages. The value specified to the right of the equal sign is deposited into the location specified to the left of the equal sign (note that for Ada and PASCAL, you can use ":=" instead of "=" in the command syntax). Type conversion, if necessary, is done according to the rules of the currently set language.

The debugger knows the compiler-generated type of a variable (whether it is an integer, real, string, array, record, and so on), as well as any dimensional constraints that apply to that variable. When you use the DEPOSIT command, the debugger checks that the value assigned to a variable is consistent with the data type and dimensional constraints of the variable.

By using qualifiers with the DEPOSIT command, you can override the type associated with a program location in order to deposit data of a different type.

The DEPOSIT command sets the *current entity* built-in symbols %CURLOC and period (.) to the location denoted by the address expression specified. Logical predecessors (%PREVLOC and circumflex (^)) and successors (%NEXTLOC and pressing the RETURN key) are based on the value of the current entity symbol.

Related commands: EXAMINE, EVALUATE, (SET, SHOW, CANCEL) RADIX, (SET, SHOW) TYPE, CANCEL TYPE/OVERRIDE.

EXAMPLES

1
```
DBG> DEPOSIT I = 7
```

This command deposits the value 7 into the integer variable I.

2
```
DBG> DEPOSIT WIDTH = CURRENT_WIDTH + 24.80
```

This command deposits the value of the expression CURRENT_WIDTH + 24.80 into the real variable WIDTH.

3
```
DBG> DEPOSIT STATUS = FALSE
```

This command deposits the value FALSE into the boolean variable STATUS.

4
```
DBG> DEPOSIT PART_NUMBER = "WG-7619.3-84"
```

This command deposits the string WG-7619.3-84 into the string variable PART_NUMBER.

5
```
DBG> DEPOSIT EMPLOYEE.ZIPCODE = 02172
```

This command deposits the value 02172 into component ZIPCODE of record EMPLOYEE.

6
```
DBG> DEPOSIT ARR(8) = 35
DBG> DEPOSIT ^ = 14
```

The first DEPOSIT command deposits the value 35 into element 8 of array ARR. As a result, element 8 becomes the current entity. The second command deposits the value 14 into the logical predecessor of element 8, namely element 7.

7
```
DBG> FOR I = 1 TO 4 DO (DEPOSIT ARR(I) = 0)
```

This command deposits the value 0 into elements 1 through 4 of array ARR.

8
```
DBG> DEPOSIT COLOR = 3
%DEBUG-E-OPTNOTALLOW, operator "DEPOSIT" not allowed on given data type
```

The debugger alerts you when you try to deposit data of the wrong type (in this case, an integer) into a variable. The E (error) message severity indicates that the debugger does not make the assignment.

9
```
DBG> DEPOSIT VOLUME = - 100
%DEBUG-I-IVALOUTBNDS, value assigned is out of bounds at or near '-'
```

The debugger alerts you when you try to deposit an out-of-bounds value into a variable (in this case a negative value). The I (informational) message severity indicates that the debugger does make the assignment.

10
```
DBG> DEPOSIT/BYTE WORK = %HEX 212
```

This command deposits the expression %HEX 212 *into location* WORK and converts it to a byte integer.

11
```
DBG> DEPOSIT/OCTAWORD  BIGINT = 111222333444555
```

This command deposits the expression 111222333444555 into location BIGINT and converts it to an octaword integer.

```
12  DBG> DEPOSIT/FLOAT BIGFLT = 1.11949*10**35
```

This command converts 1.11949*10**35 to an F_floating type value and deposits it into location BIGFLT.

```
13  DBG> DEPOSIT/ASCII:10 WORK+20 = 'abcdefghij'
```

This command deposits the string abcdefghij into location WORK+20.

```
14  DBG> DEPOSIT/INSTR  SUB2+2 = 'MOVL   #20A,R0'
```

This command deposits the instruction MOVL #20A,R0' into location SUB2+2.

```
15  DBG> DEPOSIT/TASK VAR = %TASK 2
    DBG> EXAMINE/HEX VAR
    SAMPLE.VAR: 0016A040
    DBG> EXAMINE/TASK VAR
    SAMPLE.VAR: %TASK 2
```

The DEPOSIT command deposits the Ada task value %TASK 2 into location VAR. The subsequent EXAMINE commands display the contents of VAR in hexadecimal format and as a task value, respectively.

DISABLE AST

Disables the delivery of ASTs (asynchronous system traps) in your program.

FORMAT

DISABLE AST

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The DISABLE AST command prevents interrupts from occurring while the debugger is running. Note that this does not prevent interrupts from occurring while the program is running. The ENABLE AST command re-enables the delivery of ASTs, including any pending ASTs (ASTs waiting to be delivered).

Related commands: (ENABLE, SHOW) AST.

EXAMPLE

```
DBG> DISABLE AST
DBG> SHOW AST
ASTs are disabled
```

The DISABLE AST disables the delivery of ASTs, as confirmed with the SHOW AST command.

DISPLAY

Modifies an existing screen display.

FORMAT

DISPLAY *[disp-name [AT wspec] [dkind]] [, . . .]*

PARAMETERS

disp-name

Specifies a screen display to be displayed. You may specify any of the following:

- A predefined display: SRC, OUT, PROMPT, INST, REG
- A display previously created with the SET DISPLAY command
- A pseudo-display name: %CURDISP, %CURSCROLL, %NEXTDISP, %NEXTINST, %NEXTOUTPUT, %NEXTSCROLL, %NEXTSOURCE

You must specify this parameter unless you use /GENERATE (parameter optional), or /REFRESH (parameter not allowed).

You may specify more than one display, each with an optional window specification (wspec) and display kind (dkind).

wspec

Specifies the screen window at which the display is to be positioned if you want to change the position. You may specify any of the following:

- A predefined window. For example, RH1 (right top half). See Appendix C.
- A window definition previously established with the SET WINDOW command.
- A window specification of the form *(start-line, line-count [,start-column, column-count])*. The specification can include expressions which may be based on the built-in symbols %PAGE and %WIDTH (for example, %WIDTH/4).

If you omit the *wspec* parameter, the screen position of the display is not changed.

dkind

Specifies the new display kind if you want to change the kind of display. Valid keywords are the following:

DO (cmd-list)	Specifies an automatically updated output display. The commands in cmd-list are executed each time the debugger gains control. Their output forms the contents of the display.
INSTRUCTION	Specifies an instruction display. If selected as the current instruction display with the SELECT /INSTRUCTION command, it will display the output from subsequent EXAMINE/INSTRUCTION commands.
INSTRUCTION (command)	Specifies an automatically updated instruction display. The command specified must be an EXAMINE/INSTRUCTION command. The instruction display is updated each time the debugger gains control.
OUTPUT	Specifies an output display. If selected as the current output display with the SELECT/OUTPUT command, it will display any debugger output that is not directed to another display. If selected as the current input display with the SELECT /INPUT command, it will echo debugger input. If selected as the current error display with the SELECT/ERROR command, it will display debugger diagnostic messages.
REGISTER	Specifies an automatically updated register display. The display is updated each time the debugger gains control.
SOURCE	Specifies a source display. If selected as the current source display with the SELECT /SOURCE command, it will display the output from subsequent TYPE or EXAMINE/SOURCE commands.
SOURCE (command)	Specifies an automatically updated source display. The command specified must be a TYPE or EXAMINE/SOURCE command. The source display is updated each time the debugger gains control.

You cannot change the display kind of the PROMPT display.

QUALIFIERS

/CLEAR

Erases the entire contents of a specified display.

/[NO]DYNAMIC

Controls whether a display automatically adjusts its window dimensions proportionally when a SET TERMINAL command is issued. By default (/DYNAMIC), all user-defined and predefined displays, except register displays, adjust their dimensions automatically. A register display maintains its window dimensions and location when the screen height or width are changed.

/GENERATE

Regenerates the contents of a specified display. Only automatically generated displays are regenerated. These include DO displays, register displays, source (cmd-list) displays, and instruction (cmd-list) displays. The debugger automatically regenerates all these kinds of displays before each prompt. If no display is specified, regenerates the contents of all automatically generated displays.

/HIDE

Places a specified display at the bottom of the display pasteboard. This makes visible any display previously hidden by the specified display. It also hides the specified display behind any other displays that share the same region of the screen. You cannot hide the PROMPT display.

/HIDE has the same effect as /PUSH.

/[NO]MARK_CHANGE

Controls whether the lines that change in a DO display each time it is automatically updated are marked. When you use /MARK_CHANGE, any lines in which some contents have changed since the last time the display was updated are highlighted in reverse video. This qualifier is particularly useful when you want any variables in an automatically updated display to be highlighted when they change.

/NOMARK_CHANGE (default) specifies that any lines that change in DO displays are not to be marked. This qualifier cancels the effect of a previously issued /MARK_CHANGE qualifier on the specified display.

This qualifier is not applicable to other kinds of displays.

/[NO]POP

Controls whether a specified display is placed at the top of the display pasteboard, ahead of any other displays but behind the PROMPT display. By default (/POP), the display is placed at the top of the pasteboard and hides any other displays that share the same region of the screen, except for the PROMPT display. This is the default action of the DISPLAY command.

/NOPOP preserves the order of all displays on the pasteboard (same effect as /NOPUSH).

/[NO]PUSH

/PUSH has the same effect as /HIDE. /NOPUSH preserves the order of all displays on the pasteboard (same effect as /NOPOP).

/REFRESH

Refreshes the terminal screen. If you use this qualifier, do not use any command parameters.

/REMOVE

Marks the display as being removed from the display pasteboard, so it will not be shown on the screen unless you explicitly request it with another DISPLAY command. Although a removed display is not visible on the screen, it still exists and its contents are preserved. You cannot remove the PROMPT display.

/SIZE:n

Changes the maximum size of a display to *n* lines. If more than *n* lines are written to the display, the oldest lines are lost as the new lines are added. If you omit this qualifier, the maximum size is not changed.

For an output or DO display, /SIZE:n specifies that the display should hold the *n* most recent lines of output. For a source or instruction display, *n* gives the number of source lines or lines of instructions that can be placed in the memory buffer at any one time. However, you can scroll a source display over the entire source code of the module whose code is displayed (source lines are paged into the buffer as needed). Similarly, you can scroll an instruction display over all of the instructions of the routine whose instructions are displayed (instructions are decoded from the image as needed).

DESCRIPTION

The DISPLAY command performs a variety of functions. Its major function is to show the display you have requested. The display is placed on top of the display pasteboard, ahead of the other displays but behind the PROMPT display, which cannot be hidden. The specified display thus becomes visible, and the portions of any displays that share the same region of the screen are hidden (although these displays still exist).

With certain qualifiers, you can use this command to remove displays from the terminal screen or to refresh the entire screen. You can also use this command to change the display's screen window, to change its maximum size in lines, or to change its kind or debug command list.

See Appendix B for keypad-key definitions associated with the DISPLAY command.

Related commands: (SET, SHOW, CANCEL) DISPLAY, (SET, SHOW, CANCEL) WINDOW, SELECT, EXPAND, MOVE, (SET, SHOW) TERMINAL.

EXAMPLES

1

```
DBG> DISPLAY REG
```

This command shows the predefined register display, REG, at its current window location.

2

```
DBG> DISPLAY NEWDISP AT RT2
DBG> SELECT/INPUT NEWDISP
```

The DISPLAY command shows the user-defined display NEWDISP at the right middle third of the screen. The SELECT/INPUT command selects NEWDISP as the current input display. NEWDISP will echo debugger input.

EDIT

Invokes the editor established with the SET EDITOR command. If no SET EDITOR command was issued, invokes the VAX Language-Sensitive Editor, if that editor is installed on your system.

FORMAT

EDIT *[[module-name\] line-number]*

PARAMETERS

module-name

Specifies the name of the module whose source file is to be edited. If you specify a module name, you must also specify a line number. If you omit the module name parameter, the source file whose code appears in the current source display is chosen for editing.

line-number

A positive integer that specifies the source line on which the editor's cursor is to be initially placed. If you omit this parameter, the cursor is initially positioned at the start of the source line that is centered in the debugger's current source display, or at the start of line 1 if the editor was set to /NOSTART_POSITION (see the SET EDITOR command description).

QUALIFIERS

/[NO]EXIT

Controls whether you end the debugging session prior to invoking the debugger. If you specify /EXIT, the debugging session is terminated and the editor is then invoked. If you specify /NOEXIT (default), the editing session is spawned in a subprocess and you return to your debugging session after exiting from the editor.

DESCRIPTION

If you have not specified an editor with the SET EDITOR command, the EDIT command invokes the VAX Language-Sensitive Editor in a spawned subprocess (if the VAX Language-Sensitive Editor is installed on your system). The typical (default) way to use the EDIT command is not to specify any parameters. In this case, the editing cursor is initially positioned at the start of the line that is centered in the currently selected debugger source display (the current source display).

The SET EDITOR command provides options for invoking different editors, either in a subprocess or through a callable interface.

Related commands: (SET, SHOW) EDITOR, (SET, SHOW, CANCEL) SOURCE.

EXAMPLES

1
```
DBG> EDIT
```

The EDIT command spawns the VAX Language-Sensitive Editor in a subprocess to edit the source file whose code appears in the current source display. The editing cursor will be positioned at the start of the line that was centered in the source display.

2
```
DBG> EDIT SWAP\12
```

The EDIT command spawns the VAX Language-Sensitive Editor in a subprocess to edit the source file containing the module SWAP. The editing cursor will be positioned at the start of source line 12.

3
```
DBG> SET EDITOR/CALLABLE_EDT
DBG> EDIT
```

The SET EDITOR/CALLABLE_EDT command establishes that EDT is the default editor and is invoked through its callable interface (rather than spawned in a subprocess). The EDIT command invokes EDT to edit the source file whose code appears in the current source display. The editing cursor will be positioned at the start of source line 1, because the default qualifier /NOSTART_POSITION applies to EDT.

ENABLE AST

Enables the delivery of asynchronous system traps (ASTs) in your program.

FORMAT

ENABLE AST

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The ENABLE AST command enables the delivery of asynchronous system traps (ASTs) while the debugger is running, including any pending ASTs (ASTs waiting to be delivered). Delivery of ASTs is initially enabled by default.

Related commands: (DISABLE, SHOW) AST.

EXAMPLE

```
DBG> ENABLE AST
DBG> SHOW AST
ASTs are enabled
```

The ENABLE AST command enables the delivery of ASTs, as confirmed with the SHOW AST command.

EVALUATE

Evaluates a language expression in the currently set language.

FORMAT

EVALUATE *language-expression[, . . .]*

PARAMETERS

expression

Specifies any valid expression in the source language.

QUALIFIERS

/CONDITION_VALUE

Specifies that the expression be interpreted as a VAX/VMS condition value (the kind of condition value you would specify using the condition-handling mechanism). The message text corresponding to that condition value is then displayed. The specified value must be an integer value.

/BINARY

Specifies that the result be displayed in binary radix.

/DECIMAL

Specifies that the result be displayed in decimal radix.

/HEXADECIMAL

Specifies that the result be displayed in hexadecimal radix.

/OCTAL

Specifies that the result be displayed in octal radix.

DESCRIPTION

The debugger interprets the parameter specified in an EVALUATE command as a source-language expression, evaluates it in the syntax of the source language, and displays its value as a literal in the source language.

If an expression contains symbols with different compiler-generated types, the debugger uses the type-conversion rules of the current language to evaluate the expression.

If you specify a radix command qualifier, the debugger displays the value of the expression as a literal in that radix. A radix command qualifier does not, however, affect how the debugger interprets a literal.

Debugger support for language-specific operators and constructs is described in Appendix E.

Related commands: EVALUATE/ADDRESS, (SET, SHOW, CANCEL) RADIX, (SET, SHOW, CANCEL) TYPE, SHOW SYMBOL.

EXAMPLES

1
```
DBG> EVALUATE 100.34 * (14.2 + 7.9)
2217.514
```

This command uses the debugger as a calculator by multiplying 100.34 by (14.2 + 7.9).

2
```
DBG> EVALUATE/OCTAL X
00000001512
```

This command evaluates the symbol X and displays the result in octal radix.

3
```
DBG> EVALUATE TOTAL + CURR_AMOUNT
8247.20
```

This command evaluates the sum of the values of two real variables, TOTAL and CURR_AMOUNT.

4
```
DBG> DEPOSIT WILLING = TRUE
DBG> DEPOSIT ABLE = FALSE
DBG> EVALUATE WILLING AND ABLE
False
```

The EVALUATE command evaluates the logical AND of the current values of two boolean variables, WILLING and ABLE.

5
```
DBG> EVALUATE COLOR'FIRST
RED
```

(Ada example). This command evaluates the first element of the enumeration type COLOR.

EVALUATE/ADDRESS

Evaluates an address expression and displays the result as a virtual memory address.

FORMAT

EVALUATE/ADDRESS *address-expression[, . . .]*

PARAMETERS

address-expression

Specifies an address expression of any valid form (for example, a routine name, a variable name, a label, a line number, and so on).

QUALIFIERS

/BINARY

Specifies that the result be displayed in binary radix.

/DECIMAL

Specifies that the result be displayed in decimal radix.

/HEXADECIMAL

Specifies that the result be displayed in hexadecimal radix.

/OCTAL

Specifies that the result be displayed in octal radix.

DESCRIPTION

The EVALUATE/ADDRESS command lets you determine the virtual address designated by an address expression.

By default, the address is displayed in hexadecimal radix for BLISS and MACRO and decimal radix for other languages. You can use a radix command qualifier to display address values in some other radix.

If a variable is stored in a register instead of virtual memory, the EVALUATE /ADDRESS command returns the name of the register.

Related commands: EVALUATE, (SET, SHOW, CANCEL) RADIX, SHOW SYMBOL/ADDRESS.

EXAMPLES

```
1  DBG> EVALUATE/ADDRESS MODNAME\%LINE 110
   3942
```

This command determines the value of the address expression MODNAME\%LINE 110.

```
2  DBG> EVALUATE/ADDRESS/HEX TOTAL
   0000020E
```

This command determines the value of the address expression TOTAL and displays the result in hexadecimal radix.

EVALUATE/ADDRESS

3
```
DBG> EVALUATE/ADDRESS/HEX A,B,C
000004A4
000004AC
000004A0
```

This command determines the values of the address expressions A, B, and C and displays these values in hexadecimal radix.

EXAMINE

Displays the current value of a variable.

FORMAT

EXAMINE *[address-expression[:address-expression]] [, . . .]*

PARAMETERS

address-expression

Specifies an entity to be examined. This is typically a variable name, including the name of an aggregate (array or record). More generally, an address expression may be composed of numbers and symbols, as well as one or more operators, operands, or delimiters.

If a range of entities is to be examined, the value of the address expression that denotes the first entity in the range must be less than the value of the address expression that denotes the last entity in the range.

QUALIFIERS

/ASCIC

Interprets each examined entity as a counted ASCII string with a 1-byte count field. This is an ASCII string preceded by a 1-byte count field that gives the length of the string to be displayed. /AC is also accepted.

/ASCID

Interprets each examined entity as the address of a string descriptor pointing to an ASCII string. The CLASS and DTYPE fields of the descriptor are not checked, but the LENGTH and POINTER fields provide the character length and address of the ASCII string. The string is then displayed. /AD is also accepted.

/ASCII:n

Interprets and displays each examined entity as an ASCII string of length *n* bytes (*n* characters). If *n* is omitted, the debugger attempts to determine a length from the type of the address expression.

/ASCIW

Interprets each examined entity as a counted ASCII string with a 1-word count field. This is an ASCII string preceded by a 2-byte count field that gives the length of the string to be displayed. /AW is also accepted.

/ASCIZ

Interprets and displays each examined entity as a zero-terminated ASCII string. This is an ASCII string followed by a zero byte, to indicate the end of the string. /AZ is also accepted.

/BINARY

Displays each examined entity as a binary integer.

/BYTE
Displays each examined entity in the byte integer type (length 1 byte).

/CONDITION_VALUE
Interprets each examined entity (usually register R0 in this case) as a condition-value return status and displays the message associated with that return status.

/D_FLOAT
Displays each examined entity in the D_floating type (length 8 bytes). Values of type D_floating may range from $.29*10^{-38}$ to $1.7*10^{38}$ with approximately 16 decimal digits precision.

/DATE_TIME
Interprets each examined entity as a quadword integer (64 bits) containing the internal VAX/VMS representation of date-time. Displays the value in the format *dd-mmm-yyyy hh:mm:ss.xx*.

/DECIMAL
Displays each examined entity as a decimal integer.

/DEFAULT
Displays each examined entity in the default radix.

/FLOAT
Displays each examined entity in the F_floating type (length 4 bytes). Values of type F_floating may range from $.29*10^{-38}$ to $1.7*10^{38}$ with approximately 7 decimal digits precision.

/G_FLOAT
Displays each examined entity in the G_floating type (length 8 bytes). Values of type G_floating may range from $.56*10^{-308}$ to $.9*10^{308}$ with approximately 15 decimal digits precision.

/H_FLOAT
Displays each examined entity in the H_floating type (length 16 bytes). Values of type H_floating may range from $.84*10^{-4932}$ to $.59*10^{4932}$ with approximately 33 decimal digits precision.

/HEXADECIMAL
Displays each examined entity as a hexadecimal integer.

/INSTRUCTION
Displays each examined entity as a VAX assembly-language instruction (variable length).

/[NO]LINE
Controls whether code locations are displayed in terms of line numbers (%LINE x) or in terms of routine + byte-offset. By default (/LINE), the debugger symbolizes code locations in terms of line numbers.

/LONGWORD
Displays each examined entity in the longword integer type (length 4 bytes).

/OCTAL
Displays each examined entity as an octal integer.

/OCTAWORD
Displays each examined entity in the octaword integer type (length 16 bytes).

/PACKED:n
Interprets each examined entity as a "packed" decimal number of length *n* nibbles.

/PSL
Displays each examined entity in PSL (processor status longword) format.

/PSW
Displays each examined entity in PSW (processor status word) format. /PSW is like /PSL except that only the low order word (16 bits) is displayed.

/QUADWORD
Displays each examined entity in the quadword integer type (length 8 bytes).

/SOURCE
Displays the source line corresponding to the location of each examined entity.

/[NO]SYMBOL
Controls whether symbolization occurs. By default (/SYMBOL), the debugger symbolizes all addresses, if possible; that is, it converts numeric addresses into their symbolic representation. If you specify /NOSYMBOL, the debugger suppresses symbolization. You can use /NOSYMBOL to increase the speed of command processing.

/TASK

Note: /TASK applies only to Ada programs.

Interprets each examined entity as an Ada task object and displays the task value (the name or task ID) of that task object.

/TYPE=(type-expression)
Interprets the data type of each examined entity according to the type specified by *type-expression*. You may use this qualifier when examining the contents of an untyped location, such as a virtual address.

/WORD
Displays each examined entity in the type word integer (length 2 bytes).

EXAMINE

DESCRIPTION

The EXAMINE command displays the entity at the location denoted by an address expression, in the type associated with that location.

The most common use of the EXAMINE command is to obtain the current value of variables in your program. The debugger lets you specify the variable name in the same way it is specified in the program. Moreover, the debugger knows the compiler-generated type of a variable (whether it is an integer, real, string, array, record, and so on) and displays the variable's value accordingly. This is the default behavior of the EXAMINE command.

In general, the EXAMINE command lets you obtain the contents of any program location. You can use the various qualifiers to override the default behavior and specify another format, data type, or radix.

Examination of an aggregate (a composite data structure such as an array or a record structure) displays the entire aggregate in terms of all its individual components. You do not need to specify individual elements of the aggregate. The aggregate output of an array shows the subscript and value of each array element. Similarly, the aggregate output of a record shows the name and value of each record component.

You can examine array slices by specifying a range of subscripts (use colons to separate the lower and upper bounds).

The EXAMINE command sets the *current entity* built-in symbols %CURLOC and period (.) to the location denoted by the address expression specified. Logical predecessors (%PREVLOC and circumflex (^)) and successors (%NEXTLOC and pressing the RETURN key) are based on the value of the current entity symbol.

Related Commands: DEPOSIT, EVALUATE, (SET, SHOW, CANCEL) RADIX, (SET, SHOW) TYPE, CANCEL TYPE/OVERRIDE.

EXAMPLES

1

```
DBG> EXAMINE COUNT
SUB2\COUNT:  27
```

This command displays the value of the integer variable COUNT, in module SUB2.

2

```
DBG> EXAMINE PART_NUMBER
INVENTORY\PART_NUMBER:  "LP-3592.6-84"
```

This command displays the value of the string variable PART_NUMBER.

3

```
DBG> EXAMINE SUB1\ARR3
SUB1\ARR3
   (1,1):     27.01000
   (1,2):     31.01000
   (1,3):     12.48000
   (2,1):     15.08000
   (2,2):     22.30000
   (2,3):     18.73000
```

This command displays the value of all elements in array ARR3, in module SUB1. ARR3 is a 2 by 3 element array of real numbers.

```
4  DBG> EXAMINE SUB1\ARR3(2,1:3)
   SUB1\ARR3
      (2,1):      15.08000
      (2,2):      22.30000
      (2,3):      18.73000
```

This command displays the value of the elements in a slice of array SUB1\ARR3. The slice includes "columns" 1 through 3 of "row" 2.

```
5  DBG> EXAMINE VALVES.INTAKE.STATUS
   MONITOR\VALVES.INTAKE.STATUS:  OFF
```

This command displays the value of the nested record component VALVES.INTAKE.STATUS in module MONITOR.

```
6  DBG> EXAMINE/SOURCE SWAP
   MAIN\SWAP
       47:  procedure SWAP(X,Y: in out INTEGER) is
```

This command displays the source line on which routine SWAP is declared (the location of routine SWAP).

```
7  DBG> EXAMINE/ASC WORK+20
   DETAT\WORK+20:  "abcd"
```

This command displays the value of the entity in location WORK+20 as an ASCII string (abcd).

```
8  DBG> EXAMINE/INST MAIN+2
   MAIN\MAIN+02:  MOVAL   L^MAINA,R11
```

This command displays the instruction MOVAL from the location MAIN+2.

```
9  DBG> EXAMINE/NOSYM WORKDATA
   0000086F:  03020100
```

This command displays the value of the address WORKDATA (0000086F) as a virtual memory address (03020100).

```
10 DBG> EXAMINE/HEX FIDBLK
   FDEX1$MAIN\FIDBLK
       (1):        00000008
       (2):        00000100
       (3):        000000AB
```

This command displays the value of the array variable FIDBLK in hexadecimal radix.

```
11 DBG> EXAMINE/DECIMAL/WORD NEWDATA:NEWDATA+6
   SUB2\NEWDATA:  256
   SUB2\NEWDATA+2:  770
   SUB2\NEWDATA+4:  1284
   SUB2\NEWDATA+6:  1798
```

This command displays, in decimal radix, the values of word integer entities (2-byte entities) that are in the range of locations denoted by NEWDATA through NEWDATA + 6 bytes.

```
12 DBG> EXAMINE/TASK ALPHA
   SAMPLE.ALPHA:  %TASK 2
```

This command interprets ALPHA to be the address of an Ada task object and displays the task value %TASK 2 associated with that task object.

EXIT

Ends the debugging session, or ends the execution of commands in a command procedure or DO clause.

FORMAT

EXIT

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

When you issue the EXIT command at the terminal, you cause orderly termination of the debugging session: your program's exit handlers (if any) are run, the debugger exit handler is executed (closing log files, restoring the screen and keypad states, and so on), exit status information is displayed, and control is returned to the command interpreter. You cannot then continue to debug your program by issuing the DCL commands DEBUG or CONTINUE. To restart the debugger, you must run the program again.

Note that, since EXIT runs your exit handlers, you can set breakpoints in your exit handlers and they will be activated upon typing EXIT. EXIT can thus be used to debug your exit handlers.

If you want to terminate your debugging session without running your exit handlers, use the QUIT command instead of EXIT.

When the debugger executes an EXIT command in a command procedure, control returns to the command stream that invoked the command procedure. A command stream can be the terminal, an outer (containing) command procedure, a DO clause in a SET BREAK, SET TRACE, or SET WATCH command, or a DO clause in a screen display definition. For example, if the command procedure was invoked from within a DO clause, control returns to that DO clause, where the debugger executes the next command (if any remain in the command sequence).

When the debugger executes an EXIT command in a DO clause, it ignores any remaining commands in that clause and displays its prompt.

Related commands: CTRL/Z, QUIT, CTRL/Y, CTRL/C, @file-spec.

EXAMPLE

```
DBG> EXIT
$
```

This command ends the debugging session and returns you to the DCL command level.

EXITLOOP

Exits an enclosing FOR, REPEAT, or WHILE loop.

FORMAT

EXITLOOP *[n]*

PARAMETERS

n

An integer that specifies the number of levels of nested loops to exit from. The default is 1.

QUALIFIERS

None.

DESCRIPTION

Use the EXITLOOP command to exit an enclosing FOR, REPEAT, or WHILE loop.

Related commands: FOR, REPEAT, WHILE.

EXAMPLE

```
DBG> WHILE 1 DO (STEP; IF X .GT. 3 THEN EXITLOOP)
```

The WHILE 1 command generates an endless loop that executes a STEP command with each iteration. After each STEP, the value of X is tested. If X is greater than 3, the EXITLOOP command terminates the loop.

EXPAND

Expands or contracts the window associated with a screen display.

FORMAT

EXPAND *[disp-name[, . . .]]*

PARAMETERS

disp-name

Specifies a display to be expanded or contracted. You may specify any of the following:

- A predefined display: SRC, OUT, PROMPT, INST, REG
- A display previously created with the SET DISPLAY command
- A pseudo-display name: %CURDISP, %CURSCROLL, %NEXTDISP, %NEXTINST, %NEXTOUTPUT, %NEXTSCROLL, %NEXTSOURCE

If you do not specify a display, the current scrolling display, as established by the SELECT command, is chosen.

QUALIFIERS

You must specify at least one qualifier.

/DOWN[:n]

Moves the bottom border of the display down by *n* lines (if *n* is positive) or up by *n* lines (if *n* is negative). If *n* is omitted, the border is moved down by 1 line.

/LEFT[:n]

Moves the left border of the display to the left by *n* lines (if *n* is positive) or to the right by *n* lines (if *n* is negative). If *n* is omitted, the border is moved to the left by 1 line.

/RIGHT[:n]

Moves the right border of the display to the right by *n* lines (if *n* is positive) or to the left by *n* lines (if *n* is negative). If *n* is omitted, the border is moved to the right by 1 line.

/UP[:n]

Moves the top border of the display up by *n* lines (if *n* is positive) or down by *n* lines (if *n* is negative). If *n* is omitted, the border is moved up by 1 line.

DESCRIPTION

The EXPAND command moves one or more display-window borders according to the qualifiers specified (/UP:[n], /DOWN:[n], RIGHT:[n], /LEFT:[n]).

The EXPAND command does not affect the order of a display on the display pasteboard. Depending on the relative order of displays, the EXPAND command may cause the specified display to hide or uncover another display or be hidden by another display, partially or totally.

Except for the PROMPT display, any display can be contracted to the point where it disappears (at which point it is marked as "removed"). It can then be expanded from that point. Contracting a display to the point where it disappears will cause it to lose any attributes that were selected for it. The PROMPT display cannot be contracted or expanded horizontally but can be contracted vertically to a height of 2 lines.

A window border can be expanded only up to the edge of the screen. The left and top window borders cannot be expanded beyond the left and top edges of the display, respectively. The right border can be expanded up to 255 columns from the left display edge. The bottom border of a source or instruction display can be expanded down only to the bottom edge of the display (to the end of the source module or routine's instructions). A register display cannot be expanded beyond its full size.

See Appendix B for keypad-key definitions associated with the EXPAND command.

Related commands: MOVE, DISPLAY, SELECT/SCROLL, (SET, SHOW) TERMINAL.

EXAMPLES

1
```
DBG> EXPAND/RIGHT:6
```

The EXPAND command moves the right border of the current scrolling display to the right by 6 columns.

2
```
DBG> EXPAND/UP/RIGHT:-12 OUT2
```

The EXPAND command moves the top border of display OUT2 up by 1 line, and the right border to the left by 12 columns.

3
```
DBG> EXPAND/DOWN:99 SRC
```

The EXPAND command moves the bottom border of display SRC down to the bottom edge of the screen.

EXTRACT

Saves the contents of screen displays in a file or creates a file with all of the debugger commands necessary to re-create the current screen state at a later time.

FORMAT

EXTRACT *[disp-name[, . . .]][file-spec]*

PARAMETERS

disp-name

Specifies a display to be extracted. You can use the wildcard character (*) in a display name. When using /ALL, do not specify a display name.

file-spec

Specifies the file to which the information will be written. You can specify a logical name.

If you specify /SCREEN_LAYOUT, the default specification for the file is SYS$DISK:[]DBGSCREEN.COM. Otherwise, the default specification is SYS$DISK:[]DEBUG.TXT.

QUALIFIERS

/ALL

Extracts all displays. If /ALL is used, do not specify a display name. Do not specify /SCREEN_LAYOUT with /ALL.

/APPEND

Appends the information at the end of the file, rather than creating a new file. By default, a new file is created. Do not specify /SCREEN_LAYOUT with /APPEND.

/SCREEN_LAYOUT

Writes a file that contains the debugger commands describing the current state of the screen. This information includes the screen height and width, and the position, display kind, and display attributes of every existing display. This file can then be executed with the @file-spec command to reconstruct the screen at a later time.

DESCRIPTION

When you use the EXTRACT command to save the contents of a display into a file, only those lines that are currently stored in the display's memory buffer (as determined by the /SIZE qualifier on the DISPLAY or SET DISPLAY command) are written to the file.

You cannot extract the PROMPT display into a file.

Related commands: SAVE, DISPLAY.

EXAMPLES

1
```
DBG> EXTRACT SRC
```

This command writes all the lines in display SRC into file SYS$DISK:[]DEBUG.TXT.

2
```
DBG> EXTRACT/APPEND OUT [JONES.WORK]MYFILE
```

This command appends all the lines in display OUT to the end of file [JONES.WORK]MYFILE.TXT.

3
```
DBG> EXTRACT/SCREEN_LAYOUT
```

This command writes the debugger commands needed to reconstruct the screen into file SYS$DISK:[]DBGSCREEN.COM.

FOR

Executes a sequence of commands repetitively a specified number of times.

FORMAT

FOR *name=expression1* **TO** *expression2* [**BY** *expression3*] **DO** *(command[; . . .])*

PARAMETERS

name

Specifies the name of a count variable.

expression1

Specifies an integer or enumeration type value. The *expression1* and *expression2* parameters must always be of the same type.

expression2

Specifies an integer or enumeration type value. The *expression1* and *expression2* parameters must always be of the same type.

expression3

Specifies an integer.

command

Specifies a debugger command. If you specify more than one command, they must be separated by semicolons.

QUALIFIERS

None.

DESCRIPTION

The behavior of the FOR command depends on the value of the *expression3* parameter. If *expression3* is positive, *name* is incremented from the value of *expression1* by the value of *expression3* until it is greater than the value of *expression2*.

If *expression3* is negative, *name* is decremented from the value of *expression1* by the value of *expression3* until it is less than the value of *expression2*.

If *expression3* is zero, the debugger returns an error message.

If *expression3* is left out entirely, the debugger assumes it to have the value +1.

Related commands: REPEAT, WHILE, EXITLOOP.

EXAMPLES

1
```
DBG> FOR I = 10 TO 1 BY -1 DO (EXAMINE A(I))
```

This example examines an array backwards.

2
```
DBG> FOR I = 1 TO 10 DO (DEPOSIT A(I) = 0)
```

This example initializes an array to zero.

GO

Starts or resumes program execution.

FORMAT

GO *[address-expression]*

PARAMETERS

address-expression

Specifies that program execution resume at the location denoted by the address expression. If you do not specify an address expression, execution resumes at the point of suspension or, in the case of debugger start up, at the transfer address.

QUALIFIERS

None.

DESCRIPTION

Note that specifying an address expression with the GO command can produce unexpected results because it alters the normal control flow of your program. For example, static storage may not be initialized, and so on.

Related commands: STEP, SET STEP, SET BREAK, SET TRACE, SET WATCH.

EXAMPLES

1
```
DBG> GO
   .
   .
   .
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion
```

This command starts program execution, which then completes successfully.

2
```
DBG> GO
   .
   .
   .
break at INVENTORY\RESTORE
137: procedure RESTORE;
```

This command starts program execution, which is then suspended at a breakpoint on routine RESTORE in module INVENTORY.

3
```
DBG> GO %LINE 42
   .
   .
   .
```

This command resumes program execution at line 42 of the currently executing module.

HELP

Displays online help on debugger commands and selected topics.

FORMAT

HELP *help-topic [subtopic [. . .]]*

PARAMETERS

help-topic

Specifies the name of a debugger command or topic about which you need help. You can specify the wildcard character (*), either singly or within a name.

subtopic

Specifies a subtopic, command qualifier, or command parameter about which you want further information. You can specify *, either singly or within a name.

QUALIFIERS

None.

DESCRIPTION

The debugger's online help facility provides the following information about any debugger command: a description of the command, format of the command, parameters that may be specified with the command, and qualifiers that may be specified with the command.

To obtain information about a particular qualifier or parameter, specify it as a subtopic. If you want information about all command qualifiers, specify "qualifier" as a subtopic. If you want information about all parameters, specify "parameter" as a subtopic. If you want information about all parameters, qualifiers, and any other subtopics related to a command, specify * as a subtopic.

In addition to help on commands, you can get online help on various topics such as screen features, keypad mode, and so on. Topic keywords are listed along with the commands when you type HELP.

Type HELP Release_Notes for information about any incompatibilities between the current release of the debugger and previous releases. Type HELP New_Features for summary information on new features with this release of the debugger.

For help on the keypad predefined keys, see Appendix B.

EXAMPLE

```
DBG> HELP DEFINE

   DEFINE

        Defines one or more symbols and  assigns  them  specified
        addresses for the duration of the debugging session.

Format:

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

Additional information available:

Parameters
```

This command displays help for the DEFINE command.

IF

Executes a sequence of commands conditionally.

FORMAT

IF *boolean-expression* **THEN** *(command[; . . .]) [* **ELSE** *(command[; . . .])]*

PARAMETERS

boolean-expression

Specifies a language expression that evaluates as a Boolean value (TRUE or FALSE) in the currently set language.

command

Specifies a debugger command. If you specify more than one command, you must separate them with semicolons.

QUALIFIERS

None.

DESCRIPTION

The IF command evaluates a boolean-expression. If the value is TRUE (as defined in the current language), the debugger command list in the THEN clause is executed. If the expression is FALSE, the command list in the ELSE clause is executed (if it is present).

Related commands: FOR, REPEAT, WHILE, EXITLOOP.

EXAMPLE

```
DBG> SET BREAK R DO (IF X .LT.5 THEN (GO) ELSE (EXAMINE X))
```

This command tells the debugger to suspend program execution at location R (a breakpoint) and then resume program execution if the value of X is less than 5 (FORTRAN example). If the value of X is 5 or more, the value of X is displayed.

MOVE

Moves a screen display vertically and/or horizontally across the screen.

FORMAT

MOVE *[disp-name[, . . .]]*

PARAMETERS

disp-name

Specifies a display to be moved. You may specify any of the following:

- A predefined display: SRC, OUT, PROMPT, INST, REG
- A display previously created with the SET DISPLAY command
- A pseudo-display name: %CURDISP, %CURSCROLL, %NEXTDISP, %NEXTINST, %NEXTOUTPUT, %NEXTSCROLL, %NEXTSOURCE

If you do not specify a display, the current scrolling display, as established by the SELECT command, is chosen.

QUALIFIERS

You must specify at least one qualifier.

/DOWN[:n]

Moves the display down by *n* lines (if *n* is positive) or up by *n* lines (if *n* is negative). If *n* is omitted, the display is moved down by 1 line.

/LEFT[:n]

Moves the display to the left by *n* lines (if *n* is positive) or right by *n* lines (if *n* is negative). If *n* is omitted, the display is moved to the left by 1 line.

/RIGHT[:n]

Moves the display to the right by *n* lines (if *n* is positive) or left by *n* lines (if *n* is negative). If *n* is omitted, the display is moved to the right by 1 line.

/UP[:n]

Moves the display up by *n* lines (if *n* is positive) or down by *n* lines (if *n* is negative). If *n* is omitted, the display is moved up by 1 line.

DESCRIPTION

For each display specified, the MOVE command simply creates a window of the same dimensions elsewhere on the screen and maps the display to it, while maintaining the relative position of the text within the window.

The MOVE command does not change the order of a display on the display pasteboard. Depending on the relative order of displays, the MOVE command may cause the display to hide or uncover another display or be hidden by another display, partially or totally.

A display can be moved only up to the edge of the screen.

See Appendix B for keypad-key definitions associated with the MOVE command.

Related commands: EXPAND, DISPLAY, SELECT/SCROLL, (SET, SHOW) TERMINAL.

EXAMPLES

1
```
DBG> MOVE/LEFT
```

The MOVE command moves the current scrolling display to the left by 1 column.

2
```
DBG> MOVE/UP:3/RIGHT:5 NEW_OUT
```

The MOVE command moves display NEW_OUT up by 3 lines and to the right by 5 columns.

QUIT

Ends the debugging session, or ends the execution of commands in a command procedure or DO clause (analogous to EXIT). Does not execute any exit handlers you have declared.

FORMAT

QUIT

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

When you issue the QUIT command at the terminal, you cause orderly termination of the debugging session: the debugger exit handler is executed (closing log files, restoring the screen and keypad states, and so on), exit status information is displayed, and control is returned to the command interpreter. You cannot then continue to debug your program by issuing the DCL commands DEBUG or CONTINUE. To restart the debugger, you must run the program again.

Note that, in contrast to the EXIT command, the QUIT command does not execute any exit handlers that you may have declared.

When the debugger executes a QUIT command in a command procedure, control returns to the command stream that invoked the command procedure. A command stream can be the terminal, an outer (containing) command procedure, a DO clause in a SET BREAK, SET TRACE, or SET WATCH command, or a DO clause in a screen display definition. For example, if the command procedure was invoked from within a DO clause, control returns to that DO clause, where the debugger executes the next command (if any remain in the command sequence).

When the debugger executes a QUIT command in a DO clause, it ignores any remaining commands in that clause and displays its prompt.

Related commands: EXIT, CTRL/Z, CTRL/Y, CTRL/C, @file-spec.

EXAMPLE

```
DBG> QUIT
$
```

This command, when issued from the DBG> prompt, ends the debugging session and returns you to DCL command level.

REPEAT

Executes a sequence of commands repetitively a specified number of times.

FORMAT

REPEAT *lang-exp* **DO** *(command[; . . .])*

PARAMETERS

lang-exp

Denotes any expression in the currently set language that evaluates to a positive integer.

command

Specifies a debugger command. If you specify more than one command, they must be separated by semicolons.

DESCRIPTION

The REPEAT command is a simple form of the FOR command. The REPEAT command executes a sequence of commands repetitively a specified number of times, without providing the options for establishing count parameters that the FOR command does.

Related commands: FOR, WHILE, EXITLOOP.

EXAMPLE

```
DBG> REPEAT 10 DO (STEP)
```

This command causes the debugger to STEP 10 times.

SAVE

Preserves the contents of an existing screen display in a new display.

FORMAT

SAVE *old-disp* **AS** *new-disp [, . . .]*

PARAMETERS

old-disp

Specifies the display whose contents you want to save.

new-disp

Specifies the name of the new display to be created. This new display then receives the contents of the *old-disp* display.

QUALIFIERS

None.

DESCRIPTION

The SAVE command permits a "snapshot" of an existing display to be saved in a new display for later reference. The new display is created with the same text contents as the existing display. In general, the new display is given all the attributes or characteristics of the old display except that it is removed from the screen and is never automatically updated. You can later recall the saved display to the terminal screen with the DISPLAY command.

When you use the SAVE command, only those lines that are currently stored in the display's memory buffer (as determined by the /SIZE qualifier on the DISPLAY or SET DISPLAY command) are stored in the saved display. However, in the case of a saved source or instruction display, the debugger also lets you see any other source lines associated with that module or any other instructions associated with that routine (by scrolling the saved display).

You cannot save the PROMPT display.

Related commands: EXTRACT, DISPLAY.

EXAMPLE

```
DBG> SAVE REG AS OLDREG
```

This command saves the contents of the display named REG into the newly created display named OLDREG.

SCROLL

Scrolls a screen display to make other parts of the text visible through the display window.

FORMAT

SCROLL *[disp-name]*

PARAMETERS

disp-name

Specifies a display to be scrolled. You may specify any of the following:

- A predefined display: SRC, OUT, PROMPT, INST, REG
- A display previously created with the SET DISPLAY command
- A pseudo-display name: %CURDISP, %CURSCROLL, %NEXTDISP, %NEXTINST, %NEXTOUTPUT, %NEXTSCROLL, %NEXTSOURCE

If you do not specify a display, the current scrolling display, as established by the SELECT command, is chosen.

QUALIFIERS

/BOTTOM

Scrolls down to the bottom of the display's text.

/DOWN:[n]

Scrolls down over the display's text by *n* lines to reveal text further down in the display. If *n* is omitted, the display is scrolled by approximately 3/4 of its window height.

/LEFT:[n]

Scrolls left over the display's text by *n* columns to reveal text beyond the left window border. You cannot scroll past column 1. If *n* is omitted, the display is scrolled left by 8 columns.

/RIGHT[:n]

Scrolls right over the display's text by *n* columns to reveal text beyond the right window border. You cannot scroll past column 255. If *n* is omitted, the display is scrolled right by 8 columns.

/TOP

Scrolls up to the top of the display's text.

/UP[:n]

Scrolls up over the display's text by *n* lines to reveal text further up in the display. If *n* is omitted, the display is scrolled by approximately 3/4 of its window height.

SCROLL

DESCRIPTION

The SCROLL command moves a display up, down, right, or left relative to its window so that various parts of the display text can be made visible through the window.

Use the SELECT/SCROLL command to select the target display for the SCROLL command (the *current scrolling display*).

See Appendix B for keypad-key definitions associated with the SCROLL command.

Related commands: SELECT.

EXAMPLES

1
```
DBG> SCROLL/LEFT
```

This command scrolls the current scrolling display to the left by 8 columns.

2
```
DBG> SCROLL/UP:4 ALPHA
```

This command scrolls display ALPHA 4 lines up.

SEARCH

Searches the source code for a specified string and displays source lines that contain an occurrence of the string.

FORMAT

SEARCH *[range] [string]*

PARAMETERS

range

Specifies a program region to be searched. Use any of the following formats:

mod-name	Searches the specified module from line 0 to the end of the module.
mod-name\line-num	Searches the specified module from the specified line number to the end of the module.
mod-name\line-num:line-num	Searches the specified module from the line number specified on the left of the colon to the line number specified on the right.
line-num	Searches the module designated by the current scope setting, from the specified line number to the end of the module.
line-num:line-num	Searches the module designated by the current scope setting from the line number specified on the left of the colon to the line number specified on the right.
null (no entry)	Searches the same module as that from which a source line was most recently displayed (as a result of a TYPE, EXAMINE /SOURCE, or SEARCH command, for example), beginning at the first line following the line most recently displayed and continuing to the end of the module.

string

Specifies the source code characters for which to search. If you do not specify a string, the string specified in the last SEARCH command, if any, is used.

You must enclose the string in quotation marks or apostrophes if

- The string has any leading or trailing space or tab characters
- The string contains an embedded semicolon
- The range parameter is null

If the string is enclosed in quotation marks, use a double quotation mark (""") to indicate an enclosed quotation mark. If the string is enclosed in apostrophes, use a double apostrophe ('') to indicate an enclosed apostrophe.

SEARCH

QUALIFIERS

/ALL
Specifies that the debugger search for all occurrences of the string in the specified range and display every line containing an occurrence of the string.

/IDENTIFIER
Specifies that the debugger search for an occurrence of the string in the specified range but display the string only if it is not bounded on either side by a character that can be part of an identifier in the current language.

/NEXT
Specifies that the debugger search for the first occurrence of the string in the specified range and display only the line containing this occurrence. This is the default.

/STRING
Specifies that the debugger search for and display the string as specified, and not interpret the context surrounding an occurrence of the string, as it does in the case of /IDENTIFIER. This is the default.

DESCRIPTION

SEARCH command qualifiers determine whether the debugger: (1) searches for all occurrences (/ALL) of the string or only the next occurrence (/NEXT); and (2) displays any occurrence of the string (/STRING) or only those occurrences in which the string is not bounded on either side by a character that can be part of an identifier in the current language (/IDENTIFIER).

Note that a module in which a search is to take place must be set. Use the SHOW MODULE command to determine whether a particular module is set. Then use the SET MODULE command, if necessary.

If you plan to issue several SEARCH commands with the same qualifier, you can first use the SET SEARCH command to establish a new default qualifier (for example, SET SEARCH ALL makes the SEARCH command behave like SEARCH/ALL). Then you do not have to use that qualifier with the SEARCH command. You can override the current default qualifiers for the duration of a single SEARCH command by specifying other qualifiers.

Related commands: (SET, SHOW) SEARCH, (SET, SHOW) LANGUAGE, (SET, SHOW) SCOPE, (SET, SHOW) MODULE.

EXAMPLES

1
```
DBG> SEARCH/STRING/ALL 40:50 D
module COBOLTEST
    40: 02      D2N     COMP-2 VALUE -234560000000.
    41: 02      D       COMP-2 VALUE  222222.33.
    42: 02      DN      COMP-2 VALUE -222222.333333.
    47: 02      DR0     COMP-2 VALUE  0.1.
    48: 02      DR5     COMP-2 VALUE  0.000001.
    49: 02      DR10    COMP-2 VALUE  0.00000000001.
    50: 02      DR15    COMP-2 VALUE  0.0000000000000001.
```

This command searches for all occurrences of the letter D in lines 40 through 50 of the module COBOLTEST.

2
```
DBG> SEARCH/IDENTIFIER/ALL 40:50 D
module COBOLTEST
    41: 02      D       COMP-2 VALUE  222222.33.
```

This command searches for all occurrences of the letter D in lines 40 through 50 of the module COBOLTEST. The debugger displays the only line where the letter D (the search string) is not bounded on either side by a character that can be part of an identifier in the current language.

3
```
DBG> SEARCH/NEXT 40:50 D
module COBOLTEST
    40: 02      D2N     COMP-2 VALUE -234560000000.
```

This command searches for the next occurrence of the letter D in lines 40 to 50 of the module COBOLTEST.

4
```
DBG> SEARCH/NEXT
module COBOLTEST
    41: 02      D       COMP-2 VALUE  222222.33.
```

This command searches for the next occurrence of the letter D. The debugger assumes D to be the search string because D was the last one entered and no other search string was specified.

SELECT

Selects a screen display as the current error, input, instruction, output, program, prompt, scrolling, or source display.

FORMAT

SELECT *[disp-name]*

PARAMETERS

disp-name

Specifies the display to be selected. You may specify any one of the following, with the restrictions noted in the qualifier descriptions:

- A predefined display (SRC, OUT, INST, REG, and PROMPT).
- A display previously created with the SET DISPLAY command
- A pseudo-display name: %CURDISP, %CURSCROLL, %NEXTDISP, %NEXTINST, %NEXTOUTPUT, %NEXTSCROLL, %NEXTSOURCE

If you omit this parameter and do not specify a qualifier, you "unselect" the current scrolling display (no display then has the scrolling attribute). If you omit this parameter but specify a qualifier (/INPUT, /SOURCE, and so on), you unselect the current display with that attribute (see the qualifier descriptions).

QUALIFIERS

/ERROR

If you specify a display, selects it as the *current error display*. This causes all debugger diagnostic messages to go to that display. The display specified must be either an output display or the PROMPT display.

If you do not specify a display, the PROMPT display is selected as the current error display.

By default, the PROMPT display has the error attribute.

/INPUT

If you specify a display, selects it as the *current input display*. This causes that display to echo debugger input (which always appears in the PROMPT display). The display specified must be an output display.

If you do not specify a display, the current input display is unselected and debugger input is not echoed to any display (debugger input appears only in the PROMPT display).

By default, no display has the input attribute.

/INSTRUCTION

If you specify a display, selects it as the *current instruction display*. This causes the output of all EXAMINE/INSTRUCTION commands to go to that display. The display specified must be an instruction display.

If you do not specify a display, the current instruction display is unselected and no display has the instruction attribute.

By default, for all languages except MACRO, no display has the instruction attribute. If the language is set to MACRO, the INST display has the instruction attribute by default.

/OUTPUT

If you specify a display, selects it as the *current output display*. This causes debugger output that is not already directed to another display to go to that display. The display specified must be either an output display or the PROMPT display.

If you do not specify a display, the PROMPT display is selected as the current output display.

By default, the OUT display has the output attribute.

/PROGRAM

If you specify a display, selects it as the *current program display*. This causes the debugger to try to force program input and output to that display. Currently, only the PROMPT display may be specified.

If you do not specify a display, the current program display is unselected and program input and output are no longer forced to the specified display.

By default, the PROMPT display has the program attribute, except on MicroVAX workstations, where the program attribute is unselected.

/PROMPT

Selects the specified display as the *current prompt display*. This is where the debugger prompts for input. Currently, only the PROMPT display may be specified. Moreover, you cannot unselect the PROMPT display (the PROMPT display always has the prompt attribute).

/SCROLL

If you specify a display, selects it as the *current scrolling display*. This is the default display for the SCROLL, MOVE, and EXPAND commands. Although any display may have the scroll attribute, note that you can use only the MOVE and EXPAND commands (not the SCROLL command) with the PROMPT display.

If you do not specify a display, the current scrolling display is unselected and no display has the scroll attribute.

By default, for all languages except MACRO, the SRC display has the scroll attribute. If the language is set to MACRO, the INST display has the scroll attribute by default.

Note: If no qualifier is specified, /SCROLL is assumed by default.

/SOURCE

If you specify a display, selects it as the *current source display*. This causes the output of all TYPE and EXAMINE/SOURCE commands to go to that display. The display specified must be a source display.

If you do not specify a display, the current source display is unselected and no display has the source attribute.

By default, for all languages except MACRO, the SRC display has the source attribute. If the language is set to MACRO, no display has the source attribute by default.

SELECT

DESCRIPTION

Attributes are used to select the current scrolling display and to direct various types of debugger output to particular displays. This gives you the option of mixing or isolating different types of information, such as debugger input, output, diagnostic messages, and so on in scrollable displays.

You use the SELECT command with one or more qualifiers (/ERROR, /SOURCE, and so on) to assign one or more corresponding attributes to a display. If you do not specify a qualifier, the /SCROLL qualifier is assumed by default.

If you use the SELECT command without specifying a display name, in general the attribute assignment indicated by the command qualifier is canceled ("unselected"). To reassign display attributes you must use another SELECT command. See the individual qualifier descriptions for details.

See Appendix B for keypad-key definitions associated with the SELECT command.

Related commands: SHOW SELECT, SCROLL, MOVE, EXPAND, DISPLAY, SET DISPLAY.

EXAMPLES

1
```
DBG> SELECT/SOURCE/SCROLL SRC2
```

The SELECT/SOURCE/SCROLL command selects display SRC2 as the current source and scrolling display.

2
```
DBG> SELECT/INPUT/ERROR OUT
```

The SELECT/INPUT/ERROR command selects display OUT as the current input and error display. This causes debugger input, debugger output (assuming OUT is the current output display), and debugger diagnostic messages to be logged in the OUT display in the correct sequence.

3
```
DBG> SELECT/SOURCE
```

The SELECT/SOURCE command unselects (removes the source attribute from) the currently selected source display. The output of a TYPE or EXAMINE/SOURCE command will then go to the currently selected output display.

SET ATSIGN

Establishes the default file specification that the debugger uses when searching for command procedures.

FORMAT

SET ATSIGN *file-spec*

PARAMETERS

file-spec

Specifies any part of a VAX/VMS file specification (for example, a directory name or a file type) that the debugger is to use by default when searching for a command procedure. If you do not supply a full file specification, the debugger assumes SYS$DISK:[]DEBUG.COM as the default file specification for any missing field.

You may specify a logical name that translates to a search list. In this case, the debugger processes the file specifications in the order they appear in the search list until the command procedure is found.

QUALIFIERS

None.

DESCRIPTION

When you invoke a command procedure during a debugging session, the debugger, by default, assumes that its file specification is SYS$DISK:[]DEBUG.COM. The SET ATSIGN command lets you override this default.

Related commands: @file-spec, SHOW ATSIGN.

EXAMPLES

1
```
DBG> SET ATSIGN USER:[JONES.DEBUG].DBG
DBG> @TEST
```

When you invoke @TEST, the debugger looks for the file TEST.DBG in USER:[JONES.DEBUG].

SET BREAK

Establishes a breakpoint at the location denoted by an address-expression, or at instructions of a particular class.

FORMAT

SET BREAK *[addr-expr[, . . .]] [* **WHEN***(cond-expr)] [* **DO***(command[; . . .])]*

PARAMETERS

addr-expr

Specifies an address expression (a program location) at which a breakpoint is to be set. In general, this may be a line number, a routine name, a label, or a location in memory. However, the /MODIFY and /RETURN qualifiers are used with specific kinds of address expressions. Do not use the wildcard character (*). Do not use an address expression when specifying /BRANCH, /CALL, /EXCEPTION, /INSTRUCTION[=(opcode-list)], /INTO, /[NO]JSB, /LINE, /OVER, /[NO]SHARE, or /[NO]SYSTEM.

command

Specifies a debugger command that is to be executed as part of the DO clause when break action is taken.

cond-expr

Specifies a conditional expression in the currently set language that is to be evaluated every time the breakpoint occurs. If the expression is TRUE, break action occurs, and the debugger reports that a break has occurred. If the expression is FALSE, break action does not occur. In this case, a report is not issued, the commands specified by the DO clause are not executed, and program execution is continued.

QUALIFIERS

/AFTER:n

Specifies that break action not be taken until the *n*th time the designated breakpoint is encountered (*n* is a decimal integer). Thereafter, the breakpoint occurs every time it is encountered provided that conditions in the WHEN clause are TRUE. The command SET BREAK/AFTER:1 has the same effect as the SET BREAK command.

/BRANCH

Causes the debugger to break on every branch instruction encountered (including 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, ACBG, ACBH, AOBLEQ, AOBLSS, SOBGEQ, SOBGTR, CASEB, CASEW, CASEL) during execution. Do not specify an address expression with this qualifier. See also /INTO, /OVER.

/CALL

Causes the debugger to break on every call instruction (including the CALLS, CALLG, BSBW, BSBB, JSB, RSB, and RET instructions) encountered during execution. Do not specify an address expression with this qualifier. See also /INTO, /OVER.

/EVENT=event-name

Note: /EVENT applies only to Ada and SCAN. See the VAX Ada and VAX SCAN documentation for complete information.

Causes the debugger to break every time the specified *event occurs* (if that event is defined and detected by the run-time system). If you specify an address expression with /EVENT, causes the debugger to break every time the specified event occurs for that address expression. Event names depend on the run-time facility and are identified in Appendix E for Ada and SCAN. Note that you cannot specify an address expression with certain event names.

/EXCEPTION

Causes the debugger to break every time an exception is signaled. The break action occurs before any user-written exception handlers are invoked. SET BREAK/EXCEPTION is the same as SET EXCEPTION BREAK. Do not specify an address expression with this qualifier.

/INSTRUCTION

Causes the debugger to break on every instruction executed. Do not specify an address expression with this qualifier. See also /INTO, /OVER.

/INSTRUCTION=(opcode[, . . .])

Causes the debugger to break on every instruction whose opcode is in the list. Do not specify an address expression with this qualifier. See also /INTO, /OVER.

/INTO

Sets breakpoints within called routines (as well as within the main program) when /BRANCH, /CALL, /INSTRUCTION=[(opcode-list)], or /LINE is specified; that is, when an address expression is not explicitly specified. /INTO is the default behavior and is the opposite of /OVER. When using /INTO, you can further qualify the breakpoints with the /[NO]JSB, /[NO]SHARE, and /[NO]SYSTEM qualifiers.

/[NO]JSB

Qualifies /INTO. Use /[NO]JSB with /INTO and one of these qualifiers: /BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE. /JSB is the default for all languages except DIBOL. /JSB lets the debugger break within routines that are called by the JSB or CALL instruction. /NOJSB (the DIBOL default) specifies that breakpoints not be set within routines called by JSB instructions. In DIBOL, user-written routines are called by the CALL instruction and DIBOL run-time library routines are called by the JSB instruction. Do not specify an address expression with this qualifier.

/LINE

Causes the debugger to break at the start of each new line. Do not specify an address expression with this qualifier. See also /INTO, /OVER.

/MODIFY

Causes a break at every instruction that writes to and modifies the value of the location indicated by the address expression. The address expression is typically a variable name.

The SET BREAK/MODIFY command acts exactly like a SET WATCH command and operates under the same restrictions.

If you specify an absolute address for the address expression, the debugger may not be able to associate the address with a particular data object. In this case, the debugger uses a default length of 4 bytes. You can change this length, however, by setting the type to either WORD (which changes the default length to 2 bytes) or BYTE (which changes the default length to 1 byte).

/OVER

Sets breakpoints only within the main program (not within called routines) when /BRANCH, /CALL, /INSTRUCTION=[(opcode-list)], or /LINE is specified; that is, when an address expression is not explicitly specified. /OVER is the opposite of /INTO.

/RETURN

Sets a breakpoint on the RETURN (RET) instruction from an indicated routine. This qualifier can only be applied to routines called with a CALLS or CALLG instruction; it cannot be used with JSB routines. Breaking on the RET instruction also allows you to inspect the local environment before the RET instruction removes the routine's call frame from the call stack.

For this qualifier, the address-expression parameter is an instruction address within a CALLS or CALLG routine. It may simply be a routine name, in which case it specifies the routine start address. However, you can also specify another location in a routine, so you can see only those returns that are taken after a certain code path is followed.

/[NO]SHARE

Qualifies /INTO. Use /[NO]SHARE with /INTO and one of these qualifiers: BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE. /SHARE (default) lets the debugger break within shareable image routines as well as other routines. /NOSHARE specifies that breakpoints not be set within shareable images. Do not specify an address expression with this qualifier.

/[NO]SILENT

Controls whether or not the "break . . . " message (and source code) is displayed when break action is taken. /NOSILENT (default) specifies that the message be displayed. /SILENT specifies that no message or source code be displayed. /SILENT overrides /SOURCE.

/[NO]SOURCE

Controls whether or not the source code is displayed when break action is taken. /SOURCE (default) specifies that the source code be displayed. /NOSOURCE specifies that no source code be displayed. /SILENT overrides /SOURCE.

/[NO]SYSTEM

Qualifies /INTO. Use /[NO]SYSTEM with /INTO and one of these qualifiers: /BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE. /SYSTEM (default) lets the debugger break within system routines (P1 space) as well as other routines. /NOSYSTEM specifies that breakpoints not be set within system routines. Do not specify an address expression with this qualifier.

/TEMPORARY

Causes the breakpoint to disappear after it is activated (the breakpoint does not remain permanently set).

DESCRIPTION

When a breakpoint is activated, the debugger takes the following action:

1. Suspends program execution at the breakpoint location.
2. Evaluates the expression in a WHEN clause, if one was specified when the breakpoint was set. If the value of the expression is FALSE, execution continues and the debugger does not perform the next three steps.
3. Displays the location of the breakpoint. Also, displays the line of source code corresponding to that instruction if the SOURCE parameter is in effect by virtue of a previous SET STEP SOURCE command.
4. Executes the commands in a DO clause, if one was specified when the breakpoint was set. If the DO clause contains a GO command, execution continues and the debugger does not perform the next step.
5. Issues the DBG> prompt.

The following qualifiers affect what output is seen when a breakpoint is reached:

/[NO]SILENT
/[NO]SOURCE

The following qualifiers affect the timing and duration of breakpoints:

/AFTER:n
/TEMPORARY

Breakpoints may be set on classes of instructions or events by using one of the following qualifiers:

/BRANCH
/CALL
/EVENT=event-name
/EXCEPTION
/INSTRUCTION
/INSTRUCTION=(opcode-list)
/LINE
/RETURN

The following qualifiers affect what happens at a routine call:

/INTO
/[NO]JSB
/OVER
/[NO]SHARE

/[NO]SYSTEM

The /MODIFY qualifier is used to monitor changes at program locations (typically changes in the values of variables).

Related commands: (SHOW, CANCEL) BREAK, CANCEL ALL, SET TRACE, SET WATCH, GO, STEP, (SET, SHOW) EVENT_FACILITY.

EXAMPLES

1
```
DBG> SET BREAK SWAP\%LINE 12
```

This command sets a breakpoint on line 12 of module SWAP.

2
```
DBG> SET BREAK/AFTER:3 SUB2
```

This command sets a breakpoint that will trigger on the third and subsequent times that SUB2 (a routine) is executed.

3
```
DBG> SET BREAK LOOP1 DO (EXAMINE D; STEP; EXAMINE Y; GO)
```

This command sets a breakpoint at location LOOP1. When the breakpoint is reached, the following commands are executed:

```
EXAMINE D
STEP
EXAMINE Y
GO
```

4
```
DBG> SET BREAK/TEMPORARY 1440
DBG> SHOW BREAK
breakpoint at 1440 [temporary]
```

This command sets a temporary breakpoint at location 1440. After that breakpoint is activated, it disappears.

SET DEFINE

Establishes a default qualifier (/ADDRESS, /COMMAND, or /VALUE) for the DEFINE command.

FORMAT

SET DEFINE *define-default*

PARAMETERS

define-default

Specifies the default to be established for the DEFINE command. Valid keywords (which correspond to DEFINE command qualifiers) are the following:

ADDRESS	Subsequent DEFINE commands will be treated as DEFINE /ADDRESS. This is the default.
COMMAND	Subsequent DEFINE commands will be treated as DEFINE /COMMAND.
VALUE	Subsequent DEFINE commands will be treated as DEFINE/VALUE.

QUALIFIERS

None.

DESCRIPTION

The SET DEFINE command establishes a default qualifier for subsequent DEFINE commands. The parameters that you specify in the SET DEFINE command have the same names as the DEFINE command qualifiers. DEFINE command qualifiers determine whether the DEFINE command binds a symbol to an address, a command string, or a value.

You can override the current DEFINE default for the duration of a single DEFINE command by specifying another qualifier. Use the SHOW DEFINE command to identify the current DEFINE defaults.

Related commands: SHOW DEFINE, DEFINE, DELETE, SHOW SYMBOL /DEFINED.

EXAMPLE

```
DBG> SET DEFINE VALUE
```

The SET DEFINE VALUE command specifies that subsequent DEFINE commands are to be treated as DEFINE/VALUE.

SET DISPLAY

Creates a new screen display.

FORMAT

SET DISPLAY *disp-name* [**AT** *wspec*] [*dkind*] [*, . . .*]

PARAMETERS

disp-name

Specifies the name of the display you are defining. If a display by the same name already exists, you must cancel the first display before you can define the new display.

You may specify more than one display, each with an optional window specification (wspec) and display kind (dkind).

wspec

Specifies the screen window at which the display is to be positioned. You may specify any of the following:

- A predefined window. For example, RH1 (right top half). See Appendix C.
- A window definition previously established with the SET WINDOW command.
- A window specification of the form *(start-line, line-count [,start-column, column-count])*. The specification can include expressions which may be based on the built-in symbols %PAGE and %WIDTH (for example, %WIDTH/4).

If you omit the *wspec* parameter, the display is positioned at window H1 or H2 by default, alternating between H1 and H2 with each SET DISPLAY command.

dkind

Specifies the display kind. Valid keywords are the following:

DO (cmd-list)	Specifies an automatically updated output display. The commands in cmd-list are executed each time the debugger gains control. Their output forms the contents of the display.
INSTRUCTION	Specifies an instruction display. If selected as the current instruction display with the SELECT /INSTRUCTION command, it will display the output from subsequent EXAMINE/INSTRUCTION commands.

INSTRUCTION (command)	Specifies an automatically updated instruction display. The command specified must be an EXAMINE/INSTRUCTION command. The instruction display is updated each time the debugger gains control.
OUTPUT	Specifies an output display. If selected as the current output display with the SELECT/OUTPUT command, it will display any debugger output that is not directed to another display. If selected as the current input display with the SELECT /INPUT command, it will echo debugger input. If selected as the current error display with the SELECT/ERROR command, it will display debugger diagnostic messages.
REGISTER	Specifies an automatically updated register display. The display is updated each time the debugger gains control.
SOURCE	Specifies a source display. If selected as the current source display with the SELECT /SOURCE command, it will display the output from subsequent TYPE or EXAMINE/SOURCE commands.
SOURCE (command)	Specifies an automatically updated source display. The command specified must be a TYPE or EXAMINE/SOURCE command. The source display is updated each time the debugger gains control.

If you omit the *dkind* parameter, an OUTPUT display is created.

QUALIFIERS

/[NO]DYNAMIC

Controls whether a display automatically adjusts its window dimensions in proportion when a SET TERMINAL command is issued. By default (/DYNAMIC) all newly created displays adjust their window dimensions automatically, except for register displays. A register display maintains its window dimensions when the screen height or width are changed.

/HIDE

Places a newly created display at the bottom of the display pasteboard. This hides the new display behind any previously existing displays that share the same region of the screen.

/HIDE has the same effect as /PUSH.

/MARK_CHANGE

Marks the lines that change in a DO(cmd-list) display each time the display is automatically updated. Any lines in which the contents have changed since the last time the display was updated are highlighted with reverse video. This qualifier is particularly useful when you want any variables in an automatically updated display to be highlighted when they change.

This qualifier is not applicable to other kinds of displays.

/POP
Places a newly created display at the top of the display pasteboard, ahead of any other displays except the PROMPT display. The new display then hides any other displays that share the same region of the screen, except for the PROMPT display. This is the default action of the SET DISPLAY command.

/PUSH
Has the same effect as /HIDE.

/REMOVE
Specifies that the display not be shown on the screen unless you explicitly request it with the DISPLAY command. The display is then marked as being removed from the display pasteboard, although it still exists.

/SIZE:n
Sets the maximum size of a display to be *n* lines. If more than *n* lines are written to the display, the oldest lines are lost as new lines are added. If you omit this qualifier, the default size is 64 lines, except for the predefined display OUT (100 lines).

For an output or DO display, /SIZE:n specifies that the display should hold the *n* most recent lines of output. For a source or instruction display, *n* gives the number of source lines or lines of instructions that can be placed in the memory buffer at any one time. However, you can scroll a source display over the entire source code of the module whose code is displayed (source lines are paged into the buffer as needed). Similarly, you can scroll an instruction display over all of the instructions of the routine whose instructions are displayed (instructions are decoded from the image as needed).

DESCRIPTION

The SET DISPLAY command is used to create a new display. The command lets you specify the name, window, and display kind. By default, an output display is created, and it is placed on top of the display pasteboard, ahead of any existing displays but behind the PROMPT display. You can also hide a newly created display at the bottom of the pasteboard, so it will not conceal existing displays. And you can create a new "removed" display.

Related commands: (SHOW, CANCEL) DISPLAY, DISPLAY, (SET, SHOW, CANCEL) WINDOW, SELECT, (SET, SHOW) TERMINAL.

EXAMPLES

1
```
DBG> SET DISPLAY DISP2 AT RS45
DBG> SELECT/OUTPUT DISP2
```

The SET DISPLAY command creates a new display named DISP2 essentially at the right bottom half of the screen, above the PROMPT display, which is located at S6. This is an output display by default. The SELECT/OUTPUT command then selects DISP2 as the current output display.

2
```
DBG> SET WINDOW TOP AT (1,8,45,30)
DBG> SET DISPLAY NEWINST AT TOP INSTRUCTION
DBG> SELECT/INST NEWINST
```

The SET WINDOW command creates a window named TOP starting at line 1 and column 45, and extending down for 8 lines and to the right for 30 columns. The SET DISPLAY command creates an instruction display named NEWINST to be displayed through TOP. The SELECT/INST command selects NEWINST as the current instruction display.

SET EDITOR

Establishes the editor that will be invoked by the EDIT command.

FORMAT

SET EDITOR *[command-line]*

PARAMETERS

command-line

Specifies a command line to invoke a particular editor on your system when you use the EDIT command.

You must specify a command line unless you use the /CALLABLE_EDT, /CALLABLE_LSEDIT, or /CALLABLE_TPU qualifiers. If you do not use one of these qualifiers, the editor specified in the SET EDITOR command line is spawned to a subprocess when you issue the EDIT command.

You may specify a command line with the /CALLABLE_LSEDIT and /CALLABLE_TPU qualifiers, but not with the /CALLABLE_EDT qualifier.

QUALIFIERS

/CALLABLE_EDT

Specifies that the callable version of the EDT editor is to be invoked when you use the EDIT command. Do not specify a command line with /CALLABLE_EDT (a command line of "EDT" is used).

/CALLABLE_LSEDIT

Specifies that the callable version of the VAX Language-Sensitive Editor (LSEDIT) is to be invoked when you use the EDIT command. If you also specify a command line, it is passed to callable LSEDIT. If you do not specify a command line, the default command line is "LSEDIT".

/CALLABLE_TPU

Specifies that the callable version of the VAX Text Processing Utility (VAXTPU) is to be invoked when you use the EDIT command. If you also specify a command line, it is passed to callable VAXTPU. If you do not specify a command line, the default command line is "EDIT/TPU".

/[NO]START_POSITION

Note: Currently, only the VAX Language-Sensitive Editor (specified either as LSEDIT or /CALLABLE_LSEDIT) supports this qualifier.

Controls whether the /START_POSITION qualifier is appended to the specified or default command line when the EDIT command is used. This qualifier affects the initial position of the editor's cursor. By default, (/NOSTART_POSITION), the editor's cursor is placed at the start of source line 1, regardless of which line is centered in debugger's source display or whether a line number is specified in the EDIT command. If /START_POSITION is specified, the cursor is placed either on the line whose number is specified in the EDIT command, or (if no line number is specified) on the line that is centered in the current source display.

DESCRIPTION

The SET EDITOR command may be used to specify any editor that is installed on your system. In general, the command line specified as parameter to the SET EDITOR command is spawned and executed in a subprocess. However, if you use EDT, LSEDIT, or VAXTPU, you have the option of invoking these editors in a more efficient way. You can specify the /CALLABLE_EDT, /CALLABLE_LSEDIT, or /CALLABLE_TPU qualifiers, which cause the callable versions of EDT, LSEDIT, and VAXTPU, respectively, to be invoked by the EDIT command. In the case of LSEDIT and VAXTPU, you may also specify a command line that will be executed by the callable editor.

Related commands: SHOW EDITOR, EDIT, (SET, SHOW, CANCEL) SOURCE.

EXAMPLES

1

```
DBG> SET EDITOR '@MAIL$EDIT ""'
```

The SET EDITOR command causes the EDIT command to spawn the command line '@MAIL$EDIT ""', which invokes the same editor as you use in MAIL.

2

```
DBG> SET EDITOR/CALLABLE_TPU
```

The SET EDITOR command causes the EDIT command to invoke callable VAXTPU with the default command line of EDIT/TPU.

3

```
DBG> SET EDITOR/CALLABLE_TPU EDIT/TPU/SECTION=MYSECINI.TPU$SECTION
```

The SET EDITOR command causes the EDIT command to invoke callable VAXTPU with the command line EDIT/TPU /SECTION=MYSECINI.TPU$SECTION.

4

```
DBG> SET EDITOR/CALLABLE_LSEDIT/START_POSITION
```

The SET EDITOR command causes the EDIT command to invoke callable LSEDIT with the default command line of LSEDIT. Also the /START_POSITION qualifier will be appended to the command line, so that the editing session will start on the source line that is centered in the debugger's current source display.

SET EVENT_FACILITY

Establishes the run-time library facility for eventpoints that are set with the SET BREAK/EVENT and SET TRACE/EVENT commands.

Note: The SET EVENT_FACILITY command currently applies only to Ada and SCAN. See the VAX Ada and VAX SCAN documentation for complete information.

FORMAT

SET EVENT_FACILITY *facility-name*

PARAMETERS

facility-name

Specifies a run-time library facility for eventpoints. Valid keywords are the following:

ADA Enables recognition of Ada-specific events when you use the (SET, CANCEL) BREAK/EVENT and (SET, CANCEL) TRACE/EVENT commands. Valid Ada event names are identified in Appendix E.

SCAN Enables recognition of SCAN-specific events when you use the (SET, CANCEL) BREAK/EVENT and (SET, CANCEL) TRACE/EVENT commands. Valid SCAN event names are identified in Appendix E.

QUALIFIERS

None.

DESCRIPTION

The Ada event facility lets you set breakpoints and tracepoints on tasking events and exception events. The SCAN event facility lets you set breakpoints and tracepoints on pattern-matching events.

Use the SHOW EVENT_FACILITY command to identify the events applicable to the currently set language.

Related commands: SHOW EVENT_FACILITY, (SET, CANCEL) BREAK /EVENT, SHOW BREAK, (SET, CANCEL) TRACE/EVENT, SHOW TRACE.

EXAMPLES

```
1  DBG> SET EVENT_FACILITY ADA
```

This command establishes Ada as the current run-time library facility.

SET EXCEPTION BREAK

Causes the debugger to treat any exception condition generated by your program as a breakpoint.

FORMAT

SET EXCEPTION BREAK

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The SET EXCEPTION BREAK command has the same effect as the SET BREAK/EXCEPTION command.

As a result of this command, whenever your program generates an exception condition, the debugger suspends program execution, reports the exception condition, and prompts you for input. Thus, whenever an exception breakpoint is activated, you have the opportunity to issue debugger commands. When you want to continue program execution, you can issue one of the following GO commands:

- A GO command without an address-expression parameter. In this case, the debugger fields and resignals the exception, thus allowing any user-declared exception handlers to execute.
- A GO command with an address-expression parameter. In this case, the debugger allows program execution to continue at the specified location, thus inhibiting the execution of any user-declared exception handlers.

Note that you cannot issue a STEP or CALL command to continue program execution after an exception breakpoint is activated. The debugger issues a warning message because the STEP or CALL command is illegal in this context.

Related commands: CANCEL EXCEPTION BREAK, (SET, CANCEL) BREAK /EXCEPTION.

EXAMPLE

```
DBG> SET EXCEPTION BREAK
```

This command tells the debugger to treat an exception condition as a breakpoint. This allows you to issue debugger commands when your program generates an exception condition.

SET IMAGE

Loads the run-time symbol table (RST) for one or more shareable images and establishes the current image.

FORMAT

SET IMAGE *[image-name[, . . .]]*

PARAMETERS

image-name

Specifies a shareable image that is to be "set" (that is, loaded into the RST). Do not use the wildcard character (*). When using /ALL, do not specify an image name.

QUALIFIERS

/ALL

Specifies that all shareable images are to be set. When using /ALL, do not specify an image.

DESCRIPTION

The "current" image is the current debugging context: a SHOW MODULE command identifies the modules of the current image. If only one image is specified with the SET IMAGE command, that image becomes the current image. If a list of images is specified, the last one in the list becomes the current image. If /ALL is specified, the current image is unchanged.

Before an image can be set with the SET IMAGE command, it must have been linked with the /DEBUG or /TRACE qualifier on the LINK command. If an image was linked /NOTRACE, no symbol information is available for that image and you cannot specify it with the SET IMAGE command.

Note that when you use the SET IMAGE command to establish a new current image, all definitions created with the DEFINE/ADDRESS and DEFINE/VALUE commands are deleted (definitions created with the DEFINE /COMMAND and DEFINE/KEY commands are retained, however).

Related commands: (SHOW, CANCEL) IMAGE, (SET, SHOW, CANCEL) MODULE.

EXAMPLES

1
```
DBG> SET IMAGE SHARE1
DBG> SET MODULE SUBR
DBG> SET BREAK SUBR
```

This sequence of commands shows how to set a breakpoint on routine SUBR in module SUBR of shareable image SHARE1. The SET IMAGE command sets the debugging context to SHARE1. The SET MODULE command loads the symbol records of module SUBR into the RST. The SET BREAK command sets a breakpoint on routine SUBR.

SET KEY

Establishes the current key state.

FORMAT

SET KEY

PARAMETERS

None.

QUALIFIERS

/[NO]LOG

Controls whether a message is displayed indicating that the key state has been set. /LOG (default) displays the message.

/[NO]STATE[=state-name]

Specifies a key state to be established as the current state. You may specify a predefined key state, such as GOLD, or a user-defined state. A state name can be any appropriate alphanumeric string. /NOSTATE (default) leaves the current state unchanged.

DESCRIPTION

Keypad mode must be enabled (SET MODE KEYPAD) before you can use this command. Keypad mode is enabled by default.

By default, the current key state is the "DEFAULT" state. When you define function keys using the DEFINE/KEY command, you can use the /IF_STATE qualifier of that command to assign a specific state name to the key definition. If that state is not set when you press the key, the definition will not be processed. The SET KEY/STATE command lets you change the current state to the appropriate state.

You can also change the current state by pressing a key that causes a state change (a key that was defined with the DEFINE/KEY/LOCK_STATE/SET_STATE qualifier combination).

Related commands: DEFINE/KEY, DELETE/KEY, SHOW KEY.

EXAMPLE

```
DBG> SET KEY/STATE=PROG3
```

The SET KEY command changes the key state to the PROG3 state. You can now use the key definitions that are associated with this state.

SET LANGUAGE

Establishes the current language.

FORMAT

SET LANGUAGE *language-name*

PARAMETERS

language-name

Specifies a language. Valid keywords are the following: ADA, BASIC, BLISS, C, COBOL, DIBOL, FORTRAN, MACRO, PASCAL, PLI, RPG, SCAN, and UNKNOWN.

QUALIFIERS

None.

DESCRIPTION

At debugger start up, the debugger sets the current language to that in which the module containing the transfer address (the main program) is written. If you want to debug a module written in a different source language from that of the main program, you can change the language with the SET LANGUAGE command.

The current language influences several debugger default parameters, namely, those of type, radix, and step values. In addition, languages differ on such matters as type conversions; the evaluation of expressions; and acceptable syntax, special characters, and symbols.

The SET LANGUAGE UNKNOWN command may be used when debugging a program that is written in an unsupported language. To maximize the useability of the debugger with unsupported languages, the SET LANGUAGE UNKNOWN command allows the debugger to accept data formats that match those of any supported language.

Related commands: SHOW LANGUAGE, SET TYPE, SET RADIX, SET MODE.

EXAMPLES

1
```
DBG> SET LANG COBOL
```

This command establishes COBOL as the current language.

2
```
DBG> SET LANG PASCAL
```

This command establishes PASCAL as the current language.

SET LOG

Specifies a log file to which the debugger writes after a SET OUTPUT LOG command has been issued.

FORMAT

SET LOG *file-spec*

PARAMETERS

file-spec

Denotes the file specification of the log file. If you do not supply a full file specification, the debugger assumes SYS$DISK:[ ]DEBUG.LOG as the default file specification for any missing field.

If you specify a version number and that version of the file already exists, the debugger writes to the file specified, appending the log of the debugging session onto the end of that file.

QUALIFIERS

None.

DESCRIPTION

Note that the SET LOG command only determines the name of a log file; it does not cause the debugger to create or write to the specified file. The SET OUTPUT LOG command accomplishes that.

If you have issued a SET OUTPUT LOG command but no SET LOG command, the debugger writes to the file SYS$DISK:[ ]DEBUG.LOG by default.

If the debugger is writing to a log file and you specify another log file with the SET LOG command, the debugger closes the former file and begins writing to the file specified in the SET LOG command.

Related commands: SET OUTPUT LOG, SET OUTPUT SCREEN_LOG.

EXAMPLES

1
```
DBG> SET LOG CALC
DBG> SET OUTPUT LOG
```

The SET LOG command specifies the debugger log file to be SYS$DISK:[ ]CALC.LOG. The SET OUTPUT command causes your input and debugger output to be logged to that file.

2
```
DBG> SET LOG "[CODEPROJ]FEB29.TMP"
DBG> SET OUTPUT LOG
```

The SET LOG command specifies the debugger log file to be [CODEPROJ]FEB29.TMP. The SET OUTPUT command causes your input and debugger output to be logged to that file.

SET MARGINS

Specifies the leftmost source-line character position at which to begin display of a source line and/or the rightmost character position at which to end display of a source line.

FORMAT

SET MARGINS *rm*
lm:rm
lm:
:rm

PARAMETERS

lm

The source-line character position at which to begin display of the line of source code (the left margin).

rm

The source-line character position at which to end display of the line of source code (the right margin).

QUALIFIERS

None.

DESCRIPTION

By default, the debugger displays a source line beginning at character position 1 of the source line. This is actually character position 9 on your terminal screen. The first eight character positions on the screen are reserved for the line number and cannot be manipulated by the SET MARGINS command.

If you specify a single number, the debugger sets the left margin to 1 and the right margin to the number specified.

If you specify two numbers, separated with a colon, the debugger sets the left margin to the number on the left of the colon and the right margin to the number on the right.

If you specify a single number followed by a colon, the debugger sets the left margin to that number and leaves the right margin unchanged.

If you specify a colon followed by a single number, the debugger sets the right margin to that number and leaves the left margin unchanged.

Increasing the left margin setting is useful when the source code is deeply indented. Decreasing the right margin setting (from its default value of 255) prevents the wrapping of long lines by truncating them.

The SET MARGINS command affects only the display of source lines, that is, the display resulting from commands such as TYPE and EXAMINE/SOURCE. The SET MARGINS command does not affect the display resulting from commands (such as EXAMINE, EVALUATE, SHOW MODE, and so on) that do not display source code. If a command displays source code together with other information—as, for example, STEP/SOURCE does—the display

of source code is affected by the current margin settings but the other information is not.

Related commands: SHOW MARGINS.

EXAMPLES

1
```
DBG> SHOW MARGINS
left margin: 1 , right margin: 255
DBG> TYPE 14
module FORARRAY
   14:              DIMENSION IARRAY(4:5,5), VECTOR(10), I3D(3,3,4)
```

This example displays the default margin settings for a line of source code (1 and 255).

2
```
DBG> SET MARGINS 39
DBG> SHOW MARGINS
left margin: 1 , right margin: 39
DBG> TYPE 14
module FORARRAY
   14:              DIMENSION IARRAY(4:5,5), VECTOR
```

This example shows how the display of a line of source code changes when you change the right margin setting from 255 to 39.

3
```
DBG> SET MARGINS 10:45
DBG> SHOW MARGINS
left margin: 10 , right margin: 45
DBG> TYPE 14
module FORARRAY
   14: IMENSION IARRAY(4:5,5), VECTOR(10),
```

This example shows the display of the same line of source code after both margins are changed.

4
```
DBG> SET MARGINS :100
DBG> SHOW MARGINS
left margin: 10 , right margin: 100
```

This example shows how to change the right margin setting while retaining the previous left margin setting.

5
```
DBG> SET MARGINS 5:
DBG> SHOW MARGINS
left margin: 5 , right margin: 100
```

This example shows how to change the left margin setting while retaining the previous right margin setting.

SET MAX_SOURCE_FILES

Specifies the maximum number of source files that the debugger may keep open at any one time.

FORMAT

SET MAX_SOURCE_FILES *n*

PARAMETERS

n

Specifies the maximum number of source files that the debugger may keep open at any one time. The value of *n* may not exceed 20. The default value is 5.

QUALIFIERS

None.

DESCRIPTION

By default, the debugger may keep five source files open at any one time.

Opening a source file requires the use of an I/O channel, which is a limited system resource. Both the program and the debugger use I/O channels. To ensure that the debugger does not use all available I/O channels and thus cause the program to fail (for lack of an available I/O channel), you can issue the SET MAX_SOURCE_FILES command to specify the maximum number of source files (and thus source file I/O channels) that the debugger may use at any one time.

Note that the value of MAX_SOURCE_FILES does not limit the number of source files that the debugger can open; rather, it limits the number that may be kept open at any one time. Thus, if the debugger reaches this limit, it must close a file in order to open another one.

Note too that setting MAX_SOURCE_FILES to a very small number can make the debugger's use of source files inefficient.

Related commands: SHOW MAX_SOURCE_FILES, (SET, SHOW, CANCEL) SOURCE.

EXAMPLE

```
DBG> SHOW MAX_SOURCE_FILES
max_source_files: 5
DBG> SET MAX_SOURCE_FILES 8
DBG> SHOW MAX_SOURCE_FILES
max_source_files: 8
```

The SET MAX_SOURCE_FILES 8 command enables the debugger to keep a maximum of eight files open at any one time.

SET MODE

Enables or disables a debugger mode.

FORMAT

SET MODE *mode[, . . .]*

PARAMETERS

mode

Specifies a debugger mode to be enabled or disabled. Valid keywords are the following:

DYNAMIC	Enables dynamic module setting. In dynamic module setting, the debugger sets certain modules automatically during program execution so that you do not have to issue the SET MODULE command for these modules. Whenever the debugger prompt is displayed (whenever the debugger interrupts execution), the debugger automatically sets the module enclosing the PC location and issues an informational message. If the module is already set, dynamic module setting has no effect. SET MODE DYNAMIC is the default.
NODYNAMIC	Disables dynamic module setting. This may be desirable if performance becomes a problem as more and more modules are set. When dynamic module setting is disabled, you must set modules yourself with the SET MODULE command.
G_FLOAT	Specifies that the debugger interpret double-precision floating-point constants entered in expressions as G_FLOAT (does not affect the interpretation of variables declared in your program). EXAMINE/D_FLOAT and DEPOSIT/D_FLOAT may be used to override SET MODE G_FLOAT for the duration of an EXAMINE or DEPOSIT command.
NOG_FLOAT	Specifies that the debugger interpret double-precision floating-point constants entered in expressions as D_FLOAT (does not affect the interpretation of variables declared in your program). EXAMINE/G_FLOAT and DEPOSIT/G_FLOAT may be used to override SET MODE NOG_FLOAT for the duration of an EXAMINE or DEPOSIT command. SET MODE NOG_FLOAT is the default.
KEYPAD	Enables keypad mode. When keypad mode is enabled, you can use the keys on the numeric keypad to perform certain predefined functions. Several debugger commands, especially useful in screen mode, are bound to the keypad keys (see Appendix B). You can also redefine the key functions with the DEFINE/KEY command. SET MODE KEYPAD is the default.
NOKEYPAD	Disables keypad mode. When keypad mode is disabled, the keys on the numeric keypad do not have predefined functions, nor can you assign debugger functions to those keys with the DEFINE/KEY command.
LINE	Specifies that the debugger display code locations in terms of line numbers, if possible. SET MODE LINE is the default.

SET MODE

NOLINE	Specifies that the debugger display code locations in terms of routine + byte-offset rather than in terms of line numbers.
SCREEN	Enables screen mode. When screen mode is enabled, you can divide the terminal screen into rectangular regions, so different data can be displayed in different regions. Screen mode lets you view more information more conveniently than the default, line-oriented, noscreen mode. You can use the predefined displays, or you can define your own.
NOSCREEN	Disables screen mode. SET MODE NOSCREEN is the default.
SCROLL	Enables scroll mode. When scroll mode is enabled, a screen-mode output or DO display is updated by scrolling the output line by line, as it is generated. SET MODE SCROLL is the default.
NOSCROLL	Disables scroll mode. When scroll mode is disabled, a screen-mode output or DO display is updated only once per command, instead of line by line as it is generated. Disabling scroll mode reduces the amount of screen updating that takes place and may be useful with slow terminals.
SYMBOLIC	Enables symbolic mode. When symbolic mode is enabled, the debugger displays the locations denoted by address expressions symbolically (if possible) and displays instruction operands symbolically (if possible). EXAMINE/NOSYMBOLIC may be used to override SET MODE SYMBOLIC for the duration of an EXAMINE command. SET MODE SYMBOLIC is the default.
NOSYMBOLIC	Disables symbolic mode. When symbolic mode is disabled, the debugger does not attempt to symbolize numeric addresses (it does not cause the debugger to convert names to numbers). This may speed up command processing. Numeric addresses are displayed in decimal radix by default. The SET RADIX command may be used to specify a different radix. EXAMINE/SYMBOLIC may be used to override SET MODE NOSYMBOLIC for the duration of an EXAMINE command.

QUALIFIERS

None.

DESCRIPTION

The default values of these modes are the same for all languages.

Related commands: (SHOW, CANCEL) MODE, EXAMINE, DEPOSIT, (SET, SHOW, CANCEL) RADIX.

EXAMPLE

```
DBG> SET MODE SCREEN
```

This command puts the debugger in screen mode.

SET MODULE

Loads the symbol records of a module into the debugger's run-time symbol table (RST).

FORMAT

SET MODULE *[module-name[, . . .]]*

PARAMETERS

module-name

Specifies a module whose symbol records are to be loaded into the RST. Do not use the wildcard character (*). When using /ALL, do not specify a module name.

QUALIFIERS

/ALL

Specifies that the symbol records of all modules in the current image be loaded into the RST. When using /ALL, do not specify a module name.

/ALLOCATE

Note: Since memory is allocated automatically during dynamic module setting, you need to use /ALLOCATE only when dynamic module setting is disabled (SET MODE NODYNAMIC).

Expands the debugger memory pool. The debugger will then expand its memory pool by some increment of P0 space each time its internal memory pool is exhausted. The debugger allocates additional memory by calling $EXPREG. If your program also allocates memory dynamically, the location of the allocated memory may be affected. In this case, you may want to avoid the /ALLOCATE qualifier.

/[NO]RELATED

Note: /[NO]RELATED applies only to Ada programs.

Controls whether the debugger loads into the RST the symbol records of a module that is related to a specified module through a **with**-clause or subunit relationship.

SET MODULE/RELATED (default) loads symbol records for related modules as well as for those specified. This makes names declared in related modules visible so you can reference them in debugger commands as you could reference them within the Ada source code. SET MODULE/NORELATED loads symbol records only for modules that are specified (no symbol records are loaded for related modules).

SET MODULE

DESCRIPTION

Note: The (SET, SHOW, CANCEL) MODULE commands operate on modules in the current image. This is either the main image (by default) or the image established as the current image by a previous SET IMAGE command.

Symbol records must be present in the RST if the debugger is to recognize and properly interpret the symbols declared in your program. The process by which the symbol records of a module are loaded into the RST is called *setting a module*.

At debugger start up, the debugger sets the module containing the transfer address (the main program). By default, dynamic module setting is enabled (SET MODE DYNAMIC). Therefore, the debugger sets modules automatically as the program executes so that you can reference symbols as you need them. Specifically, whenever execution is suspended, the debugger sets the module containing the currently executing routine. In the case of Ada programs, as a module is set dynamically, its related modules are also set automatically, by default, to make the appropriate symbols accessible (visible).

Dynamic module setting makes accessible most of the symbols you might need to reference. If you need to reference a symbol in an arbitrary module that is not already set, use the SET MODULE command.

If dynamic module setting is disabled (SET MODE NODYNAMIC), only the module containing the transfer address is set for you. You need to set other modules yourself with the SET MODULE command.

When dynamic module setting is enabled, memory is allocated automatically to accommodate the increasing size of the RST. If dynamic module setting is disabled, the debugger may not let you set more modules unless you either allocate more memory (with the ALLOCATE or SET MODULE /ALLOCATE command) or reduce the number of set modules (with the CANCEL MODULE command). Whether dynamic module setting is enabled or disabled, if performance becomes a problem as more modules are set, use the CANCEL MODULE command to reduce the number of set modules.

If a parameter in the SET SCOPE command designates a program location in a module that is not already set, the SET SCOPE command sets that module.

Related commands: (SHOW, CANCEL) MODULE, SET MODE [NO]DYNAMIC, (SET, SHOW, CANCEL) IMAGE, ALLOCATE.

EXAMPLES

1
```
DBG> SET MODULE SUB1
```

This command sets module SUB1 (loads the symbol records of module SUB1 into the RST).

2
```
DBG> SET MODULE/ALL
%DEBUG-W-NOFREE, no free storage available
DBG> SET MODULE/ALL/ALLOCATE
```

The SET MODULE/ALL command attempts to set all of the modules in the current image. The debugger responds that there is not sufficient memory. The SET MODULE/ALL/ALLOCATE expands the memory pool as needed to accommodate all of the modules.

```
3    DBG> SET IMAGE SHARE3
     DBG> SET MODULE MATH
     DBG> SET BREAK %LINE 31
```

The SET IMAGE command makes shareable image SHARE3 the current image. The SET MODULE command sets module MATH in image SHARE3. The SET BREAK command sets a breakpoint on line 31 of module MATH.

SET OUTPUT

Enables or disables a debugger output option.

FORMAT

SET OUTPUT *output-option[, . . .]*

PARAMETERS

output-option

Specifies an output option to be enabled or disabled. Valid keywords are the following:

LOG	Specifies that debugger input and output be recorded in a log file. If you specify the log file by the SET LOG command, the debugger writes to that file; otherwise, by default the debugger writes to SYS$DISK[ ]:DEBUG.LOG.
NOLOG	Specifies that debugger input and output not be recorded in a log file. NOLOG is the default.
SCREEN_LOG	Specifies that, while in screen mode, the screen contents be recorded in a log file as the screen is updated. To log the screen contents you must also specify SET OUTPUT LOG. See the description of the LOG option regarding specifying the log file.
NOSCREEN_LOG	Specifies that the screen contents, while in screen mode, not be recorded in a log file. NOSCREEN_LOG is the default.
TERMINAL	Specifies that debugger output be displayed at the terminal. TERMINAL is the default.
NOTERMINAL	Specifies that debugger output, except for diagnostic messages, not be displayed at the terminal.
VERIFY	Specifies that the debugger echo, on the current output device, each input command string that it is executing from a command procedure or DO clause. The current output device is by default SYS$OUTPUT, the terminal, but may be redefined with the logical name DBG$OUTPUT.
NOVERIFY	Specifies that the debugger not display each input command string that it is executing from a command procedure or DO clause. NOVERIFY is the default.

QUALIFIERS

None.

DESCRIPTION

Debugger output options control the way in which debugger responses to commands are displayed and recorded.

Related commands: SHOW OUTPUT, (SET, SHOW) LOG, SET MODE SCREEN, @file-spec, (SET, SHOW) ATSIGN.

EXAMPLE

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

This command specifies that the debugger do the following:

- Output each command string that it is executing from a command procedure or DO clause.
- Record debugger output and user input in a log file.
- Not display output at the terminal (except for diagnostic messages).

SET PROMPT

Lets you change the debugger prompt string from DBG> to a string of your choice.

FORMAT

SET PROMPT *string*

PARAMETERS

string

Specifies the string which is to become the new prompt. If the string contains blanks, semicolons, or lowercase characters, you must enclose it in single or double quotation marks.

QUALIFIERS

None.

EXAMPLE

```
DBG> SET PROMPT "$ "
$ SET PROMPT "d b g : "
d b g : SET PROMPT "DBG> "
DBG>
```

The successive SET PROMPT commands change the debugger prompt from "DBG> " to "$", to "d b g :", then back to "DBG> ".

SET RADIX

Establishes the default radix for the entry and/or display of integer data. When used with /OVERRIDE, causes all data to be displayed as integer data of the specified radix.

FORMAT

SET RADIX *radix*

PARAMETERS

radix

Specifies the default radix to be established. Valid keywords are the following:

BINARY	Sets the default radix to binary.
DECIMAL	Sets the default radix to decimal. This is the default for all languages except BLISS and MACRO.
DEFAULT	Sets the default radix to the language default.
OCTAL	Sets the default radix to octal.
HEXADECIMAL	Sets the defaut radix to hexadecimal. This is the default for BLISS and MACRO.

QUALIFIERS

/INPUT

Sets only the input radix to the specified radix.

/OUTPUT

Sets only the output radix to the specified radix.

/OVERRIDE

Causes all data to be displayed as integer data of the specified radix.

DESCRIPTION

When you use the EXAMINE, DEPOSIT, or EVALUATE commands, the default radix influences how integer data is interpreted and displayed. By default, the radix is hexadecimal for BLISS and MACRO and decimal for other languages.

The SET RADIX command lets you change the default radix for the entry and/or display of integer data (the input radix and output radix, respectively). The debugger will interpret and display integer data in the new radix. However, other values (such as floating or enumeration type values) will be interpreted and displayed as they normally would be.

The SET RADIX/OVERRIDE command causes *all* data (not just data that was originally typed as integer data) to be displayed as integer data of the specified radix.

Note that the EVALUATE, EXAMINE, and DEPOSIT commands have radix qualifiers that let you further override, for the duration of that command, any default radix previously established with the SET RADIX or SET RADIX /OVERRIDE command.

SET RADIX

Related commands: (SHOW, CANCEL) RADIX, (SET, SHOW, CANCEL) MODE, EVALUATE, EXAMINE, DEPOSIT.

EXAMPLES

1
```
DBG> SET RADIX HEX
```

This command sets the default radix to hexadecimal. This means that, by default, integer data will be interpreted and displayed in hexadecimal radix.

2
```
DBG> SET RADIX/INPUT OCT
```

This command sets the default radix for input to octal. This means that, by default, integer data that is entered will be interpreted in octal radix.

3
```
DBG> SET RADIX/OUTPUT BIN
```

This command sets the default radix for output to binary. This means that, by default, integer data will be displayed in binary radix.

4
```
DBG> SET RADIX/OVERRIDE DECIMAL
```

This command sets the override radix to decimal. This means that, by default, all data will be displayed as decimal integer data.

SET SCOPE

Establishes how the debugger looks up symbols when a path-name prefix is not specified.

FORMAT SET SCOPE *location[, . . .]*

PARAMETERS *location*

Denotes a program region to be used for the interpretation of symbols that do not have a path-name prefix. A location may be any of the following:

path-name prefix — Specifies the scope region denoted by the path-name prefix. A path-name prefix consists of the names of one or more nesting program elements (module, routine, block, and so on), with each name separated by a backslash character (\). When a path-name prefix consists of more than one name, list a nesting element to the left of the \ and a nested element to the right of the \. A common path-name prefix format is *module\routine\block*.

If you specify only a module name and that name is the same as the name of a routine, use the /MODULE qualifier; otherwise, the debugger assumes that you are specifying the routine.

n — Specifies the scope region denoted by the routine which is *n* levels down the call stack (*n* is a decimal integer). A scope region specified by an integer changes dynamically as the program executes. The value *0* denotes the routine that is currently executing, the value *1* denotes the caller of that routine, and so on down the call stack. The default scope is *0,1,2, . . . ,N*, where *N* is the number of calls in the call stack.

\ — Specifies the global scope region—that is, the set of all program locations in which a global symbol is known. The definition of a global symbol and the way it is declared depends on the language.

When you specify more than one location parameter, you establish a scope search list. If the debugger cannot interpret the symbol using the first parameter, it uses the next parameter, and continues using parameters in order of their specification until it successfully interprets the symbol or until it exhausts the parameters specified.

QUALIFIERS */MODULE*

Indicates that the name specified is the name of a module and not of a routine. You need to use /MODULE only when you specify a module name as the scope region, and that module name is the same as the name of a nested routine.

SET SCOPE

DESCRIPTION

By default, the debugger looks up a symbol specified without a path-name prefix according to the scope search list *0,1,2, . . . ,N*, where *N* is the number of calls in the call stack. This scope search list is based on your current PC and changes dynamically as your program executes. The default scope means that a symbol lookup such as "EXAMINE X" first looks for X in the routine that is currently executing (scope 0); if no X is visible there, the debugger looks in the caller of that routine (scope 1), and so on down the call stack; if X is not found in scope N, the debugger searches the rest of the run-time symbol table (RST), then searches the global symbol table (GST), if necessary.

The SET SCOPE command lets you change this default symbol lookup. This is useful if, for example, you need to use a path name repeatedly to access a multiply-defined symbol. By specifying that path-name prefix in the SET SCOPE command, you establish a new default scope for symbol lookup. You can then reference the symbol without using a path-name prefix. Note that, when you use the SET SCOPE command, the debugger searches only the program locations you specify explicitly.

If you specify a module name in a SET SCOPE command, the debugger "sets" that module if it is not already set. However, if all you want to do is set a module, it is best to use the SET MODULE command rather than disturb the current scope search list with the SET SCOPE command.

If a name you specify is the name of both a module and a nested routine, the debugger sets the scope to the routine, unless you use the /MODULE qualifier to indicate that you want to set the scope to the module.

To restore the default scope, use the CANCEL SCOPE command.

Related commands: (SHOW, CANCEL) SCOPE, SET MODULE.

EXAMPLES

1

```
DBG> EXAMINE Y
%DEBUG-W-NOUNIQUE, symbol 'Y' is not unique
DBG> SHOW SYMBOL Y
    data CHECK_IN\Y
    data INVENTORY\COUNT\Y
DBG> SET SCOPE INVENTORY\COUNT
DBG> EXAMINE Y
INVENTORY\COUNT\Y: 347.15
```

The first EXAMINE Y command indicates that symbol Y is multiply defined and cannot be resolved from the current scope search list. The SHOW SYMBOL command displays the different declarations of symbol Y. The SET SCOPE command tells the debugger to look for symbols without path-name prefixes in routine COUNT of module INVENTORY. The subsequent EXAMINE command can now interpret Y unambiguously.

2

```
DBG> SET SCOPE 1
```

This command tells the debugger to look for symbols without path-name prefixes in scope 1, which is the caller of the routine that is currently executing. If the debugger cannot find a symbol in scope 1, it looks no further.

```
3   DBG> SHOW SYMBOL X
    data ALPHA\X                    ! global X
    data ALPHA\BETA\X          ! local X
    data X (global)                ! same as ALPHA\X
    DBG> SHOW SCOPE
    scope: 0 [ = ALPHA\BETA ]
    DBG> SYMBOLIZE X
    address ALPHA\BETA\%RO:
        ALPHA\BETA\X
    DBG> SET SCOPE \
    DBG> SYMbOLIZE X
    address 00000200:
        ALPHA\X
    address 00000200: (global)
        X
```

The SHOW SYMBOL command indicates that there are two declarations of the symbol X—a global ALPHA\X (shown twice) and a local ALPHA\BETA\X. Within the current scope, the local declaration of X (ALPHA\BETA\X) is visible. After the scope is set to the global scope (SET SCOPE \), the global declaration of X is made visible.

SET SEARCH

Establishes default qualifiers (/ALL or /NEXT, /IDENTIFIER or /STRING) for the SEARCH command.

FORMAT

SET SEARCH *search-default[, . . .]*

PARAMETERS

search-default

Specifies a default to be established for the SEARCH command. Valid keywords (which correspond to SEARCH command qualifiers) are the following:

ALL	Subsequent SEARCH commands will be treated as SEARCH/ALL, rather than SEARCH/NEXT.
IDENTIFIER	Subsequent SEARCH commands will be treated as SEARCH /IDENTIFIER, rather than SEARCH/STRING.
NEXT	Subsequent SEARCH commands will be treated as SEARCH /NEXT, rather than SEARCH/ALL. This is the default.
STRING	Subsequent SEARCH commands will be treated as SEARCH /STRING, rather than SEARCH/IDENTIFIER. This is the default.

QUALIFIERS

None.

DESCRIPTION

The SET SEARCH command establishes default qualifiers for subsequent SEARCH commands. The parameters that you specify in the SET SEARCH command have the same names as the SEARCH command qualifiers. SEARCH command qualifiers determine whether the SEARCH command: (1) searches for all occurrences (ALL) of a string or only the next occurrence (NEXT); and (2) displays any occurrence of the string (STRING) or only those occurrences in which the string is not bounded on either side by a character that can be part of an identifier in the current language (IDENTIFIER).

You can override the current SEARCH default for the duration of a single SEARCH command by specifying other qualifiers. Use the SHOW SEARCH command to identify the current SEARCH defaults.

Related commands: SEARCH, SHOW SEARCH, (SET, SHOW) LANGUAGE.

EXAMPLE

```
DBG> SHOW SEARCH
search settings: search for next occurrence, as a string
DBG> SET SEARCH IDENTIFIER
DBG> SHOW SEARCH
search settings: search for next occurrence, as an identifier
DBG> SET SEARCH ALL
DBG> SHOW SEARCH
search settings: search for all occurrences, as an identifier
```

The SET SEARCH IDENTIFIER command tells the debugger to search for an occurrence of the string in the specified range but display the string only if it is not bounded on either side by a character that can be part of an identifier in the current language.

The SET SEARCH ALL command tells the debugger to search for (and display) all occurrences of the string in the specified range.

SET SOURCE

Specifies where the debugger is to search for source files.

FORMAT

SET SOURCE *directory-spec[, . . .]*

PARAMETERS

directory-spec

Specifies any part of a VAX/VMS file specification (typically a device /directory) that the debugger is to use by default when searching for a source file. For any part of a full file specification that you do not supply, the debugger uses the file specification stored in the module's symbol record—that is, the file specification that the source file had at compile time.

If you specify more than one directory in a single SET SOURCE command, separating each directory name with a comma, you create a source directory search list (you may also specify a search list logical name that is defined at your process level). The debugger handles a source directory search list by searching the first directory specified to locate the source file for a module, then the second directory specified, then the next, and so on, until it either locates the source file or exhausts the list of directories.

QUALIFIERS

/EDIT

Note: /EDIT applies mainly to Ada programs.

Specifies that the directory search list will be used to locate source files for editing when you use the EDIT command.

/MODULE=module-name

Specifies that the directory search list will be used to locate source files *only* for the specified module.

DESCRIPTION

By default, the debugger expects a source file to be in the same directory it was in at compile time (the debugger also checks that the creation and revision date and time of a source file match the information in the debugger's symbol table). If a source file has been moved to a different directory since compile time, you can use the SET SOURCE command to specify a source directory search list.

If you issue the SET SOURCE command without the /MODULE=module-name qualifier, the debugger uses the specified directory search list to locate source files for all modules that were not mentioned in a previous SET SOURCE/MODULE=module-name command.

See the qualifier descriptions for an explanation of their effects.

The /EDIT qualifier is needed when the files used for the display of source code are different from the files to be edited by means of the EDIT command. This is the case with Ada programs. For Ada programs, the (SET, SHOW, CANCEL) SOURCE commands affect the search of files used for source display (the "copied" source files in Ada program libraries); the (SET, SHOW, CANCEL) SOURCE/EDIT commands affect the search of the source files you edit when using the EDIT command. If you use /MODULE with /EDIT, the effect of /EDIT is further qualified by /MODULE.

Related commands: (CANCEL, SHOW) SOURCE, (CANCEL, SHOW) MAX_SOURCE_FILES.

EXAMPLES

1
```
DBG> SHOW SOURCE
no directory search list in effect
DBG> SET SOURCE [PROJA],[PROJB],USER$:[PETER.PROJC]
DBG> SHOW SOURCE
source directory search list for all modules:
        [PROJA]
        [PROJB]
        USER$:[PETER.PROJC]
```

The SET SOURCE command specifies that the debugger should search directories [PROJA], [PROJB], and USER$:[PETER.PROJC] for source files.

2
```
DBG> SET SOURCE/MODULE=COBOLTEST DISK$2:[PROJD],[014,015]
DBG> SHOW SOURCE
source directory search list for COBOLTEST:
        DISK$2:[PROJD]
        [014,015]
source directory search list for all other modules:
        [PROJA]
        [PROJB]
        USER$:[PETER.PROJC]
```

The SET SOURCE command specifies that the debugger should search directories DISK$2:[PROJD] and [014,015]) for source files to use with the module COBOLTEST. The SHOW SOURCE command displays the search lists established in examples 1 and 2.

SET STEP

Establishes default qualifiers (/LINE, /INTO, and so on) for the STEP command.

FORMAT

SET STEP *step-default[, . . .]*

PARAMETERS

step-default

Specifies a default to be established for the STEP command. Valid keywords (which correspond to STEP command qualifiers) are the following:

BRANCH	Subsequent STEP commands will be treated as STEP/BRANCH (step to the next branch instruction).
CALL	Subsequent STEP commands will be treated as STEP/CALL (step to the next call instruction).
EXCEPTION	Subsequent STEP commands will be treated as STEP /EXCEPTION (step to the next exception condition).
INSTRUCTION	Subsequent STEP commands will be treated as STEP /INSTRUCTION (step to the next instruction). This is the default for MACRO. You can also specify one or more instructions (INSTRUCTION=(opcode-list)). The debugger will then step to the next instruction that is in the specified list.
INTO	Subsequent STEP commands will be treated as STEP/INTO (step into called routines) rather than STEP/OVER (step over called routines). When INTO is in effect, you can qualify the types of routines to step into by means of the [NO]JSB, [NO]SHARE, and [NO]SYSTEM parameters, or by using the STEP/[NO]JSB, STEP/[NO]SHARE, and STEP/[NO]SYSTEM command/qualifier combinations (the latter three take effect only for the immediate STEP command).
JSB	If INTO is in effect, subsequent STEP commands will be treated as STEP/INTO/JSB (step into routines called by a JSB instruction as well as those called by a CALL instruction). This is the default for all languages except DIBOL.
NOJSB	If INTO is in effect, subsequent STEP commands will be treated as STEP/INTO/NOJSB (step over routines called by a JSB instruction, but step into routines called by a CALL instruction). This is the default for DIBOL.
LINE	Subsequent STEP commands will be treated as STEP/LINE (step to the next line). This is the default for all languages except MACRO.
OVER	Subsequent STEP commands will be treated as STEP/OVER (step over all called routines) rather than STEP/INTO (step into called routines). SET STEP OVER is the default.
RETURN	Subsequent STEP commands will be treated as STEP/RETURN (step to the RETURN instruction of the current routine). Thus, STEP/RETURN *n* will take you up *n* levels of the call stack.

SHARE	If INTO is in effect, subsequent STEP commands will be treated as STEP/INTO/SHARE (step into called routines in shareable images as well as into other called routines). This is the default.
NOSHARE	If INTO is in effect, subsequent STEP commands will be treated as STEP/INTO/NOSHARE (step over called routines in shareable images, but step into other routines).
SILENT	Subsequent STEP commands will be treated as STEP/SILENT (suppress the "stepped to . . . " message as well as other debugger output).
NOSILENT	Subsequent STEP commands will be treated as STEP /NOSILENT (display the "stepped to . . . " message as well as other output). This is the default.
SOURCE	Subsequent STEP commands will be treated as STEP/SOURCE (display source code after a step as well as when a breakpoint or watchpoint is activated). This is the default.
NOSOURCE	Subsequent STEP commands will be treated as STEP /NOSOURCE (do not display source code after a step or when a breakpoint or watchpoint is activated).
SYSTEM	If INTO is in effect, subsequent STEP commands will be treated as STEP/INTO/SYSTEM (step into called routines in system space (P1 space) as well as into other called routines). This is the default.
NOSYSTEM	If INTO is in effect, subsequent STEP commands will be treated as STEP/INTO/NOSYSTEM (step over called routines in system space, but step into other routines).

QUALIFIERS

None.

DESCRIPTION

The SET STEP command establishes default qualifiers for subsequent STEP commands. The parameters that you specify in the SET STEP command have the same names as the STEP command qualifiers. The following parameters affect the locations to which you step:

```
SET STEP BRANCH
SET STEP CALL
SET STEP EXCEPTION
SET STEP INSTRUCTION
SET STEP INSTRUCTION=(opcode-list)
SET STEP LINE
SET STEP RETURN
```

The following parameters affect what output is seen when a STEP command is executed:

```
SET STEP [NO]SILENT
SET STEP [NO]SOURCE
```

The following parameters affect what happens at a routine call:

```
SET STEP INTO
SET STEP [NO]JSB
SET STEP OVER
SET STEP [NO]SHARE
```

SET STEP

SET STEP [NO]SYSTEM

Each language establishes certain defaults for the STEP command. When you change the current language, STEP defaults are set to those specified in that language.

You can override the current STEP defaults for the duration of a single STEP command by specifying other qualifiers. Use the SHOW STEP command to identify the current STEP defaults.

Related commands: STEP, SHOW STEP, (SET, SHOW) LANGUAGE.

EXAMPLES

1
```
DBG> SET STEP INSTRUCTION,NOSOURCE
```

This command tells the debugger to step by instruction when a STEP command is issued, and to not display lines of source code with each STEP command.

2
```
DBG> SET STEP LINE,INTO,NOSYSTEM,NOSHARE
```

This command tells the debugger to step by line when a STEP command is issued, and to step into called routines in user space only. The debugger will step over routines in system space and in shareable images.

SET TASK

Modifies characteristics of one or more tasks or of the entire tasking system.

Note: The SET TASK command currently applies only to Ada programs. See the VAX Ada documentation for complete information.

FORMAT

SET TASK *[task-expression[, . . .]]*

PARAMETERS

task-expression

Specifies a task value. A task expression may be:

- An Ada language expression for a task value—for example, a task object name. You can use a path name.
- The task ID (for example, %TASK 2), as indicated in a SHOW TASK display.
- A pseudo-task name (%ACTIVE_TASK, %CALLER_TASK, %NEXT_TASK, or %VISIBLE_TASK).

Do not use the wildcard character (*). See the qualifier descriptions for details on how to specify tasks with particular qualifiers.

QUALIFIERS

/ABORT

Aborts the specified tasks. If no task is specified, aborts the visible task. Note that the task is marked for termination but is not immediately terminated. The effect is identical to executing the Ada statement **abort** task-name, and causes the specified tasks to become *abnormal*.

/ACTIVE

Makes the specified task the active task—the task that will run when a STEP or GO command is executed. Causes a task switch to the new active task and makes the new active task the visible task. The specified task must be in either the RUNNING or READY state. When using /ACTIVE, you must specify one, and only one, task.

/ALL

Applies the SET TASK command to all tasks. When you specify /ALL, you cannot specify a task, nor can you specify the /ACTIVE, /VISIBLE, or /TIME_SLICE qualifiers.

/[NO]HOLD

Controls whether or not a specified task is placed on HOLD. /HOLD places a specified task on HOLD. If no task is specified, /HOLD places the visible task on HOLD.

Placing a task on HOLD prevents a task from entering the RUNNING state. A task placed on HOLD is allowed to make other state transitions; in particular, it may change from the SUSPENDED to the READY state.

A task that is already in the RUNNING state (the active task) can continue to execute as long as it remains in the RUNNING state, even though it is placed on HOLD. If the task leaves the RUNNING state for any reason (including expiration of a time slice, if timeslicing is enabled), it may not return to the RUNNING state until the HOLD is removed. You can force a task into the RUNNING state with the SET TASK/ACTIVE command even if the task is on HOLD.

/NOHOLD removes a specified task from HOLD. If no task is specified, /NOHOLD removes the visible task from HOLD.

/PRIORITY=n

Sets the priority of a specified task to *n*, where *n* is a decimal integer from 0 to 15 inclusive. If no task is specified, sets the priority of the visible task to *n*. Note that this does not prevent the task's priority from later changing in the course of execution, for example, while executing a rendezvous.

/RESTORE

Causes the priority of a specified task to be restored to the value specified in **pragma** PRIORITY. If **pragma** PRIORITY was not specified, the default value of 7 is used. If no task is specified, causes the priority of the visible task to be restored.

/TIME_SLICE=t

Sets the duration otherwise specified by **pragma** TIME_SLICE to the value *t*, where *t* is a decimal integer or fixed-point value representing seconds. The SET TASK/TIME_SLICE=0.0 command disables time slicing.

/VISIBLE

Makes the specified task the visible task—the task whose stack and register set are the current context for looking up names, calls, and so on (commands such as EXAMINE are directed at the visible task). When using /VISIBLE, you must specify one, and only one, task.

Note: If no qualifier is specified, /VISIBLE is assumed by default.

DESCRIPTION

The possible task states are RUNNING, READY, SUSPENDED, and TERMINATED.

All of the SET TASK command qualifiers except for /ALL provide a means of controlling the tasking environment, by directly or indirectly causing task state transitions. The /ALL qualifier is used to apply the SET TASK command to all tasks.

Task switching can often be confusing when you are trying to debug a program. The SET TASK/TIME_SLICE and SET TASK/HOLD commands give you several ways of controlling task switching.

Related commands: SHOW TASK, SET BREAK/EVENT, SET TRACE /EVENT, EXAMINE/TASK, DEPOSIT/TASK.

EXAMPLES

1
```
DBG> SET TASK/ACTIVE %TASK 3
```

This command makes the task whose task ID is %TASK 3 the active task.

2
```
DBG> SET TASK/HOLD/ALL
DBG> SET TASK/ACTIVE %TASK 1
DBG> GO
   .
   .
   .
DBG> SET TASK/ACTIVE %TASK 3
DBG> STEP
   .
   .
   .
```

The SET TASK/HOLD/ALL command freezes the state of all tasks except the active task. The SET TASK/ACTIVE command is then used selectively (along with the GO command) to observe the behavior of one or more specified tasks in isolation.

SET TERMINAL

Sets the terminal screen width and/or height that the debugger uses when it formats screen and other output.

FORMAT

SET TERMINAL

PARAMETERS

None.

QUALIFIERS

You must specify at least one qualifier, either /PAGE or /WIDTH. You can specify both /PAGE and /WIDTH. You must specify a value for each qualifier used.

/PAGE:n

Specifies that the terminal screen height should be set to *n* lines. You may use any value from 18 to 100.

/WIDTH:n

Specifies that the terminal screen width should be set to *n* columns. You may use any value from 20 to 255. For a VT100 or VT200 series terminal, *n* is typically either 80 or 132.

DESCRIPTION

The SET TERMINAL command lets you define the portion of the screen that the debugger has available for formatting screen output. This command is particularly useful with VT100 or VT200 series terminals, where you may set the screen width to typically 80 or 132 columns. It is also useful with MicroVAX workstations, where you can increase the screen size beyond the default 24-line by 80-column DEBUG window.

When you issue the SET TERMINAL command, all screen window definitions (including those created by the user) are automatically adjusted for the new screen dimensions. For example, RH1 will change dimensions proportionally to remain the top right half of the screen.

Similarly, all "dynamic" displays are automatically adjusted to maintain their relative dimensions. By default, all predefined and user-defined displays are dynamic, except for register displays. If you have specified /NODYNAMIC in a SET DISPLAY or DISPLAY command, the display is no longer dynamic. In that case, the display will not automatically change dimensions with a SET TERMINAL command. However, you can always use the DISPLAY command to redisplay the display within any window definition (you can also use keypad-key combinations, such as BLUE-MINUS, to enter predefined DISPLAY commands).

Related commands: SHOW TERMINAL, DISPLAY/[NO]DYNAMIC, SET DISPLAY/[NO]DYNAMIC, (SET, SHOW, CANCEL) WINDOW, EXPAND.

EXAMPLE

```
DBG> SET TERMINAL/WIDTH:132
```

This command specifies that the terminal screen width be set to 132 columns.

SET TRACE

Establishes a tracepoint at the location denoted by an address-expression, or at instructions of a particular class.

FORMAT

SET TRACE *[addr-expr[, . . .]] [***WHEN***(cond-expr)] [***DO***(command[; . . .])]*

PARAMETERS

addr-expr

Specifies an address expression (a program location) at which a tracepoint is to be set. In general, this may be a line number, a routine name, a label, or a location in memory. However, the /MODIFY and /RETURN qualifiers are used with specific kinds of address expressions. Do not use the wildcard character (*). Do not use an address expression when specifying /BRANCH, /CALL, /EXCEPTION, /INSTRUCTION[=(opcode-list)], /INTO, /[NO]JSB, /LINE, /OVER, /[NO]SHARE, or /[NO]SYSTEM.

command

Specifies a debugger command that is to be executed as part of the DO clause when trace action is taken.

cond-expr

Specifies a conditional expression in the currently set language that is to be evaluated every time the tracepoint occurs. If the expression is TRUE, trace action occurs, and the debugger reports that a tracepoint has been reached. If the expression is FALSE, trace action does not occur. In this case, a report is not issued, the commands specified by the DO clause are not executed, and program execution is continued.

QUALIFIERS

/AFTER:n

Specifies that trace action not be taken until the *n*th time the designated tracepoint is encountered (*n* is a decimal integer). Thereafter, the tracepoint occurs every time it is encountered provided that conditions in the WHEN clause are TRUE. The command SET TRACE/AFTER:1 has the same effect as the SET TRACE command.

/BRANCH

Causes the debugger to trace every branch instruction encountered (including 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, ACBG, ACBH, AOBLEQ, AOBLSS, SOBGEQ, SOBGTR, CASEB, CASEW, CASEL) during execution. Do not specify an address expression with this qualifier. See also /INTO, /OVER.

/CALL

Causes the debugger to trace every call instruction (including the CALLS, CALLG, BSBW, BSBB, JSB, RSB, and RET instructions) encountered during execution. Do not specify an address expression with this qualifier. See also /INTO, /OVER.

/EVENT=event-name

Note: /EVENT applies only to Ada and SCAN. See the VAX Ada and VAX SCAN documentation for complete information.

Causes the debugger to trace every time the specified event occurs (if that event is defined and detected by the run-time system). If you specify an address expression with /EVENT, causes the debugger to trace every time the specified event occurs for that address expression. Event names depend on the run-time facility and are identified in Appendix E for Ada and SCAN. Note that you cannot specify an address expression with certain event names.

/EXCEPTION

Causes the debugger to trace every exception that is signaled. The trace action occurs before any user-written exception handlers are invoked. Do not specify an address expression with this qualifier.

/INSTRUCTION

Causes the debugger to trace every instruction executed. Do not specify an address expression with this qualifier. See also /INTO, /OVER.

/INSTRUCTION=(opcode[, . . .])

Causes the debugger to trace every instruction whose opcode is in the list. Do not specify an address expression with this qualifier. See also /INTO, /OVER.

/INTO

Sets tracepoints within called routines (as well as within the main program) when /BRANCH, /CALL, /INSTRUCTION=[(opcode-list)], or /LINE is specified; that is, when an address expression is not explicitly specified. /INTO is the default behavior and is the opposite of /OVER. When using /INTO, you can further qualify the tracepoints with the /[NO]JSB, /[NO]SHARE, and /[NO]SYSTEM qualifiers.

/[NO]JSB

Qualifies /INTO. Use /[NO]JSB with /INTO and one of these qualifiers: /BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE. /JSB is the default for all languages except DIBOL. /JSB lets the debugger set tracepoints within routines that are called by the JSB or CALL instruction. /NOJSB (the DIBOL default) specifies that tracepoints not be set within routines called by JSB instructions. In DIBOL, user-written routines are called by the CALL instruction and DIBOL run-time library routines are called by the JSB instruction. Do not specify an address expression with this qualifier.

/LINE

Causes the debugger to trace the start of each new line. Do not specify an address expression with this qualifier. See also /INTO, /OVER.

/MODIFY

Causes the debugger to report a tracepoint whenever an instruction writes to and modifies the value of a location indicated by a specified address expression. The address expression is typically a variable name.

The SET TRACE/MODIFY command acts like a SET WATCH command followed by a GO command. It operates under the same restrictions as the SET WATCH command.

If you specify an absolute address for the address expression, the debugger may not be able to associate the address with a particular data object. In this case, the debugger uses a default length of 4 bytes. You can change this length, however, by setting the type to either WORD (which changes the default length to 2 bytes) or BYTE (which changes the default length to 1 byte).

/OVER

Sets tracepoints only within the main program (not within called routines) when /BRANCH, /CALL, /INSTRUCTION=[(opcode-list)], or /LINE is specified; that is, when an address expression is not explicitly specified. /OVER is the opposite of /INTO.

/RETURN

Sets a tracepoint on the RETURN (RET) instruction from an indicated routine. This qualifier can only be applied to routines called with a CALLS or CALLG instruction; it cannot be used with JSB routines.

For this qualifier, the address-expression parameter is an instruction address within a CALLS or CALLG routine. It may simply be a routine name, in which case it specifies the routine start address. However, you can also specify another location in a routine, so you can see only those returns that are taken after a certain code path is followed.

/[NO]SHARE

Qualifies /INTO. Use /[NO]SHARE with /INTO and one of these qualifiers: BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE. /SHARE (default) lets the debugger set a tracepoint within shareable image routines as well as other routines. /NOSHARE specifies that tracepoints not be set within shareable images. Do not specify an address expression with this qualifier.

/[NO]SILENT

Controls whether or not the "trace . . . " message (and source code) is displayed when trace action is taken. /NOSILENT (default) specifies that the message be displayed. /SILENT specifies that no message or source code be displayed. /SILENT overrides /SOURCE.

/[NO]SOURCE

Controls whether or not the source code is displayed when trace action is taken. /SOURCE (default) specifies that the source code be displayed. /NOSOURCE specifies that no source code be displayed. /SILENT overrides /SOURCE.

/[NO]SYSTEM

Qualifies /INTO. Use /[NO]SYSTEM with /INTO and one of these qualifiers: /BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE. /SYSTEM (default) lets the debugger set tracepoint within system routines (P1 space) as well as other routines. /NOSYSTEM specifies that tracepoints not be set within system routines. Do not specify an address expression with this qualifier.

/TEMPORARY

Causes the tracepoint to disappear after it is activated (the tracepoint does not remain permanently set).

DESCRIPTION

When a tracepoint is activated, the debugger takes the following action:

1. Suspends program execution at the tracepoint location.
2. Evaluates the expression in a WHEN clause, if one was specified when the tracepoint was set. If the value of the expression is FALSE, execution continues and the debugger does not perform the next two steps.
3. Reports that execution has reached the tracepoint location, unless /SILENT was specified.
4. Executes the commands in a DO clause, if one was specified when the tracepoint was set.
5. Resumes execution.

The following qualifiers affect what output is seen when a tracepoint is reached:

/[NO]SILENT
/[NO]SOURCE

The following qualifiers affect the timing and duration of tracepoints:

/AFTER:n
/TEMPORARY

Tracepoints may be set on classes of instructions by using one of the following qualifiers:

/BRANCH
/CALL
/EVENT=event-name
/EXCEPTION
/INSTRUCTION
/INSTRUCTION=(opcode-list)
/LINE
/RETURN

The following qualifiers affect what happens at a routine call:

/INTO
/[NO]JSB
/OVER
/[NO]SHARE
/[NO]SYSTEM

The /MODIFY qualifier is used to monitor changes at program locations (typically changes in the values of variables).

Related commands: (SHOW, CANCEL) TRACE, CANCEL ALL, SET BREAK, SET WATCH, GO, (SET, SHOW) EVENT_FACILITY.

EXAMPLES

1
```
DBG> SET TRACE SUB1
```

This command sets a tracepoint at location (routine) SUB1.

2
```
DBG> SET TRACE/SILENT COUNTER WHEN (A = B) DO (EXAMINE Y)
```

This command sets a tracepoint on routine COUNTER that will trigger only when A equals B. When the tracepoint is triggered, variable Y is examined. The /SILENT qualifier suppress the "trace . . . " message.

3
```
DBG> SET TRACE/BRANCH
```

This command causes the debugger to trace every branch instruction.

4
```
DBG> SET TRACE/LINE/INTO/NOSHARE/NOSYSTEM
```

This command causes the debugger to trace every line, including lines in called routines, but not in shareable image routines or system routines.

SET TYPE

Establishes the default type to be associated with program locations that do not have a compiler generated type. When used with /OVERRIDE, establishes the default type to be associated with all locations, overriding any compiler generated types.

FORMAT

SET TYPE *type-keyword*

PARAMETERS

type-keyword

Specifies the default type to be established. Valid keywords are the following:

ASCIC	Sets the default type to counted ASCII string. AC is also accepted as a keyword.
ASCID	Sets the default type to ASCII string descriptor. The CLASS and DTYPE fields of the descriptor are not checked, but the LENGTH and POINTER fields provide the character length and address of the ASCII string. The string is then displayed. AD is also accepted as a keyword.
ASCII:n	Sets the default type to ASCII character string (length *n* bytes). The length indicates both the number of bytes of memory to be examined and the number of ASCII characters to be displayed. If you do not specify a value for *n*, the debugger uses the default value of 4 bytes. The value *n* is interpreted in decimal radix.
ASCIW	Sets the default type to ASCII varying string (a 2-byte length field immediately followed by an ASCII string of that length). This data type occurs in PASCAL and PL/I. AW is also accepted as a keyword.
ASCIZ	Sets the default type to zero-terminated ASCII string. AZ is also accepted as a keyword.
BYTE	Sets the default type to byte integer (length 1 byte).
DATE_TIME	Sets the default type to date-time: a quadword integer (length 64 bits) containing the internal VAX/VMS representation of date-time.
D_FLOAT	Sets the default type to D_floating (length 8 bytes). Values of type D_floating may range from $.29*10^{-38}$ to $1.7*10^{38}$ with approximately 16 decimal digits precision.
FLOAT	Sets the default type to F_floating (length 4 bytes). Values of type F_floating may range from $.29*10^{-38}$ to $1.7*10^{38}$ with approximately 7 decimal digits precision.
G_FLOAT	Sets the default type to G_floating (length 8 bytes). Values of type G_floating may range from $.56*10^{-308}$ to $.9*10^{308}$ with approximately 15 decimal digits precision.

SET TYPE

H_FLOAT	Sets the default type to H_floating (length 16 bytes). Values of type H_floating may range from $.84*10^{-4932}$ to $.59*10^{4932}$ with approximately 33 decimal digits precision.
INSTRUCTION	Sets the default type to VAX instruction (variable length, depending on the number of instruction operands and the kind of addressing modes used).
LONGWORD	Sets the default type to longword integer (length 4 bytes). This the default for all languages.
OCTAWORD	Sets the default type to octaword integer (length 16 bytes).
PACKED:n	Sets the default type to packed decimal (length 4n bits).
QUADWORD	Sets the default type to quadword integer (length 8 bytes).
TYPE=(expression)	Sets the default type to the type denoted by *expression*. *Expression* may be the name of an existing program variable or data type.
WORD	Sets the default type to word integer (length 2 bytes).

QUALIFIERS

/OVERRIDE

Associates the type specified with *all* program locations, not just those that have a compiler-generated type.

DESCRIPTION

When you use the EXAMINE, DEPOSIT, or EVALUATE commands, the default types associated with address expressions influence how program entities are interpreted and displayed.

The debugger recognizes any compiler generated types associated with program locations and interprets and displays the contents of these locations accordingly. For program locations that do not have a compiler generated type, the default type in all languages is longword integer.

The SET TYPE command lets you change the default type associated with untyped program locations. The SET TYPE/OVERRIDE command lets you set a default type for *all* program locations, not just untyped locations.

Note that the EXAMINE and DEPOSIT commands have type qualifiers that let you further override, for the duration of that command, any type previously established with the SET TYPE or SET TYPE/OVERRIDE command.

Related commands: SHOW TYPE, CANCEL TYPE/OVERRIDE, (SET, SHOW, CANCEL) RADIX, (SET, SHOW, CANCEL) MODE.

EXAMPLES

```
1    DBG> SET TYPE ASC:8
```

This command establishes 8-byte ASCII character string as the default type associated with untyped program locations.

2 `DBG> SET TYPE/OVERRIDE LONGWORD`

This command establishes longword integer as the default type associated with both untyped program locations and program locations that have compiler-generated types.

3 `DBG> SET TYPE D_FLOAT`

This command establishes D_Floating as the default type associated with untyped program locations.

4 `DBG> SET TYPE TYPE=(S_ARRAY)`

This command establishes the type of the variable S_ARRAY as the default type associated with untyped program locations.

SET WATCH

Establishes a watchpoint at the location denoted by an address-expression.

FORMAT

SET WATCH *[addr-expr[, . . .]] [*
WHEN*(cond-expr)] [*
DO*(command[; . . .])]*

PARAMETERS

addr-expr

Specifies an address expression (a program location) at which a watchpoint is to be set. For high-level languages, this is typically the name of a variable. Do not use the wildcard character (*).

command

Specifies a debugger command that is to be executed as part of the DO clause when watch action is taken.

cond-expr

Specifies a conditional expression in the currently set language that is to be evaluated every time the watchpoint occurs. If the expression is TRUE, watch action occurs, and the debugger reports that a watchpoint has been triggered. If the expression is FALSE, watch action does not occur. In this case, a report is not issued, the commands specified by the DO clause are not executed, and program execution is continued.

QUALIFIERS

/AFTER:n

Specifies that watch action not be taken until the *n*th time the designated watchpoint is encountered (*n* is a decimal integer). Thereafter, the watchpoint occurs every time it is encountered provided that conditions in the WHEN clause are TRUE. The command SET WATCH/AFTER:1 has the same effect as the SET WATCH command.

/[NO]SILENT

Controls whether or not the "watch . . . " message (and source code) is displayed when watch action is taken. /NOSILENT (default) specifies that the message be displayed. /SILENT specifies that no message or source code be displayed. /SILENT overrides /SOURCE.

/[NO]SOURCE

Controls whether or not the source code is displayed when watch action is taken. /SOURCE (default) specifies that the source code be displayed. /NOSOURCE specifies that no source code be displayed. /SILENT overrides /SOURCE.

/TEMPORARY

Causes the watchpoint to disappear after it is activated (the watchpoint does not remain permanently set).

DESCRIPTION

Whenever an instruction causes the modification of a watched location, the debugger takes the following action:

1. Suspends program execution after that instruction has completed execution.
2. Evaluates the expression in a WHEN clause, if one was specified when the watchpoint was set. If the value of the expression is FALSE, execution continues and the debugger does not perform the following steps.
3. Reports that execution has reached the watchpoint location, unless /SILENT was specified.
4. Reports the value at the watched location before modification.
5. Reports the new or modified value at the watched location.
6. Identifies the instruction that modified the watched location. Also, displays the line of source code corresponding to that instruction if the SOURCE parameter is in effect by virtue of a previous SET STEP SOURCE command.
7. Executes the commands in a DO clause, if one was specified when the watchpoint was set. If the DO clause contains a GO command, execution continues and the debugger does not perform the next step.
8. Issues the DBG> prompt.

If the watched location has a compiler-generated type, the debugger uses the length in bytes associated with that type to determine the length in bytes of the watched location. If the watched location does not have a compiler-generated type, the debugger watches four bytes of virtual memory beginning at the byte identified by the address expression.

You can set watchpoints on aggregates (that is, entire arrays or records). A watchpoint set on an array or record will trigger if any element of the array or record changes. Thus, you do not need to set watchpoints on individual array elements or record components. Note, however, that you cannot set an aggregate watchpoint on a variant record.

You cannot set a watchpoint on a variable that is on the stack or that is allocated to a register.

Related commands: (SHOW, CANCEL) WATCH, SET BREAK, SET TRACE.

EXAMPLES

```
1   DBG> SET WATCH MAXCOUNT
```

This command establishes a watchpoint at location MAXCOUNT.

SET WATCH

```
2   DBG> SET WATCH ARR
    DBG> GO
         .
         .
         .
    watch of SUBR\ARR at SUBR\%LINE 12+8
       old value:
        (1):          7
        (2):          12
        (3):          3

       new value:
        (1):          7
        (2):          12
        (3):          28

    break at SUBR\%LINE 14
```

In this example, the SET WATCH command sets a watchpoint on the three-element integer array, ARR. Execution is then resumed with the GO command. The watchpoint is triggered whenever any array element changes. In this case the third element changed.

SET WINDOW

Creates a screen window definition.

FORMAT

SET WINDOW *wname* **AT** *(start-line,line-count [,start-col,col-count])*

PARAMETERS

wname

Specifies the name of the window you are defining. If a window definition with that name already exists, it is canceled in favor of the new definition.

start-line

Specifies the starting line number of the window. This line displays the window title, or header line. The top line of the screen is line 1.

line-count

Specifies the number of text lines in the window, not counting the header line. *Line-count* must be at least 1. The sum of *start-line* and *line-count* must not exceed the current screen height.

start-col

Specifies the starting column number of the window. This is the column at which the first character of the window will be displayed. The leftmost column of the screen is column 1.

col-count

Specifies the number of characters per line in the window. *Col-count* must be at least 1. The sum of *start-col* and *col-count* must not exceed the current screen width.

QUALIFIERS

None.

DESCRIPTION

A screen window is a rectangular region on the terminal screen through which you may view a display. The SET WINDOW command establishes a window definition by associating a window name with a screen region. You specify the screen region in terms of a starting line and height (line count) and, optionally, a starting column and width (column count). If you do not specify the starting column and column count, they default to column 1 and the current screen width.

You can specify a window region in terms of expressions that use the built-in symbols %PAGE and %WIDTH.

You can use the names of any windows you have defined with the SET WINDOW command in DISPLAY and SET DISPLAY commands to position displays on the screen.

SET WINDOW

Window definitions are "dynamic"—that is, window dimensions expand and contract proportionally when a SET TERMINAL command changes the screen width or height.

Related commands: (SHOW, CANCEL) WINDOW, (SET SHOW, CANCEL) DISPLAY, DISPLAY, (SET, SHOW) TERMINAL.

EXAMPLES

1
```
DBG> SET WINDOW ONELINE AT (1,1)
```

This command defines a window named ONELINE at the top of the screen. The window is one line deep and, by default, spans the width of the screen.

2
```
DBG> SET WINDOW MIDDLE AT (9,4,30,20)
```

This command defines a window named MIDDLE at the middle of the screen. The window is 4 lines deep starting at line 9, and 20 columns wide starting at column 30.

3
```
DBG> SET WINDOW FLEX AT (%PAGE/4,%PAGE/2,%WIDTH/4,%WIDTH/2)
```

This command defines a window named FLEX that occupies a region around the middle of the screen and is defined in terms of the current screen height (%PAGE) and width (%WIDTH).

SHOW AST

Indicates whether delivery of ASTs is enabled or disabled.

FORMAT

SHOW AST

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The SHOW AST command indicates whether delivery of ASTs is enabled or disabled. Note that the command does not identify an AST whose delivery is pending. The delivery of ASTs is enabled by default and with the ENABLE AST command. The delivery of ASTs is disabled with the DISABLE AST command.

Related commands: (ENABLE, DISABLE) AST.

EXAMPLE

```
DBG> SHOW AST
ASTs are enabled
DBG> DISABLE AST
DBG> SHOW AST
ASTs are disabled
DBG>
```

The SHOW AST command indicates whether the delivery of ASTs is enabled.

SHOW ATSIGN

Identifies the default file specification established with the last SET ATSIGN command. The debugger uses this file specification when processing the @file-spec command.

FORMAT SHOW ATSIGN

PARAMETERS *None.*

QUALIFIERS *None.*

DESCRIPTION Related commands: SET ATSIGN, @file-spec.

EXAMPLES

1
```
DBG> SHOW ATSIGN
No indirect command file default in effect, using DEBUG.COM
```

If the SET ATSIGN command was not used, the debugger assumes that command procedures have the default file specification SYS$DISK:[]DEBUG.COM.

2
```
DBG> SET ATSIGN USER:[JONES.DEBUG].DBG
DBG> SHOW ATSIGN
Indirect command file default is USER:[JONES.DEBUG].DBG
```

The SHOW ATSIGN command indicates the default file specification for command procedures, as previously established with the SET ATSIGN command.

SHOW BREAK

Displays information about all breakpoints established by the SET BREAK command, including WHEN and DO clauses and /AFTER counts.

FORMAT

SHOW BREAK

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The debugger displays all information about each breakpoint that is currently set, including any optional WHEN and DO clauses.

If you established a breakpoint using the /AFTER:n command qualifier with the SET BREAK command, the SHOW BREAK command displays the current value of the decimal integer *n*, that is, the originally specified integer value minus one for each time the breakpoint location was reached. (The debugger decrements *n* each time the breakpoint location is reached until the value of *n* is zero, at which time the debugger takes break action.)

Related commands: (SET, CANCEL) BREAK.

EXAMPLE

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

This command displays information about the three breakpoints currently set, SUB1\LOOP, MAIN\MAIN, and SUB2\SUB2.

SHOW CALLS

Identifies the currently active routine calls (the call stack).

FORMAT

SHOW CALLS *[n]*

PARAMETERS

n

Specifies the number of call frames to be identified. If *n* is omitted, all currently active call frames are identified.

QUALIFIERS

None.

DESCRIPTION

The SHOW CALLS command shows a traceback that lists the sequence of routine calls leading to the currently executing routine. One line of information is displayed for each call frame, starting with the most recent call. The top line identifies the currently executing routine, the next line identifies its caller, the following line identifies the caller of the caller, and so on.

The following information is provided for each call frame:

- The name of the enclosing module.
- The name of the calling routine, provided the module is set (the first line shows the currently executing routine).
- The line number where the call was made in that routine, provided the module is set (the first line shows the line number where execution is suspended).
- The value of the PC in the calling routine at the time that control was transferred to the called routine. The PC value is shown as a virtual address relative to the virtual address of the name of the routine and also as an absolute virtual address.

Related commands: SHOW STACK, SHOW SCOPE.

EXAMPLE

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

This command displays information about the sequence of currently active procedure calls.

An asterisk next to a module name indicates that the module is set.

SHOW DEFINE

Identifies the default qualifier (/ADDRESS, /COMMAND, or /VALUE) currently in effect for the DEFINE command.

FORMAT

SHOW DEFINE

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The default qualifier for the DEFINE command is the default qualifier last established with the SET DEFINE command. If no SET DEFINE command was issued, the default qualifier is /ADDRESS.

To identify a symbol defined with the DEFINE command, use the SHOW SYMBOL/DEFINED command.

Related commands: SET DEFINE, DEFINE, DELETE, SHOW SYMBOL /DEFINED.

EXAMPLE

```
DBG> SHOW DEFINE
Current setting is: DEFINE/ADDRESS
DBG>
```

The SHOW DEFINE command indicates that the DEFINE command is set for definition by address.

SHOW DISPLAY

Identifies one or more existing screen displays.

FORMAT

SHOW DISPLAY *[disp-name[, . . .]]*

PARAMETERS

disp-name

Specifies the name of a display. If you do not specify a name, or if you specify the wildcard character (*) by itself, all display definitions are listed. You can use * within a display name. When using /ALL, do not specify a display name.

QUALIFIERS

/ALL

Lists all display definitions. When using /ALL, do not specify a display name.

DESCRIPTION

The SHOW DISPLAY command lists all displays according to their order in the display list. The most hidden display is listed first, and the display that is on top of the display pasteboard is listed last.

For each display, the SHOW DISPLAY command lists its name, maximum size, screen window, and display kind (including any debug command list). It also identifies whether or not the display is removed from the pasteboard or is dynamic (a dynamic display automatically adjusts its window dimensions if the screen size is changed with the SET TERMINAL command).

Related commands: (SET, CANCEL) DISPLAY, DISPLAY, (SET, CANCEL, SHOW WINDOW), SHOW SELECT, EXTRACT/SCREEN_LAYOUT.

EXAMPLE

```
DBG> SHOW DISPLAY
display SRC at H1, size = 64, dynamic
    kind = SOURCE (EXAMINE/SOURCE .%SOURCE_SCOPE\%PC)
display INST at H1, size = 64, removed, dynamic
    kind = INSTRUCTION (EXAMINE/INSTRUCTION .0\%PC)
display REG at RH1, size = 64, removed, not dynamic, kind = REGISTER
display OUT at S45, size = 100, dynamic, kind = OUTPUT
display EXSUM at Q3, size = 64, dynamic, kind = DO (EXAMINE SUM)
display PROMPT at S6, size = 64, dynamic, kind = PROGRAM
```

The SHOW DISPLAY command lists all displays currently defined. These include the five predefined displays (SRC, INST, REG, OUT, and PROMPT), and the user-defined DO display EXSUM. Displays INST and REG are removed from the display pasteboard: the DISPLAY command must be used in order to display them on the screen.

SHOW EDITOR

Indicates the action taken by the EDIT command, as established by the SET EDITOR command.

FORMAT SHOW EDITOR

PARAMETERS *None.*

QUALIFIERS *None.*

DESCRIPTION Related commands: SET EDITOR, EDIT.

EXAMPLES

1
```
DBG> SHOW EDITOR
The editor is SPAWNed, with command line "LSEDIT/START_POSITION=(n,1)"
```

The SHOW EDITOR command indicates that, when you issue the EDIT command, you will spawn the VAX Language-Sensitive Editor in a subprocess. The /START_POSITION qualifier that is appended to the command line indicates that the editing cursor will be initially positioned at the start of the line that is cented in the debugger's current source display.

2
```
DBG> SET EDITOR/CALLABLE_TPU
DBG> SHOW EDITOR
The editor is CALLABLE_TPU, with command line "TPU"
```

The SHOW EDITOR command indicates that, when you issue the EDIT command, you will invoke the callable version of the VAX Text Processing Utility (VAXTPU). The editing cursor will be initially positioned at the start of source line 1.

SHOW EVENT_FACILITY

Identifies the current run-time facility for eventpoints and the associated event names.

Note: The SHOW EVENT_FACILITY command currently applies only to Ada and SCAN. See the VAX Ada and VAX SCAN documentation for complete information.

FORMAT

SHOW EVENT_FACILITY

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The SHOW EVENT_FACILITY command is meaningful only with Ada or SCAN programs. The command identifies the current run-time facility and lists the associated event names that may be used with the SET BREAK /EVENT and SET TRACE/EVENT commands. The event names associated with the Ada and SCAN run-time facilities are identified in Appendix E.

Related commands: SET EVENT_FACILITY, (SET, CANCEL) BREAK /EVENT, SHOW BREAK, (SET, CANCEL) TRACE/EVENT, SHOW TRACE.

EXAMPLES

```
1   DBG> SHOW EVENT_FACILITY
    event facility is ADA
        .
        .
        .
```

This command identifies the current event facility to be Ada and lists the associated event names that may be used with a SET BREAK/EVENT or SET TRACE/EVENT command.

SHOW EXIT_HANDLERS

Identifies the exit handlers that have been declared in your program.

FORMAT

SHOW EXIT_HANDLERS

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The exit handler routines are displayed in the order that they will be called (that is, last in, first out). The routine name is displayed symbolically, if possible. Otherwise, its address is displayed. The debugger's exit handlers are not displayed.

EXAMPLE

```
DBG> SHOW EXIT_HANDLERS
exit handler at STACKS\CLEANUP
```

The SHOW EXIT_HANDLERS command identifies the exit handler routine CLEANUP, which is declared in module STACKS.

SHOW IMAGE

Displays information about one or more shareable images that are part of your running program.

FORMAT

SHOW IMAGE *[image-name]*

PARAMETERS

image-name

Specifies the name of a shareable image to be included in the display. If you do not specify a name, or if you specify the wildcard character (*) by itself, all images are listed. You can use * within an image name.

QUALIFIERS

None.

DESCRIPTION

The SHOW IMAGE command displays the following information:

- The name of the shareable image
- The start and end addresses of the image
- Whether the image has been "set" with the SET IMAGE command (loaded into the RST)
- The current image that is your debugging context (marked with an asterisk)
- The total number of images selected in the display

Related commands: (SET, CANCEL) IMAGE, (SET, SHOW, CANCEL) MODULE.

EXAMPLES

```
DBG> SHOW IMAGE SHARE*
 image name                      set    base address    end address

*SHARE                           yes    00000200        00000FFF
 SHARE1                          no     00001000        000017FF
 SHARE2                          yes    00018C00        000191FF
 SHARE3                          no     00019200        000195FF
 SHARE4                          no     00019600        0001B7FF

 total images: 5                remaining size: 33032
```

This SHOW IMAGE command identifies all of the shareable images whose names start with "SHARE" and which are associated with the program. Images SHARE and SHARE2 are set. The asterisk identifies SHARE as the current image.

SHOW KEY

Displays the key definitions created by the DEFINE/KEY command.

FORMAT SHOW KEY *[key-name]*

PARAMETERS *key-name*

Specifies a function key whose definition is to be displayed. Do not use the wildcard character (*). When using /ALL, do not specify a key name. Valid key names are the following:

Key-name	LK201	VT100-type	VT52-type
PF1	PF1	PF1	Blue
PF2	PF2	PF2	Red
PF3	PF3	PF3	Black
PF4	PF4	PF4	
KP0, KP1, . . . ,KP9	Keypad 0, . . . ,9	Keypad 0, . . . ,9	Keypad 0, . . . ,9
PERIOD	Keypad period (.)	Keypad period (.)	
COMMA	Keypad comma (,)	Keypad comma (,)	
MINUS	Keypad minus (-)	Keypad minus (-)	
ENTER	ENTER	ENTER	ENTER
E1	Find		
E2	Insert Here		
E3	Remove		
E4	Select		
E5	Prev Screen		
E6	Next Screen		
HELP	Help		
DO	Do		
F6, F7, . . . , F20	F6, F7, . . . , F20		

QUALIFIERS

/ALL

Displays all key definitions for the current state, by default, or for the states specified with the /STATE qualifier. When using /ALL, do not specify a key name.

/BRIEF

Displays only the key definitions (by default, all the qualifiers associated with a key definition are also shown, including any specified state)

/DIRECTORY

Displays the names of all the states for which keys have been defined. Do not specify other qualifiers with /DIRECTORY.

/[NO]STATE=(state-name [, . . .])

Selects one or more states for which a key definition is to be displayed. /STATE displays key definitions for the specified states. You may specify predefined key states, such as DEFAULT and GOLD, or user-defined states. A state name can be any appropriate alphanumeric string. /NOSTATE (default) displays key definitions for the current state only.

DESCRIPTION

Keypad mode must be enabled (SET MODE KEYPAD) before you can use this command. Keypad mode is enabled by default.

By default, the current key state is the "DEFAULT" state. The current state may be changed with the SET KEY/STATE command, or by pressing a key that causes a state change (a key that was defined with the DEFINE/KEY /LOCK_STATE/STATE qualifier combination).

Related commands: DEFINE/KEY, DELETE/KEY, SET KEY.

EXAMPLES

1
```
DBG> SHOW KEY/ALL
```

This example shows how to display all the keys currently defined in the current state.

2
```
DBG> SHOW KEY/STATE=BLUE KP8
GOLD keypad definitions:
  KP8 = "Scroll/Top" (noecho,terminate,nolock)
```

This example shows how to display the definition for keypad key 8.

3
```
DBG> SHOW KEY/BRIEF KP8
DEFAULT keypad definitions:
  KP8 = "Scroll/Up"
```

This example shows how to display the definition for keypad key 8 without any associated qualifiers or states.

4
```
DBG> SHOW KEY/DIRECTORY
MOVE_GOLD
MOVE_BLUE
MOVE
GOLD
EXPAND_GOLD
EXPAND_BLUE
EXPAND
DEFAULT
CONTRACT_GOLD
CONTRACT_BLUE
CONTRACT
BLUE
```

This command displays the names of the states for which keys have been defined.

SHOW LANGUAGE

Identifies the current language.

FORMAT SHOW LANGUAGE

PARAMETERS *None.*

QUALIFIERS *None.*

DESCRIPTION

The current language is the language last established with the SET LANGUAGE command. If no SET LANGUAGE command was issued, the current language is, by default, the language established at debugger start up (the language of the module containing the transfer address).

Related commands: SET LANGUAGE.

EXAMPLE

```
DBG> SHOW LANGUAGE
language: BASIC
```

This command displays the name of the current language as BASIC.

SHOW LOG

Identifies the current log file and reports whether the debugger is writing to that file.

FORMAT

SHOW LOG

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The current log file is the log file last established by a SET LOG command. If no SET LOG command was issued, the current log file is the file SYS$DISK:[]DEBUG.LOG, by default.

Related commands: SET LOG, SET OUTPUT [NO]LOG, SET OUTPUT [NO]SCREEN_LOG.

EXAMPLES

1

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

This command displays the name of the current log file as DEBUG.LOG (the default log file) and reports that the debugger is not writing to it.

2

```
DBG> SET LOG PROG4
DBG> SET OUTPUT LOG
DBG> SHOW LOG
logging to USER$:[JONES.WORK]PROG4.LOG
```

The SET LOG command establishes that the current log file is PROG4.LOG (in the current default directory). The SET OUTPUT LOG command causes the debugger to log debugger input and output into that file. The SHOW LOG command confirms that the debugger is writing to the log file PROG4.COM in the current default directory.

SHOW MARGINS

Displays the current source-line margin settings for the display of source code.

FORMAT

SHOW MARGINS

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The current margin settings are the margin settings last established with the SET MARGINS command. If no SET MARGINS command was issued, the left margin is set to 1 and the right margin is set to 255 by default.

Related commands: SET MARGINS.

EXAMPLES

1
```
DBG> SHOW MARGINS
left margin: 1 , right margin: 255
```

This command displays the default margin settings of 1 and 255.

2
```
DBG> SET MARGINS 50
DBG> SHOW MARGINS
left margin: 1 , right margin: 50
```

This command displays the default left margin setting of 1 and the modified right margin setting of 50.

3
```
DBG> SET MARGINS 10:60
DBG> SHOW MARGINS
left margin: 10 , right margin: 60
```

This command displays both margin settings modified to 10 and 60.

SHOW MAX_SOURCE_FILES

Displays the maximum number of source files that the debugger may keep open at any one time.

FORMAT

SHOW MAX_SOURCE_FILES

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The maximum number of source files that the debugger may keep open at any one time may be specified using the SET MAX_SOURCE_FILES command. If no SET MAX_SOURCE_FILES command was issued, the maximum number of files is 5, by default.

Related commands: SET MAX_SOURCE_FILES, (SET, SHOW, CANCEL) SOURCE.

EXAMPLE

```
DBG> SHOW MAX_SOURCE_FILES
max_source_files: 7
```

This command shows that the debugger may keep a maximum of 7 source files open at any one time.

SHOW MODE

Identifies the current debugger modes (screen or no screen, keypad or nokeypad, and so on) and the current radix.

FORMAT

SHOW MODE

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The current debugger modes are the modes last established with the SET MODE command. If no SET MODE command was issued, the current modes are, by default: DYNAMIC, NOG_FLOAT (d_float), KEYPAD, LINE, NOSCREEN, SCROLL, SYMBOLIC.

Related commands: (SET, CANCEL) MODE, (SET, SHOW, CANCEL) RADIX.

EXAMPLE

```
DBG> SHOW MODE
modes: symbolic, line, d_float, screen, scroll, keypad, dynamic
input radix :decimal
output radix:decimal
```

The SHOW MODE command displays the current modes and current input and output radix.

SHOW MODULE

Displays information about one or more modules in your program.

FORMAT SHOW MODULE *[module-name]*

PARAMETERS *module-name*

Specifies the name of a module to be included in the display. If you do not specify a name, or if you specify the wildcard character (*) by itself, all modules are listed. You can use * within a module name. Shareable image modules are selected only if the /SHARE qualifier is specified.

QUALIFIERS */[NO]RELATED*

Note: /[NO]RELATED applies only to Ada programs.

Controls whether the debugger includes, in the SHOW MODULE display, any module that is related to a specified module through a **with**-clause or subunit relationship.

SHOW MODULE/RELATED displays related modules as well as those specified. The display identifies the exact relationship. By default (/NORELATED), no related modules are selected for display (only the modules specified are selected).

/[NO]SHARE

Controls whether the debugger includes, in the SHOW MODULE display, any shareable image modules that have been linked with your program but are external to your program. These shareable images are primarily run-time library images, but they also include any shareable images called by your program.

Dummy modules whose names have the prefix "SHARE$" are created for each shareable image in your program. SHOW MODULE/SHARE displays these shareable image modules, as well as modules in the main image. By default (/NOSHARE) no shareable image modules are selected for display.

Setting a shareable image module loads the universal symbols for that image into the run-time symbol table. Note that this feature overlaps the effect of the newer SET IMAGE and SHOW IMAGE commands.

DESCRIPTION

Note: The (SET, SHOW, CANCEL) MODULE commands operate on modules in the current image. This is either the main image (by default) or the image established as the current image by a previous SET IMAGE command.

The SHOW MODULE command displays the following information about one or more modules selected for display:

- The module name
- The language in which the module is written
- Whether or not the symbol records of the module have been loaded into the debugger's run-time symbol table (RST)
- The space (in bytes) required in the RST for symbol records in that module
- The total number of modules selected in the display
- The number of unused bytes in the space allocated for the RST

Related commands: (SET, CANCEL) MODULE, (SET, SHOW, CANCEL) IMAGE, SET MODE [NO]DYNAMIC, SHOW SYMBOL, (SET, SHOW, CANCEL) SCOPE.

EXAMPLE

```
DBG> SHOW MODULE FOO,MAIN,SUB*
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.
```

The SHOW MODULE command displays information about the modules FOO and MAIN, and all modules having the prefix SUB.

SHOW OUTPUT

Displays the current output options.

FORMAT

SHOW OUTPUT

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The current output options are the options last established with the SET OUTPUT command. If no SET OUTPUT command was issued, the output options are, by default: NOLOG, NOSCREEN_LOG, TERMINAL, NOVERIFY.

Related commands: SET OUTPUT, SET LOG, SET MODE SCREEN.

EXAMPLE

```
DBG> SHOW OUTPUT
noverify, terminal, screen_log, logging to USER$:[JONES.WORK]DEBUG.LOG;9
```

This command shows that the debugger is not displaying input command strings as it executes command procedures and commands in DO clauses; is displaying output on the terminal; is writing output to the log file USER$:[JONES.WORK]DEBUG.LOG;9; and is logging the screen contents as they are updated in screen mode.

SHOW RADIX

Displays the current radix for the entry and display of integer data or, if the /OVERRIDE command qualifier is specified, the current override radix.

FORMAT

SHOW RADIX

PARAMETERS

None.

QUALIFIERS

/OVERRIDE

Identifies the current override radix.

DESCRIPTION

The current radix for the entry and display of integer data is the radix last established with the SET RADIX command. If no SET RADIX command was issued, the radix for both input and output is hexadecimal for BLISS and MACRO and decimal for other languages.

The current override radix for the display of all data is the override radix last established with the SET RADIX/OVERRIDE command. If no SET RADIX /OVERRIDE command was issued, the override radix is "none".

Related commands: (SET, CANCEL) RADIX.

EXAMPLES

1

```
DBG> SHOW RADIX
input radix: decimal
output radix: decimal
```

This command identifies the input radix and output radix as decimal.

2

```
DBG> SET RADIX/OVERRIDE HEX
DBG> SHOW RADIX/OVERRIDE
output override radix: hexadecimal
```

The SET RADIX/OVERRIDE command sets the override radix to hexadecimal and the SHOW RADIX/OVERRIDE command indicates the override radix. This means that all data will be displayed as hexadecimal integer data in commands such as EXAMINE and so on.

SHOW SCOPE

Displays the current scope search list for symbol lookup.

FORMAT SHOW SCOPE

PARAMETERS *None.*

QUALIFIERS *None.*

DESCRIPTION

The current scope search list designates one or more program locations (specified by path names and/or other special characters) to be used in the interpretation of symbols that are specified without path-name prefixes in debugger commands.

The current scope search list is the scope search list last established with the SET SCOPE command. If no SET SCOPE command was issued, the current scope search list is *0,1,2, . . . ,N*, by default.

The default scope means that, for a symbol without a path-name prefix, a symbol lookup such as "EXAMINE X" first looks for X in the routine that is currently executing (scope 0); if no X is visible there, the debugger looks in the caller of that routine (scope 1), and so on down the call stack; if X is not found in scope N, the debugger searches the rest of the run-time symbol table (RST), then searches the global symbol table (GST), if necessary.

If you have used a decimal integer in the SET SCOPE command to represent a routine in the call stack, the SHOW SCOPE command displays the name of the routine represented by the integer, if possible.

Related commands: (SET, CANCEL) SCOPE.

EXAMPLE

```
DBG> SHOW SCOPE
scope: MAIN, SUB1, 0[=SUB2],\
```

This example shows how to search for a symbol X (EXAMINE X) in MAIN, then in SUB1, then in the routine that contains the current PC, and finally globally.

SHOW SEARCH

Identifies the default qualifiers (/ALL or /NEXT, /IDENTIFIER or /STRING) currently in effect for the SEARCH command.

FORMAT

SHOW SEARCH

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The default qualifiers for the SEARCH command are the default qualifiers last established with the SET SEARCH command. If no SET SEARCH command was issued, the default qualifiers are /NEXT and /STRING.

Related commands: SET SEARCH, SEARCH, (SET, SHOW) LANGUAGE.

EXAMPLE

```
DBG> SHOW SEARCH
search settings: search for next occurrence, as a string
DBG> SET SEARCH IDENT
DBG> SHOW SEARCH
search settings: search for next occurrence, as an identifier
DBG> SET SEARCH ALL
DBG> SHOW SEARCH
search settings: search for all occurrences, as an identifier
```

The first SHOW SEARCH command displays the default settings for the SET SEARCH command. By default, the debugger searches for and displays the next occurrence of the string.

The second SHOW SEARCH command indicates that the debugger will search for the next occurrence of the string, but will only display the string if it is not bounded on either side by a character that can be part of an identifier in the current language.

The third SHOW SEARCH command indicates that the debugger will search for all occurrences of the string, but will only display the strings if they are not bounded on either side by a character that can be part of an identifier in the current language.

SHOW SELECT

Identifies the displays currently selected for each of the display attributes: error, input, instruction, output, program, prompt, scroll, and source.

FORMAT SHOW SELECT

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The display attributes have the following properties:

- A display that has the *error* attribute displays debugger diagnostic messages.
- A display that has the *input* attribute echoes your debugger input.
- A display that has the *instruction* attribute displays the decoded assembly language instruction stream of the routine being debugged. The display is updated when you enter an EXAMINE/INSTRUCTION command.
- A display that has the *output* attribute displays any debugger output that is not directed to another display.
- A display that has the *program* attribute displays program input and output. Currently only the PROMPT display can have the program attribute.
- A display that has the *prompt* attribute is where the debugger prompts for input. Currently, only the PROMPT display can have the PROMPT attribute.
- A display that has the *scroll* attribute is the default display for the SCROLL, MOVE, and EXPAND commands.
- A display that has the *source* attribute displays the source code of the module being debugged, if available. The display is updated when you enter a TYPE or EXAMINE/SOURCE command.

Related commands: SELECT, SHOW DISPLAY.

EXAMPLE

```
DBG> SHOW SELECT
display selections:
     scroll = SRC
     input  = none
     output = OUT
     error  = PROMPT
     source = SRC
     instruction = none
     program = PROMPT
     prompt = PROMPT
```

The SHOW SELECT command identifies the displays currently selected for each of the display attributes. The display selections shown are the default selections for all languages except MACRO.

SHOW SOURCE

Displays the source directory search lists currently in effect.

FORMAT SHOW SOURCE

PARAMETERS *None.*

QUALIFIERS */EDIT*

Note: /EDIT applies mainly to Ada programs.

Identifies the search list for source files to be edited when you use the EDIT command.

DESCRIPTION

If a source directory search list has not been established by means of the SET SOURCE or SET SOURCE/MODULE=module-name commands, the SHOW SOURCE command indicates that no directory search list is currently in effect. In this case, the debugger expects each source file to be in the same directory that it was in at compile time (the debugger also checks that the creation date and time of a source file match the information in the debugger's symbol table).

The SET SOURCE/MODULE=module-name command establishes a source directory search list for a particular module. The SET SOURCE command establishes a source directory search list for all modules not explicitly mentioned in a SET SOURCE/MODULE=module-name command. When those commands have been used, the SHOW SOURCE command identifies the source directory search list associated with each search categories.

The /EDIT qualifier is needed when the files used for the display of source code are different from the files to be edited by means of the EDIT command. This is the case with Ada programs. For Ada programs, the SHOW SOURCE command identifies the search list of files used for source display (the "copied" source files in Ada program libraries); the SHOW SOURCE/EDIT command identifies the search list for the source files you edit when using the EDIT command.

Related commands: (SET, CANCEL) SOURCE, (SET, SHOW) MAX_SOURCE_FILES.

EXAMPLES

1

```
DBG> SHOW SOURCE
no directory search list in effect
DBG> SET SOURCE [PROJA],[PROJB],DISK:[PETER.PROJC]
DBG> SHOW SOURCE
source directory search list for all modules:
        [PROJA]
        [PROJB]
        DISK:[PETER.PROJC]
```

This example shows how the SET SOURCE command tells the debugger to search the directories [PROJA],[PROJB], and DISK:[PETER.PROJC].

2

```
DBG> SET SOURCE/MODULE=COBOLTEST DISK$2:[PROJD],[014,015]
DBG> SHOW SOURCE
source directory search list for COBOLTEST:
        DISK$2:[PROJD]
        [014,015]
source directory search list for all other modules:
        [PROJA]
        [PROJB]
        DISK:[PETER.PROJC]
```

This example shows how the SET SOURCE command tells the debugger to search the directories DEVICE:[PROJD] and [014,015]) for source files to use with the module COBOLTEST.

SHOW STACK

Displays information from the current call stack.

FORMAT

SHOW STACK *[n]*

PARAMETERS

n

Specifies the number of frames to display. If *n* is omitted, information about all stack frames is displayed.

QUALIFIERS

None.

DESCRIPTION

For each call frame, the SHOW STACK command displays information such as the condition handler, saved register values, and the argument list, if any. The latter is the list of arguments passed to the subroutine with that call. In some cases the argument list may contain the addresses of actual arguments. In such cases, "DBG> EXAMINE address" will return the values of these arguments.

Related commands: SHOW CALLS.

EXAMPLES

1

```
DBG> SHOW STACK
stack frame 0 (2146814812)

    condition handler: 0
       SPA:            0
       S:              0
       mask:           ^M<R2>
       PSW:            0000 (hexadecimal)
    saved AP:          7
    saved FP:          2146814852
    saved PC:          EIGHTQUEENS
    saved R2:          0
    argument list:(1)  EIGHTQUEENS

stack frame 1 (2146814852)

    condition handler: SHARE$PASRTL+888
       SPA:            0
       S:              0
       mask:           none saved
       PSW:            0000 (hexadecimal)
    saved AP:          2146814924
    saved FP:          2146814904
    saved PC:          SHARE$DEBUG+667
```

The SHOW STACK command displays information about all stack frames at the current PC location.

SHOW STEP

Identifies the default qualifiers (/INTO, /INSTRUCTION, /NOSILENT and so on) currently in effect for the STEP command.

FORMAT

SHOW STEP

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The default qualifiers for the STEP command are the default qualifiers last established by the SET STEP command. If no SET STEP command was issued, the default qualifiers are those established by the current language.

Related commands: SET STEP, STEP, (SET, SHOW) LANGUAGE.

EXAMPLE

```
DBG> SET STEP INTO,NOSYSTEM,NOSHARE,INSTRUCTION,NOSOURCE
DBG SHOW STEP
step type: nosystem, noshare, nosource, nosilent, into routine calls, by instruction
```

This SHOW STEP command indicates that the debugger will:

- Step into called routines, but not those in system space or in shareable images
- Step by instruction
- Not display lines of source code while stepping

SHOW SYMBOL

Displays symbols in the debugger's run-time symbol table.

FORMAT

SHOW SYMBOL *symbol-name* [**IN** *scope*[, . . .]]
[,*symbol-name* [**IN** *scope*[, . . .]]

PARAMETERS

symbol-name

Specifies a symbol to be identified. A valid symbol name is a single identifier or a label name of the form *%LABEL n*, where *n* is an integer. Compound names such as RECORD.FIELD or ARRAY[1,2] are not valid. If you specify the wildcard character (*) by itself, all symbols are listed. You can use * within a display name.

scope

Specifies the name of a module, routine, or lexical block, or a numeric scope. It has the same syntax as the scope specification in a SET SCOPE command and may include path-name qualification. All specified scopes must be in set modules.

The SHOW SYMBOL command displays only those symbols in the RST that both match the specified name and are declared within the lexical entity specified by the scope parameter. If the scope parameter is omitted, all set modules and the GST (global symbol table) are searched for symbols that match the name specified by the namespec parameter.

QUALIFIERS

/ADDRESS

Displays the address specification for each selected symbol. The address specification is the method of computing the symbol's address. It may merely be the symbol's virtual address, but it may also involve indirection or an offset from a register value. Some symbols have address specifications too complicated to present in any understandable way. These address specifications are labeled "complex address specifications."

/DEFINED

Displays symbols you have defined with the DEFINE command.

/DIRECT

Displays only those symbols that are declared directly in the scope parameter. Symbols declared in lexical entities nested within the scope specified by the scope parameters will not be shown.

/LOCAL

Displays symbols that are defined with the DEFINE/LOCAL command.

/TYPE

Displays data type information for each selected symbol.

/USE_CLAUSE

Note: /USE_CLAUSE applies only to Ada programs.

Identifies any Ada package that a specified block, subprogram, or package names in a **use** clause. If the symbol specified is a package, also identifies any block, subprogram, package, and so on that names the specified symbol in a **use** clause.

DESCRIPTION

The SHOW SYMBOL command displays information that the debugger has about a given symbol. This information may not be the same as what the compiler had or even what you see in your source code. Nonetheless, it is useful for understanding why the debugger may act as it does when handling symbols.

Related commands: DEFINE, SYMBOLIZE.

EXAMPLES

1
```
DBG> SHOW SYMBOL I
data FORARRAY\I
```

This command shows that symbol I is located in module FORARRAY.

2
```
DBG> SHOW SYMBOL INTARRAY1
data FORARRAY\INTARRAY1
```

This command shows that symbol INTARRAY1 is located in module FORARRAY.

3
```
DBG> SHOW SYMBOL/ADDRESS INTARRAY1
data FORARRAY\INTARRAY1
     descriptor address: 0009DE8B
```

This command shows that symbol INTARRAY1 is located in module FORARRAY and has a virtual address of 0009DE8B.

4
```
DBG> SHOW SYMBOL *tarr*
data FORARRAY\INTARRAY1
data FORARRAY\INTARRAY2
data FORARRAY\INTARRAY3
```

This command displays all the symbol names containing the string "tarr" and their locations in module FORARRAY.

5
```
DBG> SHOW SYMBOL/TYPE/ADDRESS *
```

This command displays all information about all symbols.

6
```
DBG> DEFINE/COMMAND SB=SET BREAK
DBG> SHOW SYMBOL/DEFINED SB
defined SB
    bound to: SET BREAK
    was defined /command
```

The DEFINE/COMMAND command defines SB as a symbol for the command SET BREAK. The SHOW SYMBOL/DEFINED command displays that definition.

SHOW TASK

Displays information about the tasks of a tasking program.

Note: The SHOW TASK command currently applies only to Ada programs. See the VAX Ada documentation for complete information.

FORMAT

SHOW TASK *[task-expression[, . . .]]*

PARAMETERS

task-expression

Specifies a task value. A task expression may be:

- An Ada language expression for a task value—for example, a task object name. You can use a path name.
- The task ID (for example, %TASK 2), as indicated in a SHOW TASK display.
- A pseudo-task name (%ACTIVE_TASK, %CALLER_TASK, %NEXT_TASK, or %VISIBLE_TASK).

Do not use the wildcard character (*). See the qualifier descriptions for details on how to specify tasks with particular qualifiers.

QUALIFIERS

/ALL

Selects all tasks that currently exist in the program for display. Do not specify a task with /ALL.

/CALLS[=n]

Performs a SHOW CALLS command for each task selected for display. You can use the SHOW CALLS command to obtain the current PC of a task.

/FULL

Displays additional information about each task selected for display. /FULL provides additional information if used either by itself, or with the /CALLS or /STATISTICS qualifier.

/[NO]HOLD

Selects either tasks that are on HOLD, or tasks that are not on HOLD for display.

If you do not specify a task, /HOLD selects all tasks that are on HOLD. If you specify a task list, /HOLD selects the tasks in the task list that are on HOLD.

If you do not specify a task, /NOHOLD selects all tasks that are not on HOLD. If you specify a task list, /NOHOLD selects the tasks in the task list that are not on HOLD.

/PRIORITY=(n[, . . .])

If you do not specify a task, selects all tasks that have any of the specified priorities, *n*, where *n* is a decimal integer from 0 to 15 inclusive. If you specify a task list, selects the tasks in the task list that have any of the priorities specified.

/STATE=(state[, . . .])

If you do not specify a task, selects all tasks that are in any of the specified states (the possible states are RUNNING, READY, SUSPENDED, or TERMINATED). If you specify a task list, selects the tasks in the task list that are in any of the states specified.

/STATISTICS

Displays tasking statistics for the entire tasking system. You can use this information to measure the performance of your tasking program. The larger the number of total schedulings (also known as context switches), the more tasking overhead there is. When you specify /STATISTICS, the only other permissible qualifier is /FULL.

/TIME_SLICE

Displays the current value of **pragma** TIME_SLICE.

DESCRIPTION

You can select tasks for display with the SHOW TASK command by specifying any of the following:

- A task list—that is, a list of task expressions.
- Task selection qualifiers: /ALL, /[NO]HOLD, /PRIORITY, /STATE.
- Both a task list and task selection qualifiers. Only the tasks that satisfy all specified criteria are selected for display.

If no task parameters or task selection qualifiers are given, the SHOW TASK command displays summary information about the visible task.

Related commands: SET TASK, SET BREAK/EVENT, SET TRACE/EVENT, EXAMINE/TASK, DEPOSIT/TASK.

EXAMPLES

1

```
DBG> SHOW TASK/ALL

  task id    pri hold state   substate        task object
* %TASK 1     7       RUN                     122624
  %TASK 2     7  HOLD SUSP  Accept            TASK_EXAMPLE.MONITOR
  %TASK 3     6       READY Entry call        TASK_EXAMPLE.CHECK_IN
```

The SHOW TASK/ALL command provides basic information on all the tasks of a program that are currently in existence—namely, tasks that have been created and whose master has not yet terminated. One line is devoted to each task. The active task is marked with an asterisk and is always the task that is in the RUN state.

2

```
DBG> SHOW TASK %ACTIVE_TASK,%TASK 3,MONITOR
```

This command selects the active task, %TASK 3, and task MONITOR for display.

3 `DBG> SHOW TASK/PRIORITY=6`

This command selects all tasks with priority 6 for display.

4 `DBG> SHOW TASK/STATE=(RUN,SUSP)`

This command selects all tasks that are either running or suspended for display.

5 `DBG> SHOW TASK/STATE=SUSP/NOHOLD`

This command selects all tasks that are both suspended and not on hold for display.

6 `DBG> SHOW TASK/STATE=(RUN,SUSP)/PRIO=7 %VISIBLE_TASK,%TASK 3`

This command selects for display those tasks among the visible task and %TASK 3 that are in either the RUNNING or SUSPENDED STATE, and have priority 7.

SHOW TERMINAL

Displays the current terminal screen height (page) and width being used to format output.

FORMAT

SHOW TERMINAL

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The current terminal screen height and width are the height and width last established by the SET TERMINAL command. If no SET TERMINAL command was issued, the current height and width are, by default, the height and width known to the VAX/VMS terminal driver, as displayed by the DCL command SHOW TERMINAL (usually 24 lines and 80 columns, respectively, for VT-series terminals).

Related commands: SET TERMINAL, SHOW DISPLAY, SHOW WINDOW.

EXAMPLE

```
DBG>  SHOW TERMINAL
terminal width: 80
         page:  24
```

This command displays the current terminal screen width and height (page) as 80 columns and 24 lines, respectively.

SHOW TRACE

Displays information about all tracepoints established by the SET TRACE command, including WHEN and DO clauses and /AFTER counts.

FORMAT

SHOW TRACE

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The debugger displays all information about each tracepoint that is currently set, including any optional WHEN and DO clauses.

If you established a tracepoint using the /AFTER:n command qualifier with the SET TRACE command, the SHOW TRACE command displays the current value of the decimal integer *n*, that is, the originally specified integer value minus one for each time the tracepoint location was reached. (The debugger decrements *n* each time the tracepoint location is reached until the value of *n* is zero, at which time the debugger takes trace action.)

Related commands: (SET, CANCEL) TRACE.

EXAMPLE

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

When the debugger encounters routine MULT in module CALC, or one of the instructions CALLS, CALLG, BSBW, BSBB, JSB, RSB, or RET, it will suspend program execution, report a message announcing its arrival at the tracepoint, and resume program execution.

SHOW TYPE

Displays the current type for program locations that do not have a compiler-generated type or, if the /OVERRIDE command qualifier is specified, the current override type.

FORMAT

SHOW TYPE

PARAMETERS

None.

QUALIFIERS

/OVERRIDE

Identifies the current override type.

DESCRIPTION

The current type for program locations that do not have a compiler-generated type is the type last established by the SET TYPE command. If no SET TYPE command was issued, the type for those locations is longword integer.

The current override type for all program locations is the override type last established by the SET TYPE/OVERRIDE command. If no SET TYPE /OVERRIDE command was issued, the override type is "none".

Related commands: SET TYPE, CANCEL TYPE/OVERRIDE.

EXAMPLES

1
```
DBG> SET TYPE QUADWORD
DBG> SHOW TYPE
type: quadword integer
```

The SET TYPE command sets the type for locations that do not have a compiler generated type to quadword. The SHOW TYPE command displays the current default type for those locations as quadword integer. This means that the debugger will interpret and display entities at those locations as quadword integers unless you specify otherwise (for example with a type qualifier on the EXAMINE command).

2
```
DBG> SHOW TYPE/OVERRIDE
type/override: none
```

This command indicates that no override type has been defined.

SHOW WATCH

Displays information about all watchpoints established by the SET WATCH command, including WHEN and DO clauses and /AFTER counts.

FORMAT

SHOW WATCH

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The debugger displays all information about each watchpoint that is currently set, including any optional WHEN and DO clauses.

If you established a watchpoint using the /AFTER:n command qualifier with the SET WATCH command, the SHOW WATCH command displays the current value of the decimal integer *n*, that is, the originally specified integer value minus one for each time the watchpoint location was reached. (The debugger decrements *n* each time the watchpoint location is reached until the value of *n* is zero, at which time the debugger takes watch action.)

Related commands: (SET, CANCEL) WATCH.

EXAMPLE

```
DBG> SHOW WATCH
watchpoint of MAIN\ALPHA
watchpoint of SUB2\TABLE+20
```

This command displays two watchpoints, one at location MAIN\ALPHA and the other at location SUB2\TABLE+20.

SHOW WINDOW

Displays the name and screen position of predefined and user-defined windows.

FORMAT

SHOW WINDOW *[wname[, . . .]]*

PARAMETERS

wname

Specifies the name of a window. If you do not specify a name, or if you specify the wildcard character (*) by itself, all window definitions are listed. You can use * within a window name. When using /ALL, do not specify a window name.

QUALIFIERS

/ALL

Lists all window definitions. Do not specify a window definition name with /ALL.

DESCRIPTION

Related commands: (SET, CANCEL) WINDOW, (SET, SHOW, CANCEL) DISPLAY, SHOW SELECT, (SET, SHOW) TERMINAL.

EXAMPLE

```
DBG> SHOW WINDOW LH*,RH*
window LH1 at (1,11,1,40)
window LH12 at (1,23,1,40)
window LH2 at (13,11,1,40)
window RH1 at (1,11,42,39)
window RH12 at (1,23,42,39)
window RH2 at (13,11,42,39)
```

This command displays the name and screen position of all screen window definitions whose names starts with LH or RH.

SPAWN

Lets you execute DCL commands without terminating a debugging session or losing the current debugging context.

FORMAT

SPAWN *[dcl-command]*

PARAMETERS

dcl-command

Specifies a DCL command. If you specify a DCL command, the command is executed in a subprocess. Control is returned to the debugging session when the DCL command terminates.

If you do not specify a DCL command, a subprocess is created and you can then enter DCL commands. Either logging out of the spawned process or attaching to the parent process (with the DCL ATTACH comand) will allow you to continue your debugging session.

If the DCL command contains a semicolon, you must enclose the command in quotation marks. Otherwise the semicolon is interpreted as a debugger command separator. To include a quotation mark inside the string, enter two consecutive quotation marks.

QUALIFIERS

/NOWAIT

Executes the subprocess in parallel with the debugging session. You can issue debugger commands while the subprocess is running. If you use /NOWAIT, you should specify a DCL command with the SPAWN command; the DCL command is executed in the subprocess.

DESCRIPTION

The SPAWN command acts exactly like the DCL SPAWN command. You can edit files, compile programs, read mail, and so on without ending your debugging session or losing your current debugging context.

In addition, you can spawn a DCL SPAWN command. DCL picks up the second SPAWN command and interprets the qualifier.

Related commands: ATTACH.

EXAMPLES

```
1   DBG> SPAWN
    $
```

The SPAWN command with no parameter specified spawns a subprocess at DCL level. You can now enter DCL commands. Log out to return to the debugger prompt.

```
2    DBG> SPAWN/NOWAIT SPAWN/OUT=MYCOM.LOG @MYCOM
     DBG>
```

The SPAWN/NOWAIT command spawns a subprocess that is executed in parallel with the debugging session. This subprocess spawns the execution of the command procedure MYCOM.COM. The output from that operation is written to the file MYCOM.LOG.

STEP

Causes the debugger to execute your program by line, by instruction, or by some other step unit. The step behavior depends on the step mode previously established by a SET STEP command and on the qualifier used with the STEP command.

FORMAT

STEP *[n]*

PARAMETERS

n

Specifies the number of lines or instructions to be executed by the STEP command. If you do not specify the parameter *n*, the debugger executes one line or one instruction. The parameter *n* is always interpreted as a decimal integer.

QUALIFIERS

/BRANCH

Causes the debugger to step to the next branch instruction. STEP/BRANCH does the same as SET BREAK/BRANCH;GO except that it does not create a permanent breakpoint.

/CALL

Causes the debugger to step to the next call or return instruction. STEP /CALL does the same as SET BREAK/CALL;GO except that it does not create a permanent breakpoint.

/EXCEPTION

Causes the debugger to step to the next exception condition. STEP /EXCEPTION does the same as SET BREAK/EXCEPTION;GO except that it does not create a permanent breakpoint.

/INSTRUCTION

Causes the debugger to step a single machine instruction. STEP /INSTRUCTION does the same as SET BREAK/TEMPORARY /INSTRUCTION;GO. This is the default behavior for language MACRO.

/INSTRUCTION=(opcode[, . . .])

Causes the debugger to step to the next machine instruction whose opcode is specified in the list. STEP/INSTRUCTION=(opcode[, . . .]) does the same as SET BREAK/TEMPORARY/INSTRUCTION=(opcode[, . . .]);GO.

/INTO

If you are at a call to a routine, causes the debugger to step into that routine. Otherwise, has the same effect as STEP without a qualifier.

The STEP/INTO behavior may be qualified as follows:

- If SET STEP NOJSB was previously specified, or if you specify STEP /INTO/NOJSB, you step over a routine that was called by a JSB instruction (see the description of the /OVER qualifier).
- If SET STEP NOSHARE was previously specified, or if you specify STEP /INTO/NOSHARE, you step over a routine that is in a shareable image.
- If SET STEP NOSYSTEM was previously specified, or if you specify STEP /INTO/NOSYSTEM you step over a routine that is in system (P1) space.

/[NO]JSB

/[NO]JSB qualifies a previous SET STEP INTO command or a current STEP /INTO command. If you are at a routine call, /JSB lets the debugger step into the routine, whether it is called by a CALL instruction or by a JSB instruction. This is the default for all languages except DIBOL. /NOJSB lets the debugger step into a routine called by a CALL instruction but causes the debugger to step over a routine called by a JSB instruction (see description of /OVER qualifier). In DIBOL, user-written routines are called by the CALL instruction and DIBOL run-time library routines are called by the JSB instruction.

/LINE

Causes the debugger to step to the next line of your program. This is the default behavior for all languages except MACRO.

/OVER

If you are at a call to a routine, causes the debugger to step over the routine. The routine is executed. However, any code executed in the routine, up to and including the corresponding RETURN instruction, is considered part of a single STEP. This is the default behavior.

/RETURN

Causes the debugger to step to the return instruction of the routine you are now in. Thus, STEP/RETURN *n* will take you up *n* levels of the call stack.)

/[NO]SHARE

/[NO]SHARE qualifies a previous SET STEP INTO command or a current STEP/INTO command. If you are at a call to a shareable image routine, /SHARE lets the debugger step into that routine. This is the default. /NOSHARE causes the debugger to step over that shareable image routine (see description of /OVER qualifier).

/[NO]SILENT

Controls whether the "stepped to . . . " message and other output associated with the STEP command is displayed. /SILENT specifies that no message or other output be displayed. /NOSILENT is the default and specifies that the step message and other output be displayed.

/[NO]SOURCE

Controls whether the source code corresponding to the current program location is displayed after the STEP command is executed. /SOURCE is the default and specifies that source code be displayed. /NOSOURCE specifies that no source code be displayed.

/[NO]SYSTEM

/[NO]SYSTEM qualifies a previous SET STEP INTO command or a current STEP/INTO command. If you are at a call to a system routine (in P1 space), /SYSTEM lets the debugger step into that routine. This is the default. /NOSYSTEM causes the debugger to step over that system routine (see description of /OVER qualifier).

DESCRIPTION

STEP command qualifiers determine the exact stepping behavior. In general, when you issue a STEP command, the debugger takes the following action:

1. Executes an instruction or a set of instructions.
2. Reports the instruction or line that follows the last instruction executed.
3. Reports the source line corresponding to the line or instruction that follows the last instruction executed (but only if the SOURCE parameter is in effect by virtue of STEP/SOURCE or SET STEP SOURCE and source lines are available).
4. Issues the DBG> prompt.

The following qualifiers affect the location to which you step:

/BRANCH
/CALL
/EXCEPTION
/INSTRUCTION
/INSTRUCTION=(opcode-list)
/LINE
/RETURN

The following qualifiers affect what output is seen on a step:

/[NO]SILENT
/[NO]SOURCE

The following qualifiers affect what happens at a routine call:

/INTO
/[NO]JSB
/OVER
/[NO]SHARE
/[NO]SYSTEM

Each language establishes default STEP conditions (use the SHOW STEP command to identify these). If you plan to issue several STEP commands with the same qualifiers, you can first use the SET STEP command to establish new default qualifiers (for example, SET STEP INTO NOSYSTEM makes the STEP command behave like STEP/INTO/NOSYSTEM). Then you do not have to use those qualifiers with the STEP command. You can override the current default qualifiers for the duration of a single STEP command by specifying other qualifiers.

Related commands: (SET, SHOW) STEP, GO, (SET, SHOW) LANGUAGE.

EXAMPLES

1
```
DBG> STEP
stepped to FORSQUARE$MAIN\%LINE 4
     4:         OPEN(UNIT=8, FILE='DATAFILE.DAT', STATUS='OLD')
```

This command tells the debugger to execute the next line (by default). The PC is then positioned at the beginning of line 4.

2
```
DBG> STEP/INSTRUCTION
stepped to MAIN\MAIN+14: MOVL 222,R0
```

This command tells the debugger to execute the next instruction. The PC is then positioned at instruction MOVL, located at MAIN\MAIN+14.

3
```
DBG> STEP/INTO
stepped to routine SUB1: MOVAL L^0000060C,R11
```

This command tells the debugger to step into the routine that is being called at the current PC location. The PC is then positioned at routine SUB1.

SYMBOLIZE

Converts a virtual address to a symbolic representation.

FORMAT

SYMBOLIZE *address-expression[, . . .]*

PARAMETERS

address-expression

Specifies an address expression to be symbolized. Do not use the wildcard character (*).

QUALIFIERS

None.

DESCRIPTION

If the address is a static address, it is symbolized as the nearest preceding symbol name, plus an offset. If the address is also a code address, the line number is included in the symbolization if a line number can be found that covers the address.

If the address is a register address, the debugger displays all symbols in all SET modules that are bound to that register. The full path name of each such symbol is displayed. The register name itself ("%R5," for example) is also displayed.

If the address is a stack location in the call frame of a routine in a SET module, the debugger searches for all symbols in that routine whose addresses are relative to the Frame Pointer (FP) or the Stack Pointer (SP). The closest preceding symbol name plus an offset is displayed as the symbolization of the address. A symbol whose address specification is too complex is ignored.

If the debugger can find no symbolization for the address, a message is displayed.

Related commands: SET MODE [NO]SYMBOLIC, SET MODE [NO]LINE.

EXAMPLE

```
DBG> SYMBOLIZE %R5
address PROG\%R5:
    PROG\X
```

This example shows that the local variable X in routine PROG is located in register R5.

TYPE

Displays lines of source code.

FORMAT

TYPE *[[mod-nam\]lin-num[:lin-num]*
[,[mod-nam\]lin-num[:lin-num][, . . .]]

PARAMETERS

mod-nam

Specifies the module that contains the source lines to be displayed. If you specify a module name along with the line numbers, use standard path-name notation: insert a backslash (\) between the module name and the line numbers.

If you do not specify a module name, the debugger uses the current scope search list (as established by a previous SET SCOPE command, or the default scope search list 0,1, . . . ,N if no SET SCOPE command was issued) to find source lines for display. The debugger looks for the source lines of the module associated with each successive scope region in the search list until source lines are found.

lin-num

Specifies a compiler-generated line number (a number used to label a source language statement or statements).

If you specify a single line number, the debugger displays the source code corresponding to that line number.

If you specify a list of line numbers, separating each with a comma, the debugger displays the source code corresponding to each of the line numbers.

If you specify a range of line numbers, separating the starting and ending line numbers in the range with a colon, the debugger displays the source code corresponding to that range of line numbers.

You can read through all the source language statements in your program by specifying a range of line numbers starting from 1 and ending at a number equal to or greater than the largest line number in the program listing.

QUALIFIERS

None.

DESCRIPTION

The TYPE command displays the lines of source code that correspond to the specified line numbers. The line numbers used by the debugger to identify lines of source code are generated by the compiler and appear in the compiler listing.

After displaying a single line of source code, you can display the next line by issuing a TYPE command without a line number, that is, by issuing a TYPE command and then pressing the RETURN key. You can then display the next line and successive lines by repeating this sequence, in effect, reading through your source program one line at a time.

TYPE

Related commands: SET MODE [NO]SCREEN, EXAMINE/SOURCE, SET STEP [NO]SOURCE, STEP/[NO]SOURCE, SET (BREAK, TRACE, WATCH) /[NO]SOURCE.

EXAMPLES

1
```
DBG> TYPE 160
module COBOLTEST
   160: START-IT-PARA.
DBG> TYPE
module COBOLTEST
   161:          MOVE SC1 TO ES0.
```

The first TYPE command displays line 160 of the source code and the second TYPE command displays the next line.

2
```
DBG> TYPE 160:163
module COBOLTEST
   160: START-IT-PARA.
   161:          MOVE SC1 TO ES0.
   162:          DISPLAY ES0.
   163:          MOVE SC1 TO ES1.
```

This command displays lines 160 through 163 of the source code.

3
```
DBG> TYPE COBOLTEST\160,22:24
module COBOLTEST
   160: START-IT-PARA.
module COBOLTEST
    22: 02      SC2V2   PIC S99V99      COMP VALUE  22.33.
    23: 02      SC2V2N  PIC S99V99      COMP VALUE -22.33.
    24: 02      CPP2    PIC PP99        COMP VALUE  0.0012.
```

This command displays lines 160 and lines 22 through 24 in the module COBOLTEST.

UNDEFINE

The UNDEFINE command is identical to the DELETE command. See the description of the DELETE command.

UNDEFINE/KEY

The UNDEFINE/KEY command is identical to the DELETE/KEY command. See the description of the DELETE/KEY command.

WHILE

Executes a sequence of commands conditionally.

FORMAT

WHILE *boolean-expression* **DO** *(command[; . . .])*

PARAMETERS

boolean-expression

Specifies a language expression that evaluates as a Boolean value (TRUE or FALSE) in the currently set language.

command

Specifies a debugger command. If you specify more than one command, separate them with semicolons.

QUALIFIERS

None.

DESCRIPTION

The WHILE command evaluates a boolean expression in the current language. If the value is TRUE, the command list in the DO clause is executed. The command then repeats the sequence, reevaluating the boolean-expression and executing the command-list until the expression is evaluated as FALSE.

If the boolean-expression is FALSE, the WHILE command terminates.

Related commands: FOR, REPEAT, EXITLOOP.

EXAMPLE

```
DBG> WHILE (X.EQ.0) DO (STEP/SILENT)
```

This command tells the debugger to keep stepping through the program until X no longer equals 0 (FORTRAN example).

Part III Appendixes

A Command Defaults

This appendix identifies the defaults associated with debugger commands.

Command	Default
@file-spec	For any field of the file specification that is not specified, the default is SYS$DISK:[ ]DEBUG.COM. This may be changed with a SET ATSIGN command.
CALL	Arguments are passed by address (%ADDR).
DEFINE	DEFINE/ADDRESS
DEFINE/KEY	DEFINE/KEY/ECHO/NOIF_STATE/NOLOCK_ STATE/LOG/NOSET_STATE/NOTERMINATE
DELETE/KEY	DELETE/KEY/LOG/NOSTATE
DEPOSIT	Language expressions and address expressions are interpreted according to the currently set language.
DISPLAY	DISPLAY/DYNAMIC/NOMARK_CHANGE/POP. The current display kind, window, and size remain unchanged.
EDIT	EDIT/NOEXIT. The default is to SPAWN the VAX Language Sensitive Editor. This may be changed with a SET EDITOR command. The default source file to be edited is the file whose source code appears in the current source display. The default position of the editing cursor is either the start of the line that is centered in the current source display, or the start of line 1 if the editor was set to /NOSTART_POSITION.
ENABLE (DISABLE) AST	ENABLE AST
EVALUATE	Language expressions are interpreted according to the currently set language.
EXAMINE	Address expressions are interpreted according to the currently set language.
EXPAND	EXPAND/DOWN, /UP: 1 line. EXPAND/LEFT, RIGHT: 1 column.
EXTRACT	If you specify /SCREEN_LAYOUT, the default specification for the output file is SYS$DISK:[ ]DBGSCREEN.COM. Otherwise, the default specification for the output file is SYS$DISK:[ ]DEBUG.TXT.
MOVE	MOVE/DOWN, /UP: 1 line. MOVE/LEFT, RIGHT: 1 column.
SCROLL	SCROLL/DOWN, /UP: 3/4 of window height. SCROLL/LEFT, /RIGHT: 8 columns.
SEARCH	SEARCH/NEXT/STRING

Command	Default
SELECT	SELECT/SCROLL
SET ATSIGN	SET ATSIGN SYS$DISK:[ ]DEBUG.COM
SET BREAK	SET BREAK/INTO/JSB/SHARE/SYSTEM /NOSILENT/SOURCE
SET DEFINE	SET DEFINE ADDRESS
SET DISPLAY	SET DISPLAY/DYNAMIC/POP/SIZE:64. The default window is either H1 or H2, alternating between these two with each newly created display. The default display kind is "output".
SET EDITOR	SET EDITOR/NOSTART_POSITION.
SET IMAGE	The current image is the main image.
SET KEY	SET KEY/STATE=DEFAULT
SET LANGUAGE	The default language is the language of the module that contains the image transfer address (main program).
SET LOG	SET LOG SYS$DISK:[ ]DEBUG.LOG
SET MARGINS	SET MARGINS 1:255 (left margin: 1, right margin: 255)
SET MAX_SOURCE_FILES	SET MAX_SOURCE_FILES 5
SET MODE	SET MODE DYNAMIC, NOG_FLOAT, KEYPAD, LINE, NOSCREEN, SCROLL, SYMBOLIC
SET OUTPUT	SET OUTPUT NOLOG, NOSCREEN_LOG, TERMINAL, NOVERIFY
SET PROMPT	SET PROMPT 'DBG>
SET RADIX	For BLISS and MACRO: SET RADIX HEXADECIMAL. For all other languages: SET RADIX DECIMAL.
SET SCOPE	The debugger looks up a symbol specified without a path-name prefix according to the scope search list 0,1, . . . ,N (where N is the number of calls in the call stack). If the symbol is not found, the debugger searches the run-time symbol table, then the global symbol table if necessary.
SET SEARCH	SET SEARCH NEXT,STRING
SET SOURCE	The debugger looks for a source debugger's symbol table.
SET STEP	For MACRO: SET STEP SOURCE, NOSILENT, OVER, INSTRUCTION. For all other languages: SET STEP SOURCE, NOSILENT, OVER, LINE.
SET TERMINAL	The values of /PAGE and /WIDTH default to those set at DCL level (see the *VAX/VMS DCL Dictionary* or type the DCL command HELP SET TERMINAL).
SET TRACE	SET TRACE/INTO/JSB/SHARE/SYSTEM /NOSILENT/SOURCE

Command	Default
SET TYPE	The default type for typed locations is the compiler-generated type. The default type for other locations is long integer.
STEP	For MACRO: STEP/OVER/INSTRUCTION. For all other languages: STEP/OVER/LINE.
TYPE	If a module name is specified, source lines in that module are displayed, if available. If no module name is specified, the debugger uses the current scope search list (as established by a previous SET SCOPE command, or the default scope search list 0,1, . . . ,N if no SET SCOPE command was issued) to find source lines for display. The debugger looks for the source lines of the module associated with each successive scope region in the search list until source lines are found. Also, if no line is specified after a single source line has been displayed with the TYPE command, the next line is displayed by default.

B Predefined Key Functions

When you invoke the debugger, certain predefined functions (commands, sequences of commands, and command terminators) are assigned to keys on the numeric keypad, to the right of the main keyboard. By using these keys you can issue certain commands with fewer keystrokes than if you were to type them out at the keyboard. For example, pressing the COMMA (,) keypad key is equivalent to typing out GO and then pressing the RETURN key. Terminals and workstations that have an LK201 keyboard have additional programmable keys compared to those on VT100 keyboards (for example, "Help" or "Remove"), and some of these keys are also assigned debugger functions.

Use of the function keys requires that keypad mode be enabled (SET MODE KEYPAD). Keypad mode is enabled when you invoke the debugger. If you do not want keypad mode enabled, perhaps because the program being debugged uses the keypad for itself, you can disable keypad mode by entering the SET MODE NOKEYPAD command.

The keypad key functions that are predefined when you invoke the debugger are identified in summary form in Figure B–1. Tables B–1 through B–4 identify all key definitions in detail. Most keys are used for manipulating screen displays in screen mode. To use screen mode commands, you must first enable screen mode: press keypad key PF3 (SET MODE SCREEN).

B.1 DEFAULT, GOLD, and BLUE Functions

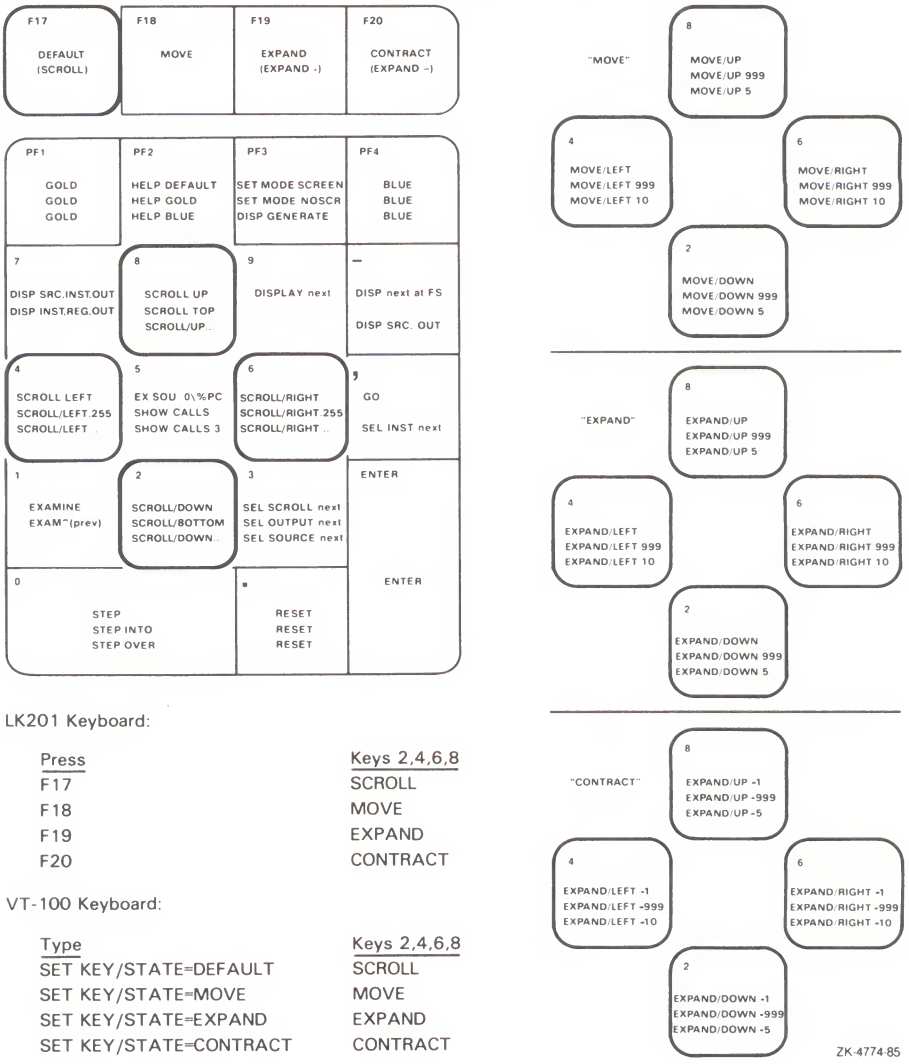
A given key typically has three predefined functions:

- One function is entered by pressing the given key by itself. This is the *DEFAULT* function.
- A second function is entered by pressing the PF1 key and then the given key. This is the *GOLD* function, because PF1 is also called the GOLD key.
- A third function is entered by pressing the PF4 key and then the given key. This is the *BLUE* function, because PF4 is also called the BLUE key.

In Figure B–1, the DEFAULT, GOLD, and BLUE functions are listed within each key's outline, from top to bottom respectively. For example, pressing keypad key 0 enters the command STEP (DEFAULT function); pressing key PF1 and then key 0 enters the command STEP/INTO (GOLD function); pressing key PF4 and then key 0 enters the command STEP/OVER (BLUE function).

All command sequences assigned to keypad keys are terminated (executed immediately) except for the BLUE functions of keys 2, 4, 6, and 8. These unterminated commands are symbolized with a trailing ellipsis (. . .) in Figure B–1. To terminate the command, supply a parameter and then press RETURN. For example, to scroll down 12 lines, press key PF4, then press keypad key 2, then type :12 at the keyboard, then press RETURN.

Figure B–1 Keypad Key Functions Predefined by the Debugger



LK201 Keyboard:

Press	Keys 2,4,6,8
F17	SCROLL
F18	MOVE
F19	EXPAND
F20	CONTRACT

VT-100 Keyboard:

Type	Keys 2,4,6,8
SET KEY/STATE=DEFAULT	SCROLL
SET KEY/STATE=MOVE	MOVE
SET KEY/STATE=EXPAND	EXPAND
SET KEY/STATE=CONTRACT	CONTRACT

B.2 Key Definitions Specific to LK201 Keyboards

Table B–1 lists keys that are specific to LK201 keyboards and do not appear on VT100 keyboards. For each key, the table identifies the equivalent command and, for some keys, an equivalent keypad key that you may use if you do not have an LK201 keyboard.

Table B–1 Key Definitions Specific to LK201 Keyboards

LK201 Key	Command Sequence Invoked	Equivalent Keypad Key
F17	SET KEY/STATE=DEFAULT	None
F18	SET KEY/STATE=MOVE	None
F19	SET KEY/STATE=EXPAND	None
F20	SET KEY/STATE=CONTRACT	None
Help	HELP KEYPAD SUMMARY	None
Next Screen	SCROLL/DOWN	2
Prev Screen	SCROLL/UP	8
Remove	DISPLAY/REMOVE %CURSCROLL	None
Select	SELECT/SCROLL %NEXTSCROLL	3

B.3 Keys that Scroll, Move, Expand, and Contract Displays

By default, keypad keys 2, 4, 6, and 8 scroll the current scrolling display. Each key controls a direction (down, left, right, and up, respectively). By pressing keys F18, F19, or F20, you can place the keypad in the MOVE, EXPAND, or CONTRACT states. When the keypad is in the MOVE state, keys 2, 4, 6, and 8 may be used to move the current scrolling display down, left, and so on. Similarly, in the EXPAND and CONTRACT states, the four keys may be used to expand or contract the current scrolling display. (See Figure B–1 and Table B–2. Alternative key definitions for VT100 keyboards are described later in this section.)

To scroll, move, expand, or contract a display, proceed as follows:

1. Press key 3 repeatedly, as needed, to select the current scrolling display from the display list.
2. Press key F17, F18, F19, or F20 to put the keypad in the DEFAULT (scroll), MOVE, EXPAND, or CONTRACT state, respectively.
3. Press keys 2, 4, 6, and 8 to perform the desired function. Use the PF1 (GOLD) and PF4 (BLUE) keys to control the amount of scrolling or movement.

Table B–2 Keys that Change the Key State

Key	Description
PF1	Invokes the GOLD function of the next key you press.
PF4	Invokes the BLUE function of the next key you press.
F17	Puts the keypad in the DEFAULT state, enabling the scroll-display functions of keys 2, 4, 6, and 8. The keypad is in the DEFAULT state when you invoke the debugger.
F18	Puts the keypad in the MOVE state, enabling the move-display functions of keys 2, 4, 6, and 8.

Table B–2 (Cont.) Keys that Change the Key State

Key	Description
F19	Puts the keypad in the EXPAND state, enabling the expand-display functions of keys 2, 4, 6, and 8.
F20	Puts the keypad in the CONTRACT state, enabling the contract-display functions of keys 2, 4, 6, and 8.

If you have a VT100 keyboard, you can simulate the effect of LK201 keys F17 through F20 by defining the key sequences GOLD-KP9 and BLUE-KP9 (currently undefined) as shown below. With these definitions, pressing GOLD-KP9 will put the keypad in the DEFAULT (scroll) state; pressing BLUE-KP9 will cycle the keypad through the DEFAULT, MOVE, EXPAND, and CONTRACT states (like cycling through keys F17 through F20). You may want to store these key definitions in a command procedure, such as your debugger initialization file.

```
DEFINE/KEY/IF_STATE=(GOLD,MOVE_GOLD,EXPAND_GOLD,CONTRACT_GOLD)/TERMINATE KP9 "Set Key/State=DEFAULT/Nolog"
DEFINE/KEY/IF_STATE=(BLUE)/TERMINATE KP9 "Set Key/State=MOVE/Nolog"
DEFINE/KEY/IF_STATE=(MOVE_BLUE)/TERMINATE KP9 "Set Key/State=EXPAND/Nolog"
DEFINE/KEY/IF_STATE=(EXPAND_BLUE)/TERMINATE KP9 "Set Key/State=CONTRACT/Nolog"
DEFINE/KEY/IF_STATE=(CONTRACT_BLUE)/TERMINATE KP9 "Set Key/State=DEFAULT/Nolog"
```

B.4 Online Keypad Key Diagrams

Online HELP for the keypad keys is available by pressing the Help key and also the PF2 key, either by itself or with other keys (see Table B–3). You can also use the SHOW KEY command to identify key definitions.

Table B–3 Keys that Invoke Online Help to Display Keypad Diagrams

Key or Key Sequence	Command Sequence Invoked	Description
Help	HELP KEYPAD SUMMARY	Shows a diagram of the keypad keys and summarizes each key's function.
PF2	HELP KEYPAD DEFAULT	Shows a diagram of the keypad keys and their DEFAULT functions.
PF1-PF2	HELP KEYPAD GOLD	Shows a diagram of the keypad keys and their GOLD functions.
PF4-PF2	HELP KEYPAD BLUE	Shows a diagram of the keypad keys and their BLUE functions.
F18-PF2	HELP KEYPAD MOVE_DEFAULT	Shows a diagram of the keypad keys and their MOVE DEFAULT functions.
F18-PF1-PF2	HELP KEYPAD MOVE_GOLD	Shows a diagram of the keypad keys and their MOVE GOLD functions.
F18-PF4-PF2	HELP KEYPAD MOVE_BLUE	Shows a diagram of the keypad keys and their MOVE BLUE functions.
F19-PF2	HELP KEYPAD EXPAND_DEFAULT	Shows a diagram of the keypad keys and their EXPAND DEFAULT functions.

Table B–3 (Cont.) Keys that Invoke Online Help to Display Keypad Diagrams

Key or Key Sequence	Command Sequence Invoked	Description
F19-PF1-PF2	HELP KEYPAD EXPAND_GOLD	Shows a diagram of the keypad keys and their EXPAND GOLD functions.
F19-PF4-PF2	HELP KEYPAD EXPAND_BLUE	Shows a diagram of the keypad keys and their EXPAND BLUE functions.
F20-PF2	HELP KEYPAD CONTRACT_DEFAULT	Shows a diagram of the keypad keys and their CONTRACT DEFAULT functions.
F20-PF1-PF2	HELP KEYPAD CONTRACT_GOLD	Shows a diagram of the keypad keys and their CONTRACT GOLD functions.
F20-PF4-PF2	HELP KEYPAD CONTRACT_BLUE	Shows a diagram of the keypad keys and their CONTRACT BLUE functions.

B.5 Debugger Key Definitions

Table B–4 identifies all key definitions.

Table B–4 Debugger Key Definitions

Key	State	Commands Invoked or Function
0	DEFAULT	STEP
	GOLD	STEP/INTO
	BLUE	STEP/OVER
1	DEFAULT	EXAMINE. Examines the logical successor of the current entity, if one is defined (the next location).
	GOLD	EXAMINE ^. Lets you examine the logical predecessor of the current entity, if one is defined (the previous location).
	BLUE	Undefined
2	DEFAULT	SCROLL/DOWN
	GOLD	SCROLL/BOTTOM
	BLUE	SCROLL/DOWN (not terminated). To terminate the command, supply the number of lines to be scrolled (:n) and/or a display name.
	MOVE	MOVE/DOWN
	MOVE_GOLD	MOVE/DOWN:999
	MOVE_BLUE	MOVE/DOWN:5
	EXPAND	EXPAND/DOWN
	EXPAND_GOLD	EXPAND/DOWN:999
	EXPAND_BLUE	EXPAND/DOWN:5
	CONTRACT	EXPAND/DOWN:-1
	CONTRACT_GOLD	EXPAND/DOWN:-999

Table B–4 (Cont.) Debugger Key Definitions

Key	State	Commands Invoked or Function
	CONTRACT_BLUE	EXPAND/DOWN:-5
3	DEFAULT	SELECT/SCROLL %NEXTSCROLL. Selects as the current scrolling display the next display in the display list after the current scrolling display.
	GOLD	SELECT/OUTPUT %NEXTOUTPUT. Selects the next output display in the display list as the current output display.
	BLUE	SELECT/SOURCE %NEXTSOURCE. Selects the next source display in the display list as the current source display.
4	DEFAULT	SCROLL/LEFT
	GOLD	SCROLL/LEFT:255
	BLUE	SCROLL/LEFT (not terminated). To terminate the command, supply the number of lines to be scrolled (:n) and/or a display name.
	MOVE	MOVE/LEFT
	MOVE_GOLD	MOVE/LEFT:999
	MOVE_BLUE	MOVE/LEFT:10
	EXPAND	EXPAND/LEFT
	EXPAND_GOLD	EXPAND/LEFT:999
	EXPAND_BLUE	EXPAND/LEFT:10
	CONTRACT	EXPAND/LEFT:-1
	CONTRACT_GOLD	EXPAND/LEFT:-999
	CONTRACT_BLUE	EXPAND/LEFT:-10
5	DEFAULT	EXAM/SOURCE .%SOURCE_SCOPE\%PC; EXAM/INST .0\%PC. In line (noscreen) mode, lets you see the source line or instruction to be executed next. In screen mode, centers the current source display on the next source line to be executed, and the current instruction display on the next instruction to be executed.
	GOLD	SHOW CALLS
	BLUE	SHOW CALLS 3
6	DEFAULT	SCROLL/RIGHT
	GOLD	SCROLL/RIGHT:255
	BLUE	SCROLL/RIGHT (not terminated). To terminate the command, supply the number of lines to be scrolled (:n) and/or a display name.
	MOVE	MOVE/RIGHT
	MOVE_GOLD	MOVE/RIGHT:999
	MOVE_BLUE	MOVE/RIGHT:10
	EXPAND	EXPAND/RIGHT

Table B–4 (Cont.) Debugger Key Definitions

Key	State	Commands Invoked or *Function*
	EXPAND_GOLD	EXPAND/RIGHT:999
	EXPAND_BLUE	EXPAND/RIGHT:10
	CONTRACT	EXPAND/RIGHT:-1
	CONTRACT_GOLD	EXPAND/RIGHT:-999
	CONTRACT_BLUE	EXPAND/RIGHT:-10
7	DEFAULT	DISPLAY SRC AT LH1, INST AT RH1, OUT AT S45, PROMPT AT S6; SELECT/SCROLL /SOURCE SRC; SELECT/INST INST; SELECT /OUT OUT. Displays the SRC, INST, OUT, and PROMPT displays with the proper attributes.
	GOLD	DISPLAY INST AT LH1, REG AT RH1, OUT AT S45, PROMPT AT S6; SELECT/SCROLL/INST INST; SELECT/OUT OUT. Displays the INST, REG, OUT, and PROMPT displays with the proper attributes. Useful for MACRO.
	BLUE	Undefined
8	DEFAULT	SCROLL/UP
	GOLD	SCROLL/TOP
	BLUE	SCROLL/UP (not terminated). To terminate the command, supply the number of lines to be scrolled (:n) and/or a display name.
	MOVE	MOVE/UP
	MOVE_GOLD	MOVE/UP:999
	MOVE_BLUE	MOVE/UP:5
	EXPAND	EXPAND/UP
	EXPAND_GOLD	EXPAND/UP:999
	EXPAND_BLUE	EXPAND/UP:5
	CONTRACT	EXPAND/UP:-1
	CONTRACT_GOLD	EXPAND/UP:-999
	CONTRACT_BLUE	EXPAND/UP:-5
9	DEFAULT	DISPLAY %NEXTDISP. Displays the next display in the display list through its current window (removed displays are not included).
	GOLD	Undefined
	BLUE	Undefined
PF1		See Table B–2.
PF2		See Table B–3.
PF3	DEFAULT	SET MODE SCREEN; SET STEP NOSOURCE. Enables screen mode and suppresses the output of source lines that would normally appear in the output display (since that output is redundant when the source display is present).

Table B–4 (Cont.) Debugger Key Definitions

Key	State	Commands Invoked or Function
	GOLD	SET MODE NOSCREEN; SET STEP SOURCE. Disables screen mode and restores the output of source lines.
	BLUE	DISPLAY/GENERATE. Regenerates the contents of all automatically updated displays.
PF4		See Table B–2.
COMMA	DEFAULT	GO
	GOLD	Undefined
	BLUE	SELECT/INSTRUCTION %NEXTINST. Selects the next instruction display in the display list as the current instruction display.
MINUS	DEFAULT	DISPLAY %NEXTDISP AT S12345, PROMPT AT S6; SELECT/SCROLL %CURDISP. Displays the next display in the display list at essentially full screen (top of screen to top of PROMPT display). Selects that display as the current scrolling display.
	GOLD	Undefined
	BLUE	DISPLAY SRC AT H1, OUT AT S45, PROMPT AT S6; SELECT/SCROLL/SOURCE SRC; SELECT /OUT OUT. Displays the SRC, OUT, and PROMPT displays with the proper attributes. This is the default display configuration for all languages except MACRO.
ENTER		Lets you enter (terminate) a command. Same effect as RETURN.
PERIOD		Cancels the effect of pressing state keys which do not lock the state, such as GOLD and BLUE. Does not affect the operation of state keys which lock the state, such as MOVE, EXPAND, and CONTRACT.
Next Screen		SCROLL/DOWN
Prev Screen		SCROLL/UP
Remove		DISPLAY/REMOVE %CURSCROLL. Removes the current scrolling display from the display list.
Select		SELECT/SCROLL %NEXTSCROLL. Selects as the current scrolling display the next display in the display list after the current scrolling display.
F17		See Table B–2.
F18		See Table B–2.
F19		See Table B–2.

Table B–4 (Cont.) Debugger Key Definitions

Key	State	Commands Invoked or Function
F20		See Table B–2.
CTRL/W		DISPLAY/REFRESH

C Screen-Mode Reference Information

This appendix contains summarized reference information related to screen mode. The following topics are covered:

- Display kinds
- Display attributes
- Predefined displays
- Screen-related built-in symbols
- Screen dimensions and predefined windows

C.1 Display Kinds

The SET DISPLAY and DISPLAY commands accept these *display-kind* keywords and parameters:

DO (cmd-list)	Specifies an automatically updated output display. The commands in cmd-list are executed each time the debugger gains control. Their output forms the contents of the display.
INSTRUCTION	Specifies an instruction display. If selected as the current instruction display with the SELECT /INSTRUCTION command, it will display the output from subsequent EXAMINE/INSTRUCTION commands.
INSTRUCTION (command)	Specifies an automatically updated instruction display. The command specified must be an EXAMINE/INSTRUCTION command. The instruction display is updated each time the debugger gains control.
OUTPUT	Specifies an output display. If selected as the current output display with the SELECT/OUTPUT command, it will display any debugger output that is not directed to another display. If selected as the current input display with the SELECT /INPUT command, it will echo debugger input. If selected as the current error display with the SELECT/ERROR command, it will display debugger diagnostic messages.
REGISTER	Specifies an automatically updated register display. The display is updated each time the debugger gains control.
SOURCE	Specifies a source display. If selected as the current source display with the SELECT /SOURCE command, it will display the output from subsequent TYPE or EXAMINE/SOURCE commands.

SOURCE (command)	Specifies an automatically updated source display. The command specified must be a TYPE or EXAMINE/SOURCE command. The source display is updated each time the debugger gains control.

C.2 Display Attributes

The SELECT command assigns an attribute to a display according to the qualifier used with that command. The following list identifies each of the SELECT command qualifiers, its effect, and the display kinds to which you can assign that attribute.

SELECT Qualifier	Description
/ERROR	Selects the specified display as the current error display. Directs any subsequent debugger diagnostic message to that display. It must be either an output display or the PROMPT display. If no display is specified, selects the PROMPT display as the current error display.
/INPUT	Selects the specified display as the current input display. Echoes any subsequent debugger input in that display. It must be an output display. If no display is specified, unselects the current input display: debugger input is not echoed to any display.
/INSTRUCTION	Selects the specified display as the current instruction display. Directs the output of any subsequent EXAMINE /INSTRUCTION command to that display. It must be an instruction display. Keypad key sequence BLUE-COMMA selects the next instruction display in the display list as the current instruction display. If no display is specified, unselects the current instruction display: no display has the instruction attribute.
/OUTPUT	Selects the specified display as the current output display. Directs any subsequent debugger output to that display, except where a particular type of output is being directed to another display (such as diagnostic messages going to the current error display). The specified display must be either an output display or the PROMPT display. Keypad key sequence GOLD-3 selects the next output display in the display list as the current output display. If no display is specified, selects the PROMPT display as the current output display.
/PROGRAM	Selects the specified display as the current program display. Tries to force any subsequent program input or output to that display. Currently, only the PROMPT display may be specified. If no display is specified, unselects the current program display: program output is no longer forced to the PROMPT display.

SELECT Qualifier	Description
/PROMPT	Selects the specified display as the current prompt display, where the debugger prompts for input. Currently, only the PROMPT display may be specified. You cannot unselect the PROMPT display.
/SCROLL	Selects the specified display as the current scrolling display. Makes that display the default display for any subsequent SCROLL, MOVE, or EXPAND command. You can specify any display (however, note that the PROMPT display cannot be scrolled). /SCROLL is the default if you do not specify a qualifier with the SELECT command. Key 3 selects as the current scrolling display the next display in the display list after the current scrolling display. If no display is specified, unselects the current scrolling display: no display has the scroll attribute.
/SOURCE	Selects the specified display as the current source display. Directs the output of any subsequent TYPE or EXAMINE /SOURCE command to that display. It must be a source display. Keypad key sequence BLUE-3 selects the next source display in the display list as the current source display. If no display is specified, unselects the current source display: no display has the source attribute.

By default, when you invoke screen mode, the predefined displays are selected for attributes as follows:

Attribute	Predefined Display
Error	PROMPT
Input	no display selected
Instruction	INST (for MACRO only)
Output	OUT
Program	PROMPT
Prompt	PROMPT
Scroll	SRC for all languages except MACRO, INST for MACRO
Source	SRC for all languages except MACRO

C.3 Predefined Displays

Properties of the predefined displays SRC, OUT, PROMPT, INST and REG are summarized in this section.

C.3.1 SRC (Source Display)

Note: The debugger does not provide source line display for MACRO. If the language is set to MACRO, SRC is marked as removed from the display pasteboard. The INST display is put in its place.

SRC is an automatically updated source display. It shows the source code of the module being debugged, if that source code is available. The arrow points to the source line corresponding to the current PC location.

The default characteristics of the SRC display are the following:

Display kind	source (examine/source .%source_scope\%pc)
Attributes	scroll, source for all languages except MACRO
Position	H1 (removed for MACRO)
Size	64 lines
Dynamic	Yes

%SOURCE_SCOPE is a built-in scope that signifies scope 0 when source lines are available for scope 0. Otherwise, %SOURCE_SCOPE signifies scope N, where N is the first level down the call stack where source lines are available. Thus, when source lines are available for the currently executing module, the SRC display is centered on the PC location. If source lines are not available for that module, the debugger attempts to display source lines in the caller of that module (scope 1). If source lines are also not available at that level, the debugger tries scope 2, and so on. If no source lines are available for any scope, the debugger issues a diagnostic message.

C.3.2 OUT (Output Display)

OUT shows all debugger output that is not directed to another display.

The default characteristics of the OUT display are the following:

Display kind	output
Attribute	output
Position	S45
Size	100 lines
Dynamic	Yes

C.3.3 PROMPT (Prompt Display)

PROMPT is where the debugger prompts for input and, by default, forces program output and prints debugger diagnostic messages.

PROMPT has different properties and restrictions than other displays. This is to eliminate possible confusion when manipulating that display:

- You cannot hide, remove, permanently delete, or scroll PROMPT.
- You can contract PROMPT down to 2 lines. You cannot contract PROMPT horizontally.

The default characteristics of the PROMPT display are the following:

Display kind	program
Attributes	error, prompt, program (no other display may have the prompt or program attributes)
Position	S6
Size	Not applicable (PROMPT is not scrollable)
Dynamic	Yes

C.3.4 INST (Instruction Display)

INST is an automatically updated instruction display. It shows the instruction stream of the routine being debugged. The instructions displayed are decoded from the image being debugged. The arrow points to the instruction at your current PC.

The default characteristics of the INST display are the following:

Display kind	instruction (examine/instruction .0\%pc)
Attributes	instruction, scroll (for MACRO only)
Position	H1, removed for all languages except MACRO
Size	64 lines
Dynamic	Yes

C.3.5 REG (Register Display)

REG automatically shows the current values of all VAX machine registers, the four condition code bits (C,V, Z, and N) of the processor status longword (PSL), and the top several values on the stack and on the current argument list. Values in this display are highlighted when they change as you execute the program.

The default characteristics of the REG display are the following:

Display kind	register
Attribute	none
Position	RH1, removed
Size	64 lines
Dynamic	No

C.4 Screen-Related Built-in Symbols

The following built-in symbols are available for specifying displays and screen parameters in language expressions:

- %SOURCE_SCOPE—Used to display source code. %SOURCE_SCOPE is described in Section 3.1.
- %PAGE, %WIDTH—Used to specify the current screen height and width.
- %CURDISP, %CURSCROLL, %NEXTDISP, %NEXTINST, %NEXTOUTPUT, %NEXTSCROLL, %NEXTSOURCE—Pseudo-display names, used to specify displays in the display list.

C.4.1 Terminal Height and Width

The built-in symbols %PAGE and %WIDTH return, respectively, the current height and width of the terminal screen. These symbols may be used in various expression, such as for window specifications. For example, the following command defines a window named MIDDLE that occupies a region around the middle of the screen:

```
DBG> SET WINDOW MIDDLE AT (%PAGE/4,%PAGE/2,%WIDTH/4,%WIDTH/2)
```

C.4.2 Pseudo-Display Names

Each time you refer to a specific display with a DISPLAY or SET DISPLAY command, the display list is updated and reordered, if necessary. The most recently referenced display is put at the tail of the display list since that display is pasted last on the pasteboard (the display list may be identified by issuing a SHOW DISPLAY command).

The debugger accepts seven *pseudo-display* names that refer to displays relative to their positions in the display list. These names, listed below, let you refer to displays by their relative positions in the list instead of by their explicit names. Pseudo-display names are used mainly in keypad or command definitions.

Pseudo-display names treat the display list as a circular list. Therefore, you can issue any commands that use pseudo-display names to cycle through the display list until you reach the display you want.

%CURDISP	Refers to the current display. This is the display most recently referenced with a DISPLAY or SET DISPLAY command—the least occluded display.
%CURSCROLL	Refers to the current scrolling display. This is the default display for the SCROLL, MOVE, and EXPAND commands, as well as for the associated keypad keys (2, 4, 6, and 8).
%NEXTDISP	Refers to the next display in the list after the current display. The next display is the display that follows the topmost display. Because the display list is circular, this is the display at the bottom of the pasteboard—the most occluded display.
%NEXTINST	Refers to the next instruction display in the display list after the current instruction display. The current instruction display is the display which receives the output from EXAMINE /INSTRUCTION commands.
%NEXTOUTPUT	Refers to the next output display in the display list after the current output display. An output display receives debugger output that is not already directed to another display.
%NEXTSCROLL	Refers to the next display in the display list after the current scrolling display.
%NEXTSOURCE	Refers to the next source display in the display list after the current source display. The current source display is the display which receives the output from TYPE and EXAMINE /SOURCE commands.

C.5 Screen Dimensions and Predefined Windows

On a VT-series terminal, the screen consists of 24 lines by 80 or 132 columns. On a MicroVAX workstation, the screen is larger in both height and width. The debugger can accommodate screen sizes up to 100 lines by 255 columns.

All of the debugger predefined windows are identified in this section. The window names apply to all screen sizes; however, the lines and columns specified apply only to VT-series terminals whose width is set at 80 columns. In windows occupying the left or right half of the screen (columns 1-40 and 42-80, respectively), column 41 is reserved as a border.

In addition to the full height and width of the screen, the predefined windows include all possible regions that result from dividing the screen vertically into halves, thirds, quarters, and sixths, and horizontally into left and right halves.

The following conventions apply to the names of predefined windows: the prefixes L and R denote left and right windows, respectively; other letters denote the full screen (FS) or fractions of the screen height (H: half, T: third, Q: quarter, S: sixth); the trailing numbers denote specific fractions of the screen height, starting from the top (for example, T1, T2, and T3 are the top, middle and bottom third).

Window Name	Start-lin,Lin-count, Start-col,Col-count	Window Location
FS	(1,23,1,80)	Full screen
H1	(1,11,1,80)	Top half
H12	(1,23,1,80)	Full screen
H2	(13,11,1,80)	Bottom half
LFS	(1,23,1,40)	Left full screen
LH1	(1,11,1,40)	Left half
LH12	(1,23,1,40)	Left full screen
LH2	(13,11,1,40)	Left bottom half
LQ1	(1,5,1,40)	Left top quarter
LQ12	(1,11,1,40)	Left top half
LQ123	(1,17,1,40)	Left top three quarters
LQ1234	(1,23,1,40)	Left full screen
LQ2	(7,5,1,40)	Left second quarter
LQ23	(7,11,1,40)	Left middle two quarters
LQ234	(7,17,1,40)	Left bottom three quarters
LQ3	(13,5,1,40)	Left third quarter
LQ34	(13,11,1,40)	Left bottom half
LQ4	(19,5,1,40)	Left bottom quarter
LS1	(1,3,1,40)	Left top sixth
LS12	(1,7,1,40)	Left top third
LS123	(1,11,1,40)	Left top half
LS1234	(1,15,1,40)	Left top two thirds
LS12345	(1,19,1,40)	Left top five sixths

Window Name	Start-lin,Lin-count, Start-col,Col-count	Window Location
LS123456	(1,23,1,40)	Left half
LS2	(5,3,1,40)	Left second sixth
LS23	(5,7,1,40)	Left second and third sixths
LS234	(5,11,1,40)	Left second, third, and fourth sixths
LS2345	(5,15,1,40)	Left second, third, fourth, and fifth sixths
LS23456	(5,19,1,40)	Left bottom five sixths
LS3	(9,3,1,40)	Left third sixth
LS34	(9,7,1,40)	Left middle third
LS345	(9,11,1,40)	Left third, fourth, and fifth sixths
LS3456	(9,15,1,40)	Left bottom two thirds
LS4	(13,3,1,40)	Left fourth sixth
LS45	(13,7,1,40)	Left fourth and fifth sixths
LS456	(13,11,1,40)	Left bottom half
LS5	(17,3,1,40)	Left fifth sixth
LS56	(17,7,1,40)	Left bottom third
LS6	(21,3,1,40)	Left bottom sixth
LT1	(1,7,1,40)	Left top third
LT12	(1,15,1,40)	Left top two thirds
LT123	(1,23,1,40)	Left half
LT2	(9,7,1,40)	Left middle third
LT23	(9,15,1,40)	Left bottom two thirds
LT3	(17,7,1,40)	Left bottom third
Q1	(1,5,1,80)	Top quarter
Q12	(1,11,1,80)	Top half
Q123	(1,17,1,80)	Top three quarters
Q1234	(1,23,1,80)	Full screen
Q2	(7,5,1,80)	Second quarter
Q23	(7,11,1,80)	Middle two quarters
Q234	(7,17,1,80)	Bottom three quarters
Q3	(13,5,1,80)	Third quarter
Q34	(13,11,1,80)	Bottom half
Q4	(19,5,1,80)	Bottom quarter
RFS	(1,23,42,39)	Right full screen
RH1	(1,11,42,39)	Right top half
RH12	(1,23,42,39)	Right full screen
RH2	(13,11,42,39)	Right bottom half
RQ1	(1,5,42,39)	Right top quarter
RQ12	(1,11,42,39)	Right top half
RQ123	(1,17,42,39)	Right top three quarters

Window Name	Start-lin,Lin-count, Start-col,Col-count	Window Location
RQ1234	(1,23,42,39)	Right half
RQ2	(7,5,42,39)	Right second quarter
RQ23	(7,11,42,39)	Right middle two quarters
RQ234	(7,17,42,39)	Right bottom three quarters
RQ3	(13,5,42,39)	Right third quarter
RQ34	(13,11,42,39)	Right bottom half
RQ4	(19,5,42,39)	Right bottom quarter
RS1	(1,3,42,39)	Right top sixth
RS12	(1,7,42,39)	Right top third
RS123	(1,11,42,39)	Right top half
RS1234	(1,15,42,39)	Right top two thirds
RS12345	(1,19,42,39)	Right top five sixths
RS123456	(1,23,42,39)	Right full screen
RS2	(5,3,42,39)	Right second sixth
RS23	(5,7,42,39)	Right second and third sixths
RS234	(5,11,42,39)	Right second, third, and fourth sixths
RS2345	(5,15,42,39)	Right second, third, fourth, and fifth sixths
RS23456	(5,19,42,39)	Right bottom five sixths
RS3	(9,3,42,39)	Right third sixth
RS34	(9,7,42,39)	Right middle third
RS345	(9,11,42,39)	Right third, fourth, and fifth sixths
RS3456	(9,15,42,39)	Right bottom two thirds
RS4	(13,3,42,39)	Right fourth sixth
RS45	(13,7,42,39)	Right fourth and fifth sixth
RS456	(13,11,42,39)	Right bottom half
RS5	(17,3,42,39)	Right fifth sixth
RS56	(17,7,42,39)	Right bottom third
RS6	(21,3,42,39)	Right bottom sixth
RT1	(1,7,42,39)	Right top third
RT12	(1,15,42,39)	Right top two thirds
RT123	(1,23,42,39)	Right full screen
RT2	(9,7,42,39)	Right middle third
RT23	(9,15,42,39)	Right bottom two thirds
RT3	(17,7,42,39)	Right bottom third
S1	(1,3,1,80)	Top sixth
S12	(1,7,1,80)	Top third
S123	(1,11,1,80)	Top half
S1234	(1,15,1,80)	Top four sixths
S12345	(1,19,1,80)	Top five sixths

Window Name	Start-lin,Lin-count, Start-col,Col-count	Window Location
S123456	(1,23,1,80)	Full screen
S2	(5,3,1,80)	Top second sixth
S23	(5,7,1,80)	Second and third sixths
S234	(5,11,1,80)	Second, third, and fourth sixths
S2345	(5,15,1,80)	Second, third, fourth, and fifth sixths
S23456	(5,19,1,80)	Bottom five sixths
S3	(9,3,1,80)	Third sixth
S34	(9,7,1,80)	Middle third
S345	(9,11,1,80)	Third, fourth, and fifth sixths
S3456	(9,15,1,80)	Bottom two thirds
S4	(13,3,1,80)	Fourth sixth
S45	(13,7,1,80)	Fourth and fifth sixths
S456	(13,11,1,80)	Bottom half
S5	(17,3,1,80)	Fifth sixth
S56	(17,7,1,80)	Bottom third
S6	(21,3,1,80)	Bottom sixth
T1	(1,7,1,80)	Top third
T12	(1,15,1,80)	Top two thirds
T123	(1,23,1,80)	Full screen
T2	(9,7,1,80)	Middle third
T23	(9,15,1,80)	Bottom two thirds
T3	(17,7,1,80)	Bottom third

D Built-in Symbols and Logical Names

This appendix identifies all of the debugger built-in symbols and logical names.

D.1 SS$_DEBUG Condition

SS$_DEBUG (defined in SYS$LIBRARY:STARLET.OLB) is a condition you can signal from your program to invoke the debugger. Signalling SS$_DEBUG from your program is equivalent to typing CTRL/Y followed by the DCL command DEBUG at that point.

You can pass commands to the debugger at the time you signal it with SS$_DEBUG. The commands you wish the debugger to execute should be specified as you would enter them at the DBG> prompt. Multiple commands should be separated by semicolons. The commands should be passed by reference as an ASCIC string. See your language documentation for details on constructing an ASCIC string.

For example, to invoke the debugger and issue a SHOW CALLS command at a given point in your program, you could insert the following code in your program (BLISS example):

```
LIB$SIGNAL(SS$_DEBUG, 1, UPLIT BYTE(%ASCIC 'SHOW CALLS'));
```

You can obtain the definition of SS$_DEBUG at compile time from the appropriate STARLET or SYSDEF file for your language (for example STARLET.L32 for BLISS or FORSYSDEF.TLB for FORTRAN). You can also obtain the definition of SS$_DEBUG at link time in SYS$LIBRARY:STARLET.OLB (this method is less desirable).

D.2 Logical Names

The following list identifies debugger-specific process logical names.

Logical Name	Description
LIB$DEBUG	Points to the current debugger. Default: SYS$SHARE:DEBUG.EXE
DBG$HELP	Points to the debugger online-Help library file. Default: SYS$HELP:DEBUGHLP.HLB.
DBG$INIT	Points to your debugger initialization file. Default: no debugger initialization file.
DBG$INPUT	Points to your input device. Default: SYS$INPUT.
DBG$OUTPUT	Points to your output device. Default: SYS$OUTPUT.

You can use the DCL command ASSIGN or DEFINE to assign values to these logical names. For example, the following command tells the debugger the location of your debugger initialization file:

```
$ DEFINE DBG$INIT DISK$:[JONES.COMFILES]DEBUGINIT.COM
```

Note that LIB$DEBUG, DBG$HELP, and DBG$INIT accept a full or partial VAX/VMS file specification, as well as a search list.

D.2.1 Using DBG$INPUT and DBG$OUTPUT

DBG$INPUT and DBG$OUTPUT are useful when, for example, you are debugging a screen-oriented program. You can assign DBG$INPUT and DBG$OUTPUT to one terminal line (for example, TTD1:). Another terminal line (for example, TTD2:) can be devoted to the program's input and output. Then, you can

- Enter debugger commands and observe debugger output at TTD1:.
- Enter program input and observe program output at TTD2:.

Note that, on a properly secured system, terminals are protected so that you can log in but you cannot allocate a terminal. Use the following command to determine the owner UIC of TTD1:

```
$ SHOW DEVICE/FULL TTD1:
```

Typically, you will need to ask your system manager (or a suitably privileged person) to provide you with read access to the terminal. For example:

```
$ SET PROTECTION=WORLD:READ/DEVICE/OWNER=[SYSTEM] TTD1:
```

The above technique provides world read access and, therefore, allows other users to also allocate and perform I/O to TTD1:. The following technique is preferred because it uses an access control list (ACL). Assume that your UIC is [DEVEL,JONES]. Then the system manager can restrict device access to you alone as follows:

```
$ SET DEVICE/ACL=(IDENT=[DEVEL,JONES],ACCESS=READ) TTD1:
```

Another method is for you to enable SYSPRIV privilege in your own process so that you can allocate TTD1: without allowing the world to allocate it. However, note that it is risky to debug an erroneous program with SYSPRIV enabled.

Once you have read access to the terminal, you can allocate it so that you have exclusive access to it:

```
$ ALLOCATE TTD1:
```

Now you can assign DBG$INPUT and DBG$OUTPUT:

```
$ DEFINE DBG$INPUT TTD1:
$ DEFINE DBG$OUTPUT TTD1:
```

The remaining step is to make sure that the terminal type is known to the system. Use the following command:

```
$ SHOW DEVICE/FULL TTD1:
```

If the device type is "unknown", make it known to the system as follows (assuming the terminal is a VT100):

```
$ SET TERMINAL/PERMANENT/DEVICE=VT100 TTD1:
```

Now you can run your program and observe debugger input and output at TTD1. When finished, deallocate TTD1: as follows:

```
$ DEALLOCATE TTD1:
```

D.3 Built-in Symbols

The debugger's built-in symbols provide options for specifying entities in your program and let you control the debugger's scanning of language expressions. Most of the debugger built-in symbols have a percent sign (%) prefix. Descriptions of these symbols are organized as follows in the next sections:

- %R0 through %R11, %PC, %PSL, %SP, %AP, %FP—Used to specify the VAX registers.
- %NAME—Used to construct identifiers.
- %PARCNT—Used in command procedures to count parameters passed.
- %BIN, %DEC, %HEX, %OCT—Used to control radix.
- Period (.), RETURN key, circumflex (^), backslash (\), %CURLOC, %NEXTLOC, %PREVLOC, %CURVAL, %LABEL, %LINE—Used to specify program locations and the current value of an entity.
- Plus sign (+), minus sign (-), multiplication sign (*), division sign (/), at sign (@), period (.), bit field operator (<p,s,e>)—Used as operators in address expressions.
- %ADAEXC_NAME, %EXC_FACILITY, %EXC_NAME, %EXC_NUMBER, %EXC_SEVERITY—Used to obtain information about exceptions.
- %ACTIVE_TASK, %CALLER_TASK, %NEXT_TASK, %TASK, %VISIBLE_TASK—Used to specify tasks in Ada tasking programs.
- %CURDISP, %CURSCROLL, %NEXTDISP, %NEXTINST, %NEXTOUTPUT, %NEXTSCROLL, %NEXTSOURCE—Used in screen mode to specify displays in the display list (these built-in symbols are described in Appendix C).
- %PAGE, %WIDTH, %SOURCE_SCOPE—Used to specify the current terminal-screen height and width, and to display source code in screen mode (these built-in symbols are described in Appendix C).

D.3.1 Specifying the VAX Registers

The debugger built-in symbol for a VAX register is the register name preceded by the percent sign (%). These symbols are identified in the following list.

Symbol	Description
%R0 . . . %R11	General purpose registers R0 . . . R11
%PC	Program counter
%PSL	Processor status longword
%SP	Stack pointer
%AP	Argument pointer
%FP	Frame pointer

For example, the following EXAMINE command obtains the contents of the PC (the address contained in the PC):

```
DBG> EXAMINE %PC
MOD\%PC: 1553
```

D.3.2 Constructing Identifiers

The %NAME built-in symbol lets you construct identifiers that are not ordinarily legal in the current language. The syntax is as follows:

```
%NAME 'character-string'
```

In the following example, the variable with the name '12' is examined:

```
DBG> EXAMINE %NAME '12'
```

In the following example, the compiler-generated label P.AAA is examined:

```
DBG> EXAMINE %NAME 'P.AAA'
```

D.3.3 Counting Parameters Passed to Command Procedures

The %PARCNT built-in symbol may be used within a command procedure that accepts a variable number of actual parameters (%PARCNT is defined only within a debugger command procedure).

%PARCNT specifies the number of actual parameters passed to the current command procedure. In the following example, command procedure ABC.COM is invoked and three parameters are passed:

```
DBG> @ABC 111,222,333
```

Within ABC.COM, %PARCNT now has the value 3. %PARCNT is then used as a loop counter to obtain the value of each parameter passed to ABC.COM:

```
DBG> FOR I = 1 TO %PARCNT DO (DECLARE X:VALUE; EVALUATE X)
```

D.3.4 Controlling Radix

The built-in symbols %BIN, %DEC, %HEX, and %OCT are used to specify that a numeric literal that follows (or all numeric literals in a parenthesized expression that follows) should be interpreted in binary, decimal, hexadecimal, or octal radix, respectively.

For example:

```
DBG> EVALUATE/DEC %HEX 10
16
DBG> EVALUATE/DEC %HEX (10 + 10)
32
DBG> EVALUATE/DEC %BIN 10
2
DBG> EVALUATE/DEC %OCT (10 + 10)
16
DBG> EVALUATE/HEX %DEC 10
0A
```

D.3.5 Specifying Program Locations and the Current Value of an Entity

The following built-in symbols let you specify program locations and the current value of an entity.

Symbol	Description
%CURLOC . (period)	Current logical entity—the program location last referenced by an EXAMINE or DEPOSIT command.
%NEXTLOC RETURN key	Logical successor of the current entity—the program location that logically follows the location last referenced by an EXAMINE or DEPOSIT command. Because the RETURN key is a command terminator, it can be used only where a command terminator is appropriate (for example, immediately after EXAMINE, but not immediately after DEPOSIT).
%PREVLOC ^ (circumflex)	Logical predecessor of current entity—the program location that logically precedes the location last referenced by an EXAMINE or DEPOSIT command.
%CURVAL \ (backslash)	Value last displayed by an EVALUATE or EXAMINE command, or deposited by a DEPOSIT command.
%LABEL	Specifies that the numeric literal that follows is a program label (for languages like FORTRAN that have numeric program labels). You can qualify the label with a path-name prefix that specifies the containing module.
%LINE	Specifies that the numeric literal that follows is a line number in your program. You can qualify the line number with a path-name prefix that specifies the containing module.

In the following example, the variable WIDTH is examined; the value 12 is then deposited into the current location (WIDTH); this is verified by examining the current location:

```
DBG> EXAMINE WIDTH
MOD\WIDTH: 7
DBG> DEPOSIT . = 12
DBG> EXAMINE .
MOD\WIDTH: 12
DBG> EXAMINE %CURLOC
MOD\WIDTH: 12
```

In the next example, the next and previous locations in an array are examined:

```
DBG> EXAMINE PRIMES(4)
MOD\PRIMES(4): 7
DBG> EXAMINE %NEXTLOC
MOD\PRIMES(5): 11
DBG> EXAMINE RET          ! Examine next location
MOD\PRIMES(6): 13
DBG> EXAMINE %PREVLOC
MOD\PRIMES(5): 11
DBG> EXAMINE ^
MOD\PRIMES(4): 7
```

Note that using the RETURN key to signify the logical successor does not apply to all contexts. For example, you cannot press the RETURN key after typing the command DEPOSIT to indicate the next location, whereas you can always use the symbol %NEXTLOC for that purpose.

In the next example, a breakpoint is set at label 10 of module MOD4:

```
DBG> SET BREAK MOD4\%LABEL 10
```

D.3.6 Using Operators in Address Expressions

The operators that may be used in address expressions are listed below. A unary operator has one operand. A binary operator has two operands.

Symbol	Description
Plus sign (+)	Unary or binary operator. As a unary operator, indicates the unchanged value of its operand. As a binary operator, adds the preceding operand and succeeding operand together.
Minus sign (−)	Unary or binary operator. As a unary operator, indicates the negation of the value of its operand. As a binary operator, subtracts the succeeding operand from the preceding operand.
Multiplication sign (*)	Binary operator. Multiplies the preceding operand by the succeeding operand.
Division sign (/)	Binary operator. Divides the preceding operand by the succeeding operand.
At sign (@) Period (.)	Unary operators. In an address expression, the at sign (@) and period (.) each function as a "contents-of" operator. The "contents-of" operator causes its operand to be interpreted as a virtual address and thus requests the contents of (or value residing at) that address.
Bit field <p,s,e>	Unary operator. You can apply bit field selection to an address-expression. To select a bit field, you supply a bit offset (p), a bit length (s), and a sign extension bit (e), which is optional.

In the following example, the value contained in the virtual memory location X + 4 bytes is obtained:

```
EXAMINE X + 4
```

The remaining examples illustrate use of the "contents-of" operator. In the next example, the instruction at the current PC is obtained (the instruction whose address is contained in the PC and which is about to execute):

```
DBG> EXAMINE .%PC
MOD\%LINE 5: PUSHL   S^#8
```

In the next example, the source line at the PC location one level down the call stack is obtained (at the call to routine SWAP):

```
DBG> EXAMINE/SOURCE .1\%PC
MAIN\SWAP\%LINE 28:     SWAP(X,Y);
```

For the next example, assume that the value of pointer variable PTR is 7FF00000 hexadecimal, the virtual address of an entity that you want to examine. Assume further that the value of this entity is 3FF00000 hexadecimal. The following command shows how to examine entity:

```
DBG> EXAMINE/LONG .PTR
7FF00000: 3FF00000
```

In the next example, the contents-of operator (at sign or period) is used with the current location operator (period) to examine a linked list of three quadword-integer pointer variables (identified as L1, L2, and L3 in the illustration that follows). P is a pointer to the start of the list. The low longword of each pointer variable contains the address of the next variable; the high longword of each variable contains its integer value (8, 6, and 12 respectively).


```
DBG> SET TYPE QUADWORD; SET RADIX HEX
DBG> EXAMINE .P                 ! Examine the entity whose address
                                ! is contained in P
00009BC2: 00000008 00009BDA     ! High word has value 8, low word
                                ! has address of next entity (9BDA)
DBG> EXAMINE @.                 ! Examine the entity whose address
                                ! is contained in the current entity
00009BDA: 00000006 00009BF4     ! High word has value 6, low word
                                ! has address of next entity (9BF4)
DBG> EXAMINE ..                 ! Examine the entity whose address
                                ! is contained in the current entity
00009BF4: 0000000C 00000000     ! High word has value 12 (dec.), low word
                                ! has address 0 (end of list)
```

The next example shows how to use the bit-field operator. To examine the address expression X_NAME starting at bit 3 with a length of 4 bits and no sign extension, you would issue the following command:

```
DBG> EXAMINE X_NAME <3,4,0>
```

D.3.7 Obtaining Information About Exceptions

The following built-in symbols let you obtain information about the current exception and use that information to qualify breakpoints.

Symbol	Description
%ADAEXC_ NAME	Ada exception name of current exception (for Ada programs only)
%EXC_FACILITY	Name of facility of current exception
%EXC_NAME	Name of current exception
%EXC_NUMBER	Number of current exception
%EXC_SEVERITY	Severity code of current exception

For example:

```
DBG> EVALUATE %EXC_NAME
"FLTDIV_F"
DBG> SET BREAK/EXCEPTION WHEN (%EXC_NAME = "FLTDIV_F")
   .
   .
   .
DBG> EVALUATE %EXC_NUMBER
12
DBG> EVALUATE/CONDITION_VALUE %EXC_NUMBER
%SYSTEM-F-ACCVIO, access violation at PC !XL, virtual address !XL
DBG> SET BREAK/EXCEPTION WHEN (%EXC_NUMBER = 12)
```

D.3.8 Specifying Ada Tasks

The following built-in symbols may be used to specify the tasks of an Ada tasking program in debugger commands (these built-in symbols apply only to Ada tasking programs).

Symbol	Description
%ACTIVE_TASK	Currently active task—the task that will execute when a GO or STEP command is issued.
%CALLER_TASK	Task that is the entry caller of the active task during a task rendezvous.
%NEXT_TASK	Next task on debugger's task list after the task that is currently visible.
%TASK n	Specifies a task by means of its task ID (n is a decimal integer assigned by the VAX Ada run-time library to each task as it is created).
%VISIBLE_TASK	Currently visible task—the task that is the context for an EXAMINE command, for example.

Two examples follow. See the VAX Ada documentation for additional details.

```
DBG> EXAMINE MONITOR_TASK
MOD\MONITOR_TASK: %TASK 2
    .
    .
    .
DBG> WHILE %NEXT NEQ %ACTIVE DO (SET TASK %NEXT; SHOW CALLS)
```

E Summary of Debugger Support for Languages

The debugger supports most of the VAX/VMS-supported languages. Debugger support is summarized in this chapter for the following language keywords (used with the SET LANGUAGE command): ADA, BASIC, BLISS, C, COBOL, DIBOL, FORTRAN, MACRO, PASCAL, PLI, RPG, SCAN, and UNKNOWN. For each language, the following information is provided:

- Supported operators in language expressions
- Supported constructs in language expressions
- Supported data types
- Any other language-specific features (for example, event keywords in the case of ADA and SCAN).

For further information, refer to the documentation furnished with a particular language.

E.1 Debugger Support for Language ADA

This section includes information about debugger support for ADA.

E.1.1 Supported ADA Operators in Language Expressions

Supported ADA operators in language expressions follow.

Kind	Symbol	Function
Prefix	+	Unary plus (identity)
Prefix	–	Unary minus (negation)
Infix	+	Addition
Infix	–	Subtraction
Infix	*	Multiplication
Infix	/	Division
Infix	MOD	Modulus
Infix	REM	Remainder
Infix	**	Exponentiation
Prefix	ABS	Absolute value
Infix	&	Concatenation (only string types)
Infix	=	Equality (only scalar and string types)
Infix	/=	Inequality (only scalar and string types)
Infix	>	Greater than (only scalar and string types)
Infix	> =	Greater than or equal (only scalar and string types)

Kind	Symbol	Function
Infix	<	Less than (only scalar and string types)
Infix	<=	Less than or equal (only scalar and string types)
Prefix	NOT	Logical NOT
Infix	AND	Logical AND (not for bit arrays)
Infix	OR	Logical OR (not for bit arrays)
Infix	XOR	Logical exclusive OR (not for bit arrays)

Notes

1 The debugger does not support:

- Operations on entire arrays or records.
- The short-circuit control forms: **and then**, **or else**.
- The membership tests: **in**, **not in**.

2 The debugger does not support user-defined operators.

E.1.2 Supported Constructs in Language and Address Expressions for ADA

Supported constructs in language and address expressions for ADA follow.

Symbol	Construct
( )	Subscripting
.	Record component selection
.ALL	Pointer dereferencing

E.1.3 Supported ADA Data Types

Supported ADA data types follow.

ADA Type	VAX Type Name
INTEGER	Longword Integer (L)
SHORT_INTEGER	Word Integer (W)
SHORT_SHORT_INTEGER	Byte Integer (B)
SYSTEM.UNSIGNED_QUADWORD	Quadword Unsigned (QU)
SYSTEM.UNSIGNED_LONGWORD	Longword Unsigned (LU)
SYSTEM.UNSIGNED_WORD	Word Unsigned (WU)
SYSTEM.UNSIGNED_BYTE	Byte Unsigned (BU)
FLOAT	F_Floating (F)
SYSTEM.F_FLOAT	F_Floating (F)
SYSTEM.D_FLOAT	D_Floating (D)

ADA Type	VAX Type Name
LONG_FLOAT	D_Floating (D), if pragma LONG_FLOAT(D_FLOAT) is in effect. G_Floating (G), if pragma LONG_FLOAT(G_FLOAT) is in effect.
SYSTEM.G_FLOAT	G_Floating (G)
SYSTEM.H_FLOAT	H_Floating (H)
LONG_LONG_FLOAT	H_Floating (H)
Fixed	(None)
STRING	ASCII Text (T)
BOOLEAN	Aligned Bit String (V)
BOOLEAN	Unaligned Bit String (VU)
Enumeration	For any enumeration type whose value fits into an unsigned byte or word: Byte Unsigned (BU) or Word Unsigned (WU), respectively. Otherwise: No corresponding VAX data type.
Arrays	(None)
Records	(None)
Access (pointers)	(None)
Tasks	(None)

E.1.4 Supported ADA Predefined Attributes

Supported ADA predefined attributes follow.

Attribute	Debugger Support
P'CONSTRAINED	For a prefix P that denotes a record object with discriminants. The value of P'CONSTRAINED reflects the current state of P (constrained or unconstrained).
P'FIRST	For a prefix P that denotes an enumeration type or a subtype of an enumeration type. Yields the lower bound of P.
P'FIRST	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype. Yields the lower bound of the first index range.
P'FIRST(N)	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype. Yields the lower bound of the N-th index range.
P'LAST	For a prefix P that denotes an enumeration type, or a subtype of an enumeration type. Yields the upper bound of P.
P'LAST	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype. Yields the upper bound of the first index range.

Attribute	Debugger Support
P'LAST(N)	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype. Yields the upper bound of the N-th index range.
P'LENGTH	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype. Yields the number of values of the first index range (zero for a null range).
P'LENGTH(N)	For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype. Yields the number of values of the N-th index range (zero for a null range).
P'POS(X)	For a prefix P that denotes an enumeration type or a subtype of an enumeration type. Yields the position number of the value X. The first position is 0.
P'PRED(X)	For a prefix P that denotes an enumeration type or a subtype of an enumeration type. Yields the value of type P which has a position number one less than that of X.
P'SIZE	For a prefix P that denotes an object. Yields the number of bits allocated to hold the object.
P'SUCC(X)	For a prefix P that denotes an enumeration type or a subtype of an enumeration type. Yields the value of type P which has a position number one more than that of X.
P'VAL(N)	For a prefix P that denotes an enumeration type or a subtype of an enumeration type. Yields the value of type P which has the position number N. The first position is 0.

E.1.5 Support for ADA Tasking Programs and Events

E.1.5.1 Task States

The following task-state keywords may be used with the SHOW TASK /STATE command:

Task State	Description
RUNNING	Currently running on the processor. This is the active task.
READY	Eligible to execute and waiting for the processor to be made available.
SUSPENDED	Suspended—that is, waiting for an event rather than for the availability of the processor. For example, when a task is created, it remains in the suspended state until it is activated.
TERMINATED	Terminated.

E.1.5.2 Task Substates

The following task-substate keywords may appear in a SHOW TASK display:

Task Substate	Description
Abnormal	Task has been aborted.
Accept	Task is waiting at an **accept** statement that is not inside a **select** statement.
Activating	Task is elaborating its declarative part.
Activating tasks	Task is waiting for tasks it has created to finish activating.
Completed [abn]	Task is completed due to an **abort** statement, but is not yet terminated. In Ada, a task awaiting dependent tasks at its "end" is called "completed". After the dependent tasks are terminated, the state changes to terminated.
Completed [exc]	Task is completed due to an unhandled exception, but is not yet terminated. In Ada, a task awaiting dependent tasks at its "end" is called "completed". After the dependent tasks are terminated, the state changes to terminated.
Completed	Task is completed. No **abort** statement was issued, and no unhandled exception occurred.
Delay	Task is waiting at a **delay** statement.
Dependents	Task is waiting for dependent tasks to terminate.
Dependents [exc]	Task is waiting for dependent tasks to allow an unhandled exception to propagate.
Entry call	Task is waiting for its **entry** call to be accepted.
Invalid state	There is a bug in the VAX Ada run-time library.
I/O or AST	Task is waiting for I/O completion or some AST.
Not yet activated	Task is waiting to be activated by the task that created it.
Select or delay	Task is waiting at a **select** statement with a **delay** alternative.
Select or term.	Task is waiting at a **select** statement with a **terminate** alternative.
Select	Task is waiting at a **select** statement with neither an **else**, **delay**, or **terminate** alternative.
Shared resource	Task is waiting for an internal shared resource.
Terminated [abn]	Task was terminated by an **abort**.
Terminated [exc]	Task was terminated because of an unhandled exception.
Terminated	Task terminated normally.
Timed entry call	Task is waiting in a timed **entry** call.

E.1.5.3 Supported ADA Events

The following ADA event keywords may be used with the /EVENT qualifier of the SET BREAK, SET TRACE, CANCEL BREAK, and CANCEL TRACE commands. You can also display these event keywords with the SHOW EVENT_FACILITY command.

Exception-Related Events

Event Name	Description
HANDLED	Triggers when an exception is about to be handled in some Ada exception handler, including an **others** handler (see Chapter 7).
HANDLED_OTHERS	Triggers *only* when an exception is about to be handled in an **others** Ada exception handler (see Chapter 7).

Task Exception-Related Events

Event Name	Description
RENDEZVOUS_EXCEPTION	Triggers when an exception begins to propagate out of a rendezvous.
DEPENDENTS_EXCEPTION	Triggers when an exception causes a task to wait for dependent tasks in some scope. Often immediately precedes a deadlock.

Task Termination Events

Event Name	Description
TERMINATED	Triggers when a task is terminating, whether normally, by abort, or by exception.
EXCEPTION_TERMINATED	Triggers when a task is terminating due to an exception.
ABORT_TERMINATED	Triggers when a task is terminating due to an abort.

Low-Level Task Scheduling Events

Event Name	Description
RUN	Triggers when a task is about to run.
PREEMPTED	Triggers when a task is being preempted from the RUN state.
ACTIVATING	Triggers when a task is about to begin its activation (that is, at the beginning of the elaboration of the declarative part of its task body).
SUSPENDED	Triggers when a task is about to be suspended.

E.2 Debugger Support for BASIC

This section includes information about debugger support for BASIC.

E.2.1 Supported BASIC Operators in Language Expressions

Supported BASIC operators in language expressions follow.

Kind	Symbol	Function
Prefix	+	Unary plus
Prefix	-	Unary minus (negation)
Infix	+	Addition, String concatenation
Infix	–	Subtraction
Infix	*	Multiplication
Infix	/	Division
Infix	**	Exponentiation
Infix	^	Exponentiation
Infix	=	Equal to
Infix	<>	Not equal to
Infix	> <	Not equal to
Infix	>	Greater than
Infix	> =	Greater than or equal to
Infix	=>	Greater than or equal to
Infix	<	Less than
Infix	<=	Less than or equal to
Infix	= <	Less than or equal to
Prefix	NOT	Bit-wise NOT
Infix	AND	Bit-wise AND
Infix	OR	Bit-wise OR
Infix	XOR	Bit-wise exclusive OR
Infix	IMP	Bit-wise implication
Infix	EQV	Bit-wise equivalence

E.2.2 Supported Constructs in Language and Address Expressions for BASIC

Supported constructs in language and address expressions for BASIC follow.

Symbol	Construct
( )	Subscripting
::	Record component selection

E.2.3 Supported BASIC Data Types

Supported BASIC data types follow.

BASIC Type	VAX Type Name
BYTE	Byte Integer (B)
WORD	Word Integer (W)
LONG	Longword Integer (L)
SINGLE	F_Floating (F)
DOUBLE	D_Floating (D)
GFLOAT	G_Floating (G)
HFLOAT	H_Floating (H)
DECIMAL	Packed Decimal (P)
STRING	ASCII Text (T)
RFA	(None)
Arrays	(None)
Records	(None)

Notes

1 Expressions that overflow in the BASIC language will not necessarily overflow when evaluated by the debugger. The debugger will try to compute a numerically correct result, even when the BASIC rules call for overflows. This difference is particularly likely to affect DECIMAL computations.

2 BASIC constants of the forms [radix]"numeric-string"[type] (such as "12.34"GFLOAT) or n% (such as 25% for integer 25) are not supported in debugger expressions.

E.3 Debugger Support for BLISS

This section includes information about debugger support for BLISS.

E.3.1 Supported BLISS Operators in Language Expressions

Supported BLISS operators in language expressions follow.

Kind	Symbol	Function
Prefix	.	Indirection
Prefix	+	Unary plus
Prefix	–	Unary minus (negation)
Infix	+	Addition
Infix	-	Subtraction
Infix	*	Multiplication
Infix	/	Division
Infix	MOD	Remainder
Infix	^	Left shift
Infix	EQL	Equal to

Kind	Symbol	Function
Infix	EQLU	Equal to
Infix	EQLA	Equal to
Infix	NEQ	Not equal to
Infix	NEQU	Not equal to
Infix	NEQA	Not equal to
Infix	GTR	Greater than
Infix	GTRU	Greater than unsigned
Infix	GTRA	Greater than unsigned
Infix	GEQ	Greater than or equal to
Infix	GEQU	Greater than or equal to unsigned
Infix	GEQA	Greater than or equal to unsigned
Infix	LSS	Less than
Infix	LSSU	Less than unsigned
Infix	LSSA	Less than unsigned
Infix	LEQ	Less than or equal to
Infix	LEQU	Less than or equal to unsigned
Infix	LEQA	Less than or equal to unsigned
Prefix	NOT	Bit-wise NOT
Infix	AND	Bit-wise AND
Infix	OR	Bit-wise OR
Infix	XOR	Bit-wise exclusive OR
Infix	EQV	Bit-wise equivalence

E.3.2 Supported Constructs in Language and Address Expressions for BLISS

Supported constructs in language and address expressions for BLISS follow.

Symbol	Construct
[ ]	Subscripting
[fldname]	Field selection
<p,s,e>	Bit field selection

E.3.3 Supported BLISS Data Types

Supported BLISS data types follow.

BLISS Type	VAX Type Name
BYTE	Byte Integer (B)
WORD	Word Integer (W)
LONG	Longword Integer (L)
BYTE UNSIGNED	Byte Unsigned (BU)
WORD UNSIGNED	Word Unsigned (WU)
LONG UNSIGNED	Longword Unsigned (LU)
VECTOR	(None)
BITVECTOR	(None)
BLOCK	(None)
BLOCKVECTOR	(None)
REF VECTOR	(None)
REF BITVECTOR	(None)
REF BLOCK	(None)
REF BLOCKVECTOR	(None)

E.4 Debugger Support for Language C

This section includes information about debugger support for C.

E.4.1 Supported C Operators in Language Expressions

Supported C operators in language expressions follow.

Kind	Symbol	Function
Prefix	*	Indirection
Prefix	&	Address of
Prefix	SIZEOF	Size of
Prefix	-	Unary minus (negation)
Infix	+	Addition
Infix	–	Subtraction
Infix	*	Multiplication
Infix	/	Division
Infix	%	Remainder
Infix	< <	Left shift
Infix	> >	Right shift
Infix	==	Equal to
Infix	!=	Not equal to
Infix	>	Greater than
Infix	> =	Greater than or equal to
Infix	<	Less than

Kind	Symbol	Function
Infix	<=	Less than or equal to
Prefix	(tilde)	Bit-wise NOT
Infix	&	Bit-wise AND
Infix	|	Bit-wise OR
Infix	^	Bit-wise exclusive OR
Prefix	!	Logical NOT
Infix	&&	Logical AND
Infix	||	Logical OR

E.4.2 Supported Constructs in Language and Address Expressions for C

Supported constructs in language and address expressions for C follow.

Symbol	Construct
[ ]	Subscripting
.	Structure component selection
->	Pointer dereferencing

E.4.3 Supported C Data Types

Supported C data types follow.

C Type	VAX Type Name
INT	Longword Integer (L)
UNSIGNED INT	Longword Unsigned (LU)
SHORT INT	Word Integer (W)
UNSIGNED SHORT INT	Word Unsigned (WU)
CHAR	Byte Integer (B)
UNSIGNED CHAR	Byte Unsigned (BU)
FLOAT	F_Floating (F)
DOUBLE	D_Floating (D)
ENUM	(None)
STRUCT	(None)
UNION	(None)
Pointers	(None)
Arrays	(None)

Notes

1 Symbol names are case-sensitive for language C, meaning that uppercase and lowercase letters are treated as different characters.

2 Since the exclamation point (!) is an operator in C, it cannot be used as the comment delimiter. When the language is set to C, the debugger instead accepts /* as the comment delimiter. The comment continues to the end of the current line. (A matching */ is neither needed nor recognized.) To permit debugger log files to be used as debugger input, the debugger still recognizes ! as a comment delimiter if it is the first non-blank character on a line.

3 The debugger accepts the prefix asterisk (*) as an indirection operator in both C language expressions and debugger address expressions. In address expressions, prefix "*" is synonymous to prefix "." or "@" when the language is set to C.

4 The debugger does not support any of the assignment operators in C (or any other language) in order to prevent unintended modifications to the program being debugged. Hence such operators as =, +=, -=, ++, and—are not recognized. If you wish to alter the contents of a memory location, you must do so with an explicit DEPOSIT command.

E.5 Debugger Support for Language COBOL

This section includes information about debugger support for COBOL.

E.5.1 Supported COBOL Operators in Language Expressions

Supported COBOL operators in language expressions follow.

Kind	Symbol	Function
Prefix	+	Unary plus
Prefix	–	Unary minus (negation)
Infix	+	Addition
Infix	–	Subtraction
Infix	*	Multiplication
Infix	/	Division
Infix	**	Exponentiation
Infix	=	Equal to
Infix	NOT =	Not equal to
Infix	>	Greater than
Infix	NOT <	Greater than or equal to
Infix	<	Less than
Infix	NOT >	Less than or equal to
Infix	NOT	Logical NOT
Infix	AND	Logical AND
Infix	OR	Logical OR

E.5.2 Supported Constructs in Language and Address Expressions for COBOL

Supported constructs in language and address expressions for COBOL follow.

Symbol	Construct
( )	Subscripting
OF	Record component selection
IN	Record component selection

E.5.3 Supported COBOL Data Types

Supported COBOL data types follow.

COBOL Type	VAX Type Name
COMP	Longword Integer (L,LU)
COMP	Word Integer (W,WU)
COMP	Byte Integer (B,BU)
COMP	Quadword Integer (Q,QU)
COMP-1	F_Floating (F)
COMP-2	D_Floating (D)
COMP-3	Packed Decimal (P)
INDEX	Longword Integer (L)
Alphanumeric	ASCII Text (T)
Records	(None)
Numeric Unsigned (NU)	(None)
Leading Separate Sign (NL)	(None)
Leading Overpunched Sign (NLO)	(None)
Trailing Separate Sign (NR)	(None)
Trailing Overpunched Sign (NRO)	(None)

Notes

1 The debugger can show source text included in a program with the COPY or COPY REPLACING verb. However, when COPY REPLACING is used, the debugger always shows the original source text as it appeared before text replacement. In other words, the original source file is shown instead of the modified source text generated by the COPY REPLACING verb.

2 The debugger cannot show the original source lines associated with the code for a REPORT section. You can see the DATA SECTION source lines associated with a REPORT, but no source lines are associated with the compiled code that generates the report.

E.6 Debugger Support for Language DIBOL

This section includes information about debugger support for DIBOL.

E.6.1 Supported DIBOL Operators in Language Expressions

Supported DIBOL operators in language expressions follow.

Kind	Symbol	Function
Prefix	#	Round
Prefix	+	Unary plus
Prefix	–	Unary minus (negation)
Infix	+	Addition
Infix	–	Subtraction
Infix	*	Multiplication
Infix	/	Division
Infix	//	Division with fractional result
Infix	.EQ.	Equal to
Infix	.NE.	Not equal to
Infix	.GT.	Greater than
Infix	.GE.	Greater than or equal to
Infix	.LT.	Less than
Infix	.LE.	Less than or equal to
Infix	.NOT.	Logical NOT
Infix	.AND.	Logical AND
Infix	.OR.	Logical OR
Infix	.XOR.	Exclusive OR

E.6.2 Supported Constructs in Language and Address Expressions for DIBOL

Supported constructs in language and address expressions for DIBOL follow.

Symbol	Construct
( )	Substring
[ ]	Subscripting
.	Record component selection

E.6.3 Supported DIBOL Data Types

Supported DIBOL data types follow.

DIBOL Type	VAX Type Name
I1	Byte Integer (B)
I2	Word Integer (W)
I4	Longword Integer (L)
Pn	Packed Decimal String (P)
Pn.m	Packed Decimal String (P)
Dn	Numeric String, Zoned Sign (NZ)
Dn.m	Numeric String, Zoned Sign (NZ)
An	ASCII Text (T)
Arrays	(None)
Records	(None)

E.7 Debugger Support for Language FORTRAN

This section includes information about debugger support for FORTRAN.

E.7.1 Supported FORTRAN Operators in Language Expressions

Supported FORTRAN operators in language expressions follow.

Kind	Symbol	Function
Prefix	+	Unary plus
Prefix	–	Unary minus (negation)
Infix	+	Addition
Infix	–	Subtraction
Infix	*	Multiplication
Infix	/	Division
Infix	**	Exponentiation
Infix	//	Concatenation
Infix	.EQ.	Equal to
Infix	.NE.	Not equal to
Infix	.GT.	Greater than
Infix	.GE.	Greater than or equal to
Infix	.LT.	Less than
Infix	.LE.	Less than or equal to
Prefix	.NOT.	Logical NOT
Infix	.AND.	Logical AND
Infix	.OR.	Logical OR
Infix	.XOR.	Exclusive OR
Infix	.EQV.	Equivalence
Infix	.NEQV.	Exclusive OR

E.7.2 Supported Constructs in Language and Address Expressions for FORTRAN

Supported constructs in language and address expressions for FORTRAN follow.

Symbol	Construct
( )	Subscripting
.	Record component selection

E.7.3 Supported FORTRAN Predefined Symbols

Supported FORTRAN predefined symbols follow.

Symbol	Description
.TRUE.	Logical True
.FALSE.	Logical False

E.7.4 Supported FORTRAN Data Types

Supported FORTRAN data types follow.

FORTRAN Type	VAX Type Name
LOGICAL*1	Byte Unsigned (BU)
LOGICAL*2	Word Unsigned (WU)
LOGICAL*4	Longword Unsigned (LU)
INTEGER*2	Word Integer (W)
INTEGER*4	Longword Integer (L)
REAL*4	F_Floating (F)
REAL*8	D_Floating (D)
REAL*8	G_Floating (G)
REAL*16	H_Floating (H)
COMPLEX*8	F_Complex (FC)
COMPLEX*16	D_Complex (DC)
COMPLEX*16	G_Complex (GC)
CHARACTER	ASCII Text (T)
Arrays	(None)
Records	(None)

Notes

1 Even though the VAX type codes for unsigned integers (BU, WU, LU) are used internally to describe the LOGICAL data types, the debugger (like the compiler) treats LOGICAL variables and values as being signed when used in language expressions.

2 The debugger prints the numeric values of LOGICAL variables or expressions instead of TRUE or FALSE. Normally, only the low-order bit of a LOGICAL variable or value is significant (0 is FALSE and 1 is TRUE). However, VAX FORTRAN does allow all bits in a LOGICAL value to be manipulated and LOGICAL values can be used in integer expressions. For this reason, it is at times necessary to see the entire integer value of a LOGICAL variable or expression, and that is what the debugger shows.

3 COMPLEX constants such as (1.0,2.0) are not supported in debugger expressions.

E.8 Debugger Support for Language MACRO

This section includes information about debugger support for MACRO.

E.8.1 Supported Operators in Language Expressions

Language MACRO does not have expressions in the same sense as high-level languages. Only assembly-time expressions and only a limited set of operators are accepted. To permit the MACRO programmer to use expressions at debug-time as freely as in other languages, the debugger accepts a number of operators in MACRO language expressions that are not found in MACRO itself. In particular, the debugger accepts a complete set of comparison and boolean operators modeled after BLISS. It also accepts the indirection operator and the normal arithmetic operators.

Kind	Symbol	Function
Prefix	@	Indirection
Prefix	.	Indirection
Prefix	+	Unary plus
Prefix	–	Unary minus (negation)
Infix	+	Addition
Infix	–	Subtraction
Infix	*	Multiplication
Infix	/	Division
Infix	MOD	Remainder
Infix	@	Left shift
Infix	EQL	Equal to
Infix	EQLU	Equal to
Infix	NEQ	Not equal to
Infix	NEQU	Not equal to
Infix	GTR	Greater than
Infix	GTRU	Greater than unsigned
Infix	GEQ	Greater than or equal to
Infix	GEQU	Greater than or equal to unsigned
Infix	LSS	Less than

Kind	Symbol	Function
Infix	LSSU	Less than unsigned
Infix	LEQ	Less than or equal to
Infix	LEQU	Less than or equal to unsigned
Prefix	NOT	Bit-wise NOT
Infix	AND	Bit-wise AND
Infix	OR	Bit-wise OR
Infix	XOR	Bit-wise exclusive OR
Infix	EQV	Bit-wise equivalence

E.8.2 Supported Constructs in Language and Address Expressions for MACRO

Supported constructs in language and address expressions for MACRO follow.

Symbol	Construct
<p,s,e>	Bitfield selection as in BLISS

E.8.3 Supported MACRO Data Types

Supported MACRO data types follow.

MACRO Type	VAX Type Name
(Not applicable)	Byte Unsigned (BU)
(Not applicable)	Word Unsigned (WU)
(Not applicable)	Longword Unsigned (LU)
(Not applicable)	Byte Integer (B)
(Not applicable)	Word Integer (W)
(Not applicable)	Longword Integer (L)

E.9 Debugger Support for Language PASCAL

This section includes information about debugger support for PASCAL.

E.9.1 Supported PASCAL Operators in Language Expressions

Supported PASCAL operators in language expressions follow.

Kind	Symbol	Function
Prefix	+	Unary plus
Prefix	–	Unary minus (negation)
Infix	+	Addition, concatenation
Infix	–	Subtraction
Infix	*	Multiplication
Infix	/	Real division
Infix	DIV	Integer division
Infix	MOD	Modulus
Infix	REM	Remainder
Infix	**	Exponentiation
Infix	IN	Set membership
Infix	=	Equal to
Infix	<>	Not equal to
Infix	>	Greater than
Infix	> =	Greater than or equal to
Infix	<	Less than
Infix	<=	Less than or equal to
Prefix	NOT	Logical NOT
Infix	AND	Logical AND
Infix	OR	Logical OR

E.9.2 Supported Constructs in Language and Address Expressions for PASCAL

Supported constructs in language and address expressions for PASCAL follow.

Symbol	Construct
[ ]	Subscripting
.	Record component selection
^	Pointer dereferencing

E.9.3 Supported PASCAL Predefined Symbols

Supported PASCAL predefined symbols follow.

Symbol	Meaning
TRUE	Boolean True
FALSE	Boolean False
NIL	Nil pointer

E.9.4 Supported PASCAL Built-In Functions

Supported PASCAL built-in functions follow.

Symbol	Meaning
SUCC	Logical successor
PRED	Logical predecessor

E.9.5 Supported PASCAL Data Types

Supported PASCAL data types follow.

PASCAL Type	VAX Type Name
INTEGER	Longword Integer (L)
INTEGER	Word Integer (W,WU)
INTEGER	Byte Integer (B,BU)
UNSIGNED	Longword Unsigned (LU)
UNSIGNED	Word Unsigned (WU)
UNSIGNED	Byte Unsigned (BU)
SINGLE	F_Floating (F)
DOUBLE	D_Floating (D)
DOUBLE	G_Floating (G)
QUADRUPLE	H_Floating (H)
BOOLEAN	(None)
CHAR	ASCII Text (T)
VARYING OF CHAR	Varying Text (VT)
SET	(None)
FILE	(None)
Enumerations	(None)
Subranges	(None)
Typed Pointers	(None)
Arrays	(None)
Records	(None)
Variant records	(None)

Note

The debugger accepts PASCAL set constants such as [1,2,5,8..10] or [RED, BLUE] in PASCAL language expressions.

E.10 Debugger Support for Language PL/I

This section includes information about debugger support for PL/I.

E.10.1 Supported PL/I Operators in Language Expressions

Supported PL/I operators in language expressions follow.

Kind	Symbol	Function
Prefix	+	Unary plus
Prefix	–	Unary minus (negation)
Infix	+	Addition
Infix	–	Subtraction
Infix	*	Multiplication
Infix	/	Division
Infix	**	Exponentiation
Infix	||	Concatenation
Infix	=	Equal to
Infix	^=	Not equal to
Infix	>	Greater than
Infix	> =	Greater than or equal to
Infix	^ <	Greater than or equal to
Infix	<	Less than
Infix	<=	Less than or equal to
Infix	^>	Less than or equal to
Prefix	^	Bit-wise NOT
Infix	&	Bit-wise AND
Infix	|	Bit-wise OR

E.10.2 Supported Constructs in Language and Address Expressions for PL/I

Supported constructs in language and address expressions for PL/I follow.

Symbol	Construct
( )	Subscripting
.	Structure component selection
->	Pointer dereferencing

E.10.3 Supported PL/I Data Types

Supported PL/I data types follow.

PL/I Type	VAX Type Name
FIXED BINARY	Longword Integer (L)
FIXED DECIMAL	Packed Decimal (P)
FLOAT BINARY	F_Floating (F)
FLOAT DECIMAL	F_Floating (F)
FLOAT BIN/DEC	D_Floating (D)
FLOAT BIN/DEC	G_Floating (G)
FLOAT BIN/DEC	H_Floating (H)
BIT	Bit (V)
BIT	Bit Unaligned (VU)
CHARACTER	ASCII Text (T)
CHARACTER VARYING	Varying Text (VT)
FILE	(None)
Labels	(None)
Pointers	(None)
Arrays	(None)
Structures	(None)

Note

The debugger treats all numeric constants of the form n or n.n in PL/I language expressions as packed decimal constants, not integer or floating-point constants, in order to conform to PL/I language rules. The internal representation of 10 is therefore 0C01 hexadecimal, not 0A hexadecimal. You can enter floating-point constants using the syntax nEn or n.nEn. There is no PL/I syntax for entering constants whose internal representation is Longword Integer. This limitation is not normally significant when debugging since the debugger supports the PL/I type conversion rules. However, it is possible to enter integer constants by using the debugger's %HEX, %OCT, and %BIN operators.

E.11 Debugger Support for Language RPG

This section includes information about debugger support for RPG.

E.11.1 Supported RPG Operators in Language Expressions

Supported RPG operators in language expressions follow.

Kind	Symbol	Function
Prefix	+	Unary plus
Prefix	–	Unary minus (negation)
Infix	+	Addition
Infix	–	Subtraction
Infix	*	Multiplication

Kind	Symbol	Function
Infix	/	Division
Infix	=	Equal to
Infix	NOT =	Not equal to
Infix	>	Greater than
Infix	NOT <	Greater than or equal to
Infix	<	Less than
Infix	NOT >	Less than or equal to
Prefix	NOT	Logical NOT
Infix	AND	Logical AND
Infix	OR	Logical OR

E.11.2 Supported Constructs in Language and Address Expressions for RPG

Supported constructs in language and address expressions for RPG follow.

Symbol	Construct
( )	Subscripting

E.11.3 Supported RPG Data Types

Supported RPG data types follow.

RPG Type	VAX Type Name
Longword	Longword Integer (L)
Word	Word Integer (W)
Packed Decimal	Packed Decimal (P)
Character	ASCII Text (T)
Overpunched Decimal	Right Overpunched Sign (NRO)
Arrays	(None)
Tables	(None)

Note

The debugger supports access to all RPG indicators and labels used in the current program. You can thus examine labels such as *DETL and indicators such as *INLR and *IN01 through *IN99.

E.12 Debugger Support for SCAN

This section includes information about debugger support for SCAN.

E.12.1 Supported SCAN Operators in Language Expressions

Supported SCAN operators in language expressions follow.

Kind	Symbol	Function
Prefix	+	Unary plus
Prefix	–	Unary minus (negation)
Infix	+	Addition
Infix	–	Subtraction
Infix	*	Multiplication
Infix	/	Division
Infix	&	Concatenation
Infix	=	Equal to
Infix	<>	Not equal to
Infix	>	Greater than
Infix	>=	Greater than or equal to
Infix	<	Less than
Infix	<=	Less than or equal to
Prefix	NOT	Complement
Infix	AND	Intersection
Infix	OR	Union
Infix	XOR	Exclusive OR

E.12.2 Supported Constructs in Language and Address Expressions for SCAN

Supported constructs in language and address expressions for SCAN follow.

Symbol	Construct
( )	Subscripting
.	Record component selection
->	Pointer dereferencing

E.12.3 Supported SCAN Data Types

Supported SCAN data types follow.

SCAN Type	VAX Type Name
BOOLEAN	(None)
INTEGER	Longword Integer (L)
POINTER	(None)
FIXED STRING (n)	TEXT with CLASS=S

SCAN Type	VAX Type Name
VARYING STRING (n)	TEXT with CLASS=VS
DYNAMIC STRING	TEXT with CLASS=D
TREE	(None)
TREEPTR	(None)
RECORD	(None)
OVERLAY	(None)

Notes

1 There is no specific support for the following datatypes: FILE, TOKEN, GROUP, SET. Examining a FILL variable will display the contents of the specified variable as a string by default, and so may have little meaning. If the characteristics of the fill are known, then the appropriate qualifier (/HEX, and so on) applied to the command will produce a more meaningful display.

2 The following examples show how to examine SCAN TREE and TREEPTR variables. To dump an entire SCAN tree or subtree:

```
DBG> EXAMINE tree_variable([subscript], . . . )
```

To dump the contents of a SCAN subtree:

```
DBG> EXAMINE treeptr_variable
```

To dump an entire SCAN subtree:

```
DBG> EXAMINE treeptr_variable->
```

3 DEPOSIT is not supported for SCAN TREE variables. You may set breakpoints on any SCAN label, line number, MACRO, or PROCEDURE.

E.12.4 Supported SCAN Events

The following SCAN event keywords may be used with the /EVENT qualifier of the SET BREAK, SET TRACE, CANCEL BREAK, and CANCEL TRACE commands. You can also display these event keywords with the SHOW EVENT_FACILITY command.

Event Keyword	Description
TOKEN	A token is built.
PICTURE	An operand in a picture is being matched.
INPUT	A new line of the input stream is read.
OUTPUT	A new line of the output stream is written.
TRIGGER	A trigger macro is starting or terminating.
SYNTAX	A syntax macro is starting or terminating.
ERROR	Picture matching error recovery is starting or terminating.

E.13 Debugger Support for Language UNKNOWN

This section includes information about debugger support for UNKNOWN.

E.13.1 Supported Operators in Language Expressions

Supported operators in language expressions follow.

Kind	Symbol	Function
Prefix	+	Unary plus
Prefix	–	Unary minus (negation)
Infix	+	Addition
Infix	–	Subtraction
Infix	*	Multiplication
Infix	/	Division
Infix	**	Exponentiation
Infix	&	Concatenation
Infix	//	Concatenation
Infix	=	Equal to
Infix	<>	Not equal to
Infix	/=	Not equal to
Infix	>	Greater than
Infix	> =	Greater than or equal to
Infix	<	Less than
Infix	<=	Less than or equal to
Infix	EQL	Equal to
Infix	NEQ	Not equal to
Infix	GTR	Greater than
Infix	GEQ	Greater than or equal to
Infix	LSS	Less than
Infix	LEQ	Less than or equal to
Prefix	NOT	Logical NOT
Infix	AND	Logical AND
Infix	OR	Logical OR
Infix	XOR	Exclusive OR
Infix	EQV	Equivalence

E.13.2 Supported Constructs in Language and Address Expressions for UNKNOWN

Supported constructs in language and address expressions for UNKNOWN follow.

Symbol	Construct
[ ]	Subscripting
( )	Subscripting
.	Record component selection
^	Pointer dereferencing

E.13.3 Supported UNKNOWN Data Types

When the language is set to UNKNOWN, the debugger understands all data types accepted by other languages except a few very language-specific types, such a picture types and file types. In UNKNOWN language expressions, the debugger accepts most scalar VAX Standard data types.

Notes

1 For language UNKNOWN, the debugger accepts the dot-notation for record component selection. If C is a component of a record B which in turn is a component of a record A, C can be referenced as "A.B.C". Subscripts can be attached to any array components; if B is an array, for instance, C may be referenced as "A.B[2,3].C".

2 For language UNKNOWN, the debugger accepts both round and square subscript parentheses. Hence A[2,3] and A(2,3) are equivalent.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

VAX/VMS
Debugger Reference
Manual
AA–Z411C–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 user/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

— — 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

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

— — Do Not Tear - Fold Here —

Cut Along Dotted Line

VAX/VMS
Debugger Reference
Manual
AA–Z411C–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 user/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

— — 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

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

— — Do Not Tear - Fold Here —

Cut Along Dotted Line