VMS Debugger Manual

Order Number: AA–LA59A–TE

April 1988

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

Revision/Update Information: This document supersedes the *VAX/VMS Debugger Reference Manual,* Version 4.4.

Software Version: VMS Version 5.0

digital equipment corporation
maynard, massachusetts

April 1988

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™

ZK4538

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*	CANADA	INTERNATIONAL
Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061	Digital Equipment of Canada Ltd. 100 Herzberg Road Kanata, Ontario K2K 2A6 Attn: Direct Order Desk	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.

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by DIGITAL. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use DIGITAL-supported devices, such as the LN03 laser printer and PostScript™ printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

™ PostScript is a trademark of Adobe Systems, Inc.

Contents

PREFACE xix

NEW AND CHANGED FEATURES xxi

PART I USING THE VMS DEBUGGER

CHAPTER 1 INTRODUCTION TO THE VMS DEBUGGER 1–1

1.1 OVERVIEW OF THE DEBUGGER 1–1
1.1.1 Functional Features 1–2
1.1.2 Convenience Features 1–3

1.2 GETTING STARTED WITH THE DEBUGGER 1–5
1.2.1 Compiling and Linking to Prepare for Debugging 1–5
1.2.2 Starting and Terminating a Debugging Session 1–6
1.2.3 Entering Debugger Commands 1–7
1.2.4 Viewing Your Source Code 1–7
1.2.4.1 Noscreen Mode • 1–8
1.2.4.2 Screen Mode • 1–9
1.2.5 Controlling and Monitoring Program Execution 1–11
1.2.5.1 Starting and Resuming Program Execution • 1–11
1.2.5.1.1 The GO Command • 1–11
1.2.5.1.2 The STEP Command • 1–12
1.2.5.2 Determining Where Execution Is Currently Suspended • 1–13
1.2.5.3 Suspending Program Execution • 1–13
1.2.5.4 Tracing Program Execution • 1–15
1.2.5.5 Monitoring Changes in Variables • 1–16
1.2.6 Examining and Manipulating Program Data 1–17
1.2.6.1 Displaying the Value of a Variable • 1–17
1.2.6.2 Changing the Value of a Variable • 1–18
1.2.6.3 Evaluating Expressions • 1–19
1.2.7 Controlling Symbol References 1–20
1.2.7.1 Module Setting • 1–20
1.2.7.2 Resolving Multiply-Defined Symbols • 1–21
1.2.8 A Sample Debugging Session 1–22

1.3 DEBUGGER COMMAND SUMMARY 1–25

1.3.1	Starting and Terminating a Debugging Session	1–25
1.3.2	Controlling and Monitoring Program Execution	1–25
1.3.3	Examining and Manipulating Data	1–26
1.3.4	Controlling Type Selection and Radix	1–26
1.3.5	Controlling Symbol Lookup and Symbolization	1–26
1.3.6	Displaying Source Code	1–27
1.3.7	Screen Mode	1–27
1.3.8	Source Editing	1–28
1.3.9	Defining Symbols	1–28
1.3.10	Keypad Mode	1–28
1.3.11	Command Procedures, Log Files, and Initialization Files	1–29
1.3.12	Control Structures	1–29
1.3.13	Miscellaneous Commands	1–29

CHAPTER 2 STARTING AND CONTROLLING PROGRAM EXECUTION 2–1

2.1	STARTING, TERMINATING, AND INTERRUPTING A DEBUGGING SESSION	2–1
2.1.1	Invoking the Debugger with the DCL RUN Command	2–1
2.1.2	Invoking the Debugger with the DCL DEBUG Command	2–3
2.1.3	Terminating a Debugging Session	2–4
2.1.4	Interrupting and Resuming a Debugging Session	2–5
2.1.4.1	Interrupting with CTRL/Y • 2–5	
2.1.4.2	Interrupting with the SPAWN and ATTACH Commands • 2–6	
2.2	COMMANDS THAT CAUSE PROGRAM EXECUTION	2–7
2.3	USING THE STEP COMMAND	2–7
2.3.1	Changing the STEP Command Behavior	2–8
2.3.2	Stepping into and over Routines	2–9
2.4	SUSPENDING AND TRACING EXECUTION WITH BREAKPOINTS AND TRACEPOINTS	2–10
2.4.1	Setting Breakpoints or Tracepoints on Individual Program Locations	2–11
2.4.1.1	Specifying Symbolic Addresses • 2–11	
2.4.1.2	Specifying Locations in Virtual Memory • 2–13	
2.4.1.3	Obtaining and Symbolizing Virtual Memory Addresses • 2–13	
2.4.2	Setting Breakpoints or Tracepoints on Consecutive Lines or on Classes of Instructions	2–14
2.4.3	Controlling Debugger Action at Breakpoints or Tracepoints	2–15
2.4.4	Setting Breakpoints or Tracepoints on Exceptions	2–16

2.4.5	Setting Breakpoints or Tracepoints on Language-Specific Events	2–16
2.4.6	Canceling Breakpoints or Tracepoints	2–17
2.5	MONITORING CHANGES IN VARIABLES AND OTHER PROGRAM LOCATIONS	2–17
2.5.1	Watchpoint Options	2–19
2.5.2	Watching Nonstatic Variables	2–19
2.5.2.1	Execution Speed • 2–20	
2.5.2.2	Setting a Watchpoint on a Nonstatic Variable • 2–20	
2.5.2.3	Options for Watching Nonstatic Variables • 2–21	
2.6	HOW THE DEBUGGER CONTROLS PROGRAM EXECUTION	2–22
CHAPTER 3	EXAMINING AND MANIPULATING PROGRAM DATA	3–1
3.1	GENERAL CONCEPTS	3–1
3.1.1	Accessing Variables While Debugging	3–1
3.1.2	Using the EXAMINE Command	3–2
3.1.3	Using the DEPOSIT Command	3–3
3.1.4	Address Expressions and Their Associated Types	3–4
3.1.5	Evaluating Language Expressions	3–5
3.1.5.1	Using Variables in Language Expressions • 3–6	
3.1.5.2	Numeric Type Conversion by the Debugger • 3–7	
3.1.6	Address Expressions Compared to Language Expressions	3–7
3.1.7	Specifying the Current, Previous, and Next Entity	3–8
3.1.8	Language Dependencies and the Current Language	3–10
3.1.9	Specifying a Radix for Entering or Displaying Integer Data	3–10
3.1.10	Obtaining and Symbolizing Virtual Memory Addresses	3–12
3.2	EXAMINING AND DEPOSITING INTO VARIABLES	3–14
3.2.1	Scalar Types	3–14
3.2.2	ASCII String Types	3–16
3.2.3	Array Types	3–16
3.2.4	Record Types	3–18
3.2.5	Pointer (Access) Types	3–18
3.3	EXAMINING AND DEPOSITING VAX INSTRUCTIONS	3–19
3.3.1	Examining VAX Instructions	3–19
3.3.2	Depositing VAX Instructions	3–21
3.4	EXAMINING AND DEPOSITING REGISTER VALUES	3–22

3.4.1 The Processor Status Longword (PSL) 3–23

3.5 SPECIFYING A TYPE WHEN EXAMINING AND DEPOSITING 3–24
3.5.1 Defining a Type for Locations Without a Symbolic Name 3–24
3.5.2 Overriding the Current Type 3–25
3.5.2.1 Integer Types • 3–26
3.5.2.2 ASCII String Type • 3–26
3.5.2.3 User-Declared Types • 3–27

CHAPTER 4 CONTROLLING SYMBOL LOOKUP 4–1

4.1 CONTROLLING SYMBOL INFORMATION WHEN COMPILING AND LINKING 4–2
4.1.1 Compiling 4–2
4.1.2 Local and Global Symbols 4–3
4.1.3 Linking 4–4
4.1.4 Controlling Symbol Information in Debugged Images 4–5

4.2 SETTING AND CANCELING MODULES 4–5

4.3 RESOLVING MULTIPLY-DEFINED SYMBOLS 4–7
4.3.1 Scope and Symbol Lookup Conventions 4–7
4.3.2 Using SHOW SYMBOL and Path Names to Specify Symbols Uniquely 4–8
4.3.2.1 Simplifying Path Names • 4–9
4.3.2.2 Specifying Symbols in the Call Stack • 4–9
4.3.2.3 Specifying Global Symbols • 4–9
4.3.2.4 Specifying Routine Invocations • 4–9
4.3.3 Using SET SCOPE to Specify a Symbol Search Scope 4–10

4.4 DEBUGGING SHAREABLE IMAGES 4–11
4.4.1 Compiling and Linking Shareable Images for Debugging 4–11
4.4.2 Accessing Symbols in Shareable Images 4–13
4.4.2.1 Accessing Symbols in the PC Scope (Dynamic Mode) • 4–13
4.4.2.2 Accessing Symbols in Arbitrary Images • 4–13

CHAPTER 5 CONTROLLING THE DISPLAY OF SOURCE CODE 5–1

5.1 HOW THE DEBUGGER OBTAINS SOURCE CODE INFORMATION 5–1

5.2 SPECIFYING THE LOCATION OF SOURCE FILES 5–2

5.3 DISPLAYING SOURCE CODE BY SPECIFYING LINE NUMBERS 5–3

5.4 DISPLAYING SOURCE CODE BY SPECIFYING ADDRESS EXPRESSIONS 5–4

5.5 DISPLAYING SOURCE CODE BY SEARCHING FOR STRINGS 5–6

5.6 CONTROLLING SOURCE DISPLAY AFTER STEPPING AND AT EVENTPOINTS 5–7

5.7 SETTING MARGINS FOR SOURCE DISPLAY 5–8

CHAPTER 6 USING SCREEN MODE 6–1

6.1 CONCEPTS AND TERMINOLOGY 6–2

6.2 THE PREDEFINED DISPLAYS 6–4
6.2.1 The Predefined Source Display SRC 6–4
6.2.2 The Predefined Output Display OUT 6–5
6.2.3 The Predefined Prompt Display PROMPT 6–5
6.2.4 The Predefined Instruction Display INST 6–6
6.2.5 The Predefined Register Display REG 6–7

6.3 MANIPULATING EXISTING DISPLAYS 6–7
6.3.1 Scrolling a Display 6–8
6.3.2 Showing, Hiding, Removing, and Canceling a Display 6–9
6.3.3 Moving a Display Across the Screen 6–9
6.3.4 Expanding or Contracting a Display 6–10

6.4 CREATING A NEW DISPLAY 6–10

6.5	SPECIFYING A DISPLAY WINDOW	6–11
6.5.1	Specifying a Window in Terms of Lines and Columns	6–11
6.5.2	Predefined Windows	6–11
6.5.3	Creating a New Window Definition	6–12
6.6	SPECIFYING THE DISPLAY KIND	6–12
6.6.1	DO (command[; . . .]) Display Kind	6–13
6.6.2	INSTRUCTION Display Kind	6–13
6.6.3	INSTRUCTION (command) Display Kind	6–14
6.6.4	OUTPUT Display Kind	6–14
6.6.5	REGISTER Display Kind	6–15
6.6.6	SOURCE Display Kind	6–15
6.6.7	SOURCE (command) Display Kind	6–16
6.6.8	PROGRAM Display Kind	6–16
6.7	ASSIGNING DISPLAY ATTRIBUTES	6–16
6.8	A SAMPLE DISPLAY CONFIGURATION	6–18
6.9	SAVING DISPLAYS AND THE SCREEN STATE	6–19
6.10	CHANGING THE SCREEN HEIGHT AND WIDTH	6–20
CHAPTER 7	ADDITIONAL CONVENIENCE FEATURES	7–1
7.1	USING DEBUGGER COMMAND PROCEDURES	7–1
7.1.1	Basic Conventions	7–1
7.1.2	Passing Parameters to Command Procedures	7–2
7.2	USING A DEBUGGER INITIALIZATION FILE	7–4
7.3	LOGGING A DEBUGGING SESSION INTO A FILE	7–5
7.4	DEFINING SYMBOLS FOR COMMANDS, ADDRESS EXPRESSIONS, AND VALUES	7–6
7.4.1	Defining Symbols for Commands	7–6
7.4.2	Defining Symbols for Address Expressions	7–7
7.4.3	Defining Symbols for Values	7–7

7.5	ASSIGNING COMMANDS TO FUNCTION KEYS	7–7
7.5.1	Basic Conventions	7–8
7.5.2	More Advanced Techniques	7–9
7.6	USING CONTROL STRUCTURES TO ENTER COMMANDS	7–9
7.6.1	FOR Command	7–9
7.6.2	IF Command	7–10
7.6.3	REPEAT Command	7–10
7.6.4	WHILE Command	7–10
7.6.5	EXITLOOP Command	7–10
7.7	CALLING ROUTINES LINKED WITH YOUR PROGRAM	7–11
CHAPTER 8	DEBUGGING SPECIAL CASES	8–1
8.1	DEBUGGING OPTIMIZED CODE	8–1
8.1.1	Eliminated Variables	8–2
8.1.2	Coding Order	8–3
8.1.3	Use of Registers	8–4
8.1.4	Use of Condition Codes	8–5
8.2	DEBUGGING SCREEN-ORIENTED PROGRAMS	8–5
8.3	DEBUGGING MULTILANGUAGE PROGRAMS	8–7
8.3.1	Controlling the Current Debugger Language	8–7
8.3.2	Specific Differences Among Languages	8–8
8.3.2.1	Default Radix • 8–8	
8.3.2.2	Evaluating Language Expressions • 8–8	
8.3.2.3	Arrays and Records • 8–9	
8.3.2.4	Case Sensitivity • 8–9	
8.3.2.5	Initialization Code • 8–9	
8.3.2.6	Ada Predefined Breakpoints • 8–10	
8.4	DEBUGGING EXCEPTIONS AND CONDITION HANDLERS	8–10
8.4.1	Setting Breakpoints or Tracepoints on Exceptions	8–11
8.4.2	Resuming Execution at an Exception Breakpoint	8–11
8.4.3	Effect of Debugger on Condition Handling	8–13
8.4.3.1	Primary Handler • 8–14	
8.4.3.2	Secondary Handler • 8–14	
8.4.3.3	Call-Frame Handlers (User-Declared) • 8–14	
8.4.3.4	Final and Last-Chance Handlers • 8–14	

8.4.3.5 Catchall Handler • 8–15
8.4.4 Exception-Related Built-in Symbols ____ 8–15

8.5 DEBUGGING EXIT HANDLERS 8–16

8.6 DEBUGGING AST-DRIVEN PROGRAMS 8–16
8.6.1 Disabling and Enabling the Delivery of ASTs ____ 8–17
8.6.2 Call Frames Associated with ASTs in SHOW CALLS Display ____ 8–17

PART II DEBUGGER COMMAND DICTIONARY

1 GENERAL COMMAND FORMAT CD–3

2 RULES FOR ENTERING AND TERMINATING COMMANDS CD–4
2.1 Interactively at the Terminal ____ CD–4
2.2 Within a Debugger Command Procedure ____ CD–5

3 COMMANDS RECOGNIZED ONLY ON VAXSTATIONS CD–5

4 OBSOLETE COMMANDS CD–5

5 DEBUGGER COMMAND DICTIONARY CD–6
@ (EXECUTE PROCEDURE) CD–7
ATTACH CD–9
CALL CD–10
CANCEL ALL CD–13
CANCEL BREAK CD–14
CANCEL DISPLAY CD–16
CANCEL IMAGE CD–17
CANCEL MODE CD–18
CANCEL MODULE CD–19
CANCEL RADIX CD–21
CANCEL SCOPE CD–22
CANCEL SOURCE CD–23
CANCEL TRACE CD–25
CANCEL TYPE/OVERRIDE CD–27
CANCEL WATCH CD–28
CANCEL WINDOW CD–29
CTRL/C, CTRL/W, CTRL/Y, CTRL/Z CD–30

DECLARE	CD–32
DEFINE	CD–35
DEFINE/KEY	CD–37
DELETE	CD–40
DELETE/KEY	CD–42
DEPOSIT	CD–44
DISABLE AST	CD–50
DISPLAY	CD–51
EDIT	CD–55
ENABLE AST	CD–57
EVALUATE	CD–58
EVALUATE/ADDRESS	CD–60
EXAMINE	CD–62
EXIT	CD–69
EXITLOOP	CD–70
EXPAND	CD–71
EXTRACT	CD–73
FOR	CD–75
GO	CD–77
HELP	CD–79
IF	CD–81
MOVE	CD–82
QUIT	CD–84
REPEAT	CD–85
SAVE	CD–86
SCROLL	CD–87
SEARCH	CD–89
SELECT	CD–92
SET ATSIGN	CD–95
SET BREAK	CD–96
SET DEFINE	CD–102
SET DISPLAY	CD–103
SET EDITOR	CD–107
SET EVENT_FACILITY	CD–109
SET IMAGE	CD–110
SET KEY	CD–112
SET LANGUAGE	CD–113
SET LOG	CD–115
SET MARGINS	CD–116
SET MAX_SOURCE_FILES	CD–119
SET MODE	CD–120
SET MODULE	CD–123
SET OUTPUT	CD–126
SET PROMPT	CD–128
SET RADIX	CD–129

SET SCOPE	CD–131
SET SEARCH	CD–134
SET SOURCE	CD–136
SET STEP	CD–139
SET TASK	CD–142
SET TERMINAL	CD–145
SET TRACE	CD–147
SET TYPE	CD–153
SET WATCH	CD–156
SET WINDOW	CD–161
SHOW AST	CD–163
SHOW ATSIGN	CD–164
SHOW BREAK	CD–165
SHOW CALLS	CD–166
SHOW DEFINE	CD–168
SHOW DISPLAY	CD–169
SHOW EDITOR	CD–170
SHOW EVENT_FACILITY	CD–171
SHOW EXIT_HANDLERS	CD–172
SHOW IMAGE	CD–173
SHOW KEY	CD–174
SHOW LANGUAGE	CD–176
SHOW LOG	CD–177
SHOW MARGINS	CD–178
SHOW MAX_SOURCE_FILES	CD–179
SHOW MODE	CD–180
SHOW MODULE	CD–181
SHOW OUTPUT	CD–184
SHOW RADIX	CD–185
SHOW SCOPE	CD–186
SHOW SEARCH	CD–188
SHOW SELECT	CD–189
SHOW SOURCE	CD–191
SHOW STACK	CD–193
SHOW STEP	CD–194
SHOW SYMBOL	CD–195
SHOW TASK	CD–198
SHOW TERMINAL	CD–201
SHOW TRACE	CD–202
SHOW TYPE	CD–203
SHOW WATCH	CD–204
SHOW WINDOW	CD–205
SPAWN	CD–206
STEP	CD–208
SYMBOLIZE	CD–212

TYPE CD–214
WHILE CD–216

PART III APPENDIXES

APPENDIX A COMMAND DEFAULTS A–1

APPENDIX B PREDEFINED KEY FUNCTIONS B–1

B.1 DEFAULT, GOLD, AND BLUE FUNCTIONS B–1

B.2 KEY DEFINITIONS SPECIFIC TO LK201 KEYBOARDS B–3

B.3 KEYS THAT SCROLL, MOVE, EXPAND, AND CONTRACT DISPLAYS B–3

B.4 ONLINE KEYPAD KEY DIAGRAMS B–4

B.5 DEBUGGER KEY DEFINITIONS B–5

APPENDIX C SCREEN MODE REFERENCE INFORMATION C–1

C.1 DISPLAY KINDS C–1

C.2 DISPLAY ATTRIBUTES C–2

C.3 PREDEFINED DISPLAYS C–3
C.3.1 SRC (Source Display) C–4
C.3.2 OUT (Output Display) C–4
C.3.3 PROMPT (Prompt Display) C–4
C.3.4 INST (Instruction Display) C–5
C.3.5 REG (Register Display) C–5

C.4 SCREEN-RELATED BUILT-IN SYMBOLS C–6

C.4.1 Screen Height and Width C–6
C.4.2 Pseudo-Display Names C–6

C.5 SCREEN DIMENSIONS AND PREDEFINED WINDOWS C–7

APPENDIX D BUILT-IN SYMBOLS AND LOGICAL NAMES D–1

D.1 SS$_DEBUG CONDITION D–1

D.2 LOGICAL NAMES D–1

D.3 BUILT-IN SYMBOLS D–2
D.3.1 Specifying the VAX Registers D–2
D.3.2 Constructing Identifiers D–3
D.3.3 Counting Parameters Passed to Command Procedures D–3
D.3.4 Controlling Radix D–4
D.3.5 Specifying Program Locations and the Current Value of an Entity D–4
D.3.6 Using Symbols and Operators in Address Expressions D–5
D.3.7 Obtaining Information About Exceptions D–8
D.3.8 Specifying Ada Tasks D–9

APPENDIX E SUMMARY OF DEBUGGER SUPPORT FOR LANGUAGES E–1

E.1 DEBUGGER SUPPORT FOR LANGUAGE ADA E–1
E.1.1 Operators in Language Expressions E–1
E.1.2 Constructs in Language and Address Expressions E–2
E.1.3 Data Types E–2
E.1.4 Predefined Attributes E–3
E.1.5 Tasking States E–4
E.1.5.1 Task States • E–4
E.1.5.2 Task Substates • E–4
E.1.6 Events E–5

E.2 DEBUGGER SUPPORT FOR LANGUAGE BASIC E–6
E.2.1 Operators in Language Expressions E–7
E.2.2 Constructs in Language and Address Expressions E–7
E.2.3 Data Types E–7

E.3	DEBUGGER SUPPORT FOR BLISS	E–8
E.3.1	Operators in Language Expressions	E–8
E.3.2	Constructs in Language and Address Expressions	E–9
E.3.3	Data Types	E–9
E.4	DEBUGGER SUPPORT FOR LANGUAGE C	E–10
E.4.1	Operators in Language Expressions	E–10
E.4.2	Constructs in Language and Address Expressions	E–11
E.4.3	Data Types	E–11
E.5	DEBUGGER SUPPORT FOR LANGUAGE COBOL	E–12
E.5.1	Operators in Language Expressions	E–12
E.5.2	Constructs in Language and Address Expressions	E–13
E.5.3	COBOL Data Types	E–13
E.6	DEBUGGER SUPPORT FOR LANGUAGE DIBOL	E–14
E.6.1	Operators in Language Expressions	E–14
E.6.2	Constructs in Language and Address Expressions	E–14
E.6.3	Data Types	E–14
E.7	DEBUGGER SUPPORT FOR LANGUAGE FORTRAN	E–15
E.7.1	Operators in Language Expressions	E–15
E.7.2	Constructs in Language and Address Expressions	E–16
E.7.3	Predefined Symbols	E–16
E.7.4	Data Types	E–16
E.8	DEBUGGER SUPPORT FOR LANGUAGE MACRO	E–17
E.8.1	Operators in Language Expressions	E–17
E.8.2	Constructs in Language and Address Expressions	E–18
E.8.3	Data Types	E–19
E.9	DEBUGGER SUPPORT FOR LANGUAGE PASCAL	E–19
E.9.1	Operators in Language Expressions	E–19
E.9.2	Constructs in Language and Address Expressions	E–20
E.9.3	Predefined Symbols	E–20
E.9.4	Built-In Functions	E–21
E.9.5	Data Types	E–21
E.10	DEBUGGER SUPPORT FOR LANGUAGE PL/I	E–22
E.10.1	Operators in Language Expressions	E–22
E.10.2	Constructs in Language and Address Expressions	E–22

E.10.3 Data Types E–23

E.11 DEBUGGER SUPPORT FOR LANGUAGE RPG E–23
E.11.1 Operators in Language Expressions E–24
E.11.2 Constructs in Language and Address Expressions E–24
E.11.3 Data Types E–24

E.12 DEBUGGER SUPPORT FOR LANGUAGE SCAN E–25
E.12.1 Operators in Language Expressions E–25
E.12.2 Constructs in Language and Address Expressions E–25
E.12.3 Data Types E–26
E.12.4 Events E–27

E.13 DEBUGGER SUPPORT FOR LANGUAGE UNKNOWN E–27
E.13.1 Operators in Language Expressions E–27
E.13.2 Constructs in Language and Address Expressions E–28
E.13.3 Data Types E–28

INDEX

FIGURES

1–1 Keypad Key Functions Predefined by the Debugger 1–8
1–2 Default Screen Mode Display Configuration 1–10
6–1 Default Screen Mode Display Configuration 6–2
6–2 Screen Mode Source Display When Source Code Is Not Available 6–5
6–3 Screen Mode Instruction Display 6–6
6–4 Screen Mode Register Display 6–7
B–1 Keypad Key Functions Predefined by the Debugger B–2

TABLES

2–1 Controlling Debugger Activation with the LINK and RUN Commands 2–3
4–1 Compiler Options for DST Symbol Information 4–3
4–2 Effect of Compiler and Linker on DST and GST Symbol Information 4–4
B–1 Key Definitions Specific to LK201 Keyboards B–3
B–2 Keys That Change the Key State B–4
B–3 Keys That Invoke Online Help to Display Keypad Diagrams B–5
B–4 Debugger Key Definitions B–5

Preface

Intended Audience

This manual is for programmers at all levels of experience.

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

The debugger can be used with most 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, refer to the documentation furnished with that language.

Note that you can use the 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 *VMS Delta/XDelta Utility Manual*, which describes the VMS DELTA/XDELTA Utility.

Document Structure

This manual is organized in three parts:

- Part I, Using the VMS Debugger (Chapters 1 through 8), presents task-oriented and conceptual information about the debugger. Debugger commands and features, such as keypad-key definitions, screen-mode displays, and built-in symbols, are discussed in the context of their use. To simplify the discussions, many details about command syntax, qualifiers, and so on, are not included in these chapters. Refer to Parts II and III for such additional information.
- Part II 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 consists of appendixes. These provide reference information on 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 is available in the documentation furnished with that language.

Information on the linking of programs and on shareable images is available in the *VMS Linker Utility Manual*.

Conventions

Convention	Meaning
RET	In examples, a key name (usually abbreviated) shown within a box indicates that you press a key on the keyboard; in text, a key name is not enclosed in a box. In this example, the key is the RETURN key. (Note that the RETURN key is not usually shown in syntax statements or in all examples; however, assume that you must press the RETURN key after entering a command or responding to a prompt.)
CTRL/C	A key combination, shown in uppercase with a slash separating two key names, indicates that you hold down the first key while you press the second key. For example, the key combination CTRL/C indicates that you hold down the key labeled CTRL while you press the key labeled C. In examples, a key combination is enclosed in a box.
$ SHOW TIME 05-JUN-1988 11:55:22	In examples, system output (what the system displays) is shown in black. User input (what you enter) is shown in red.
$ TYPE MYFILE.DAT . . .	In examples, a vertical series of periods, or ellipsis, means either that not all the data that the system would display in response to a command is shown or that not all the data a user would enter is shown.
input-file, . . .	In command syntax or examples, a horizontal ellipsis indicates that additional parameters, values, or other information can be entered, that preceding items can be repeated one or more times, or that optional arguments in a statement have been omitted.
[logical-name]	Brackets indicate that the enclosed item is optional. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
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.

New and Changed Features

The following technical changes have been made to the VMS Debugger since VAX/VMS Version 4.4. See the command dictionary and any referenced chapter or appendix for complete details.

The following enhancements have been made to MACRO support, to match the support available with other languages.

- Source display is now available (in both line mode and screen mode).
- When the language is set to MACRO, the default behavior of the STEP command is STEP/LINE (not STEP/INSTRUCTION, the previous behavior).
- When the language is set to MACRO, the debugger interprets an address expression used in a language expression as the current *value* stored at that address (not the *address* denoted by the address expression, the previous behavior). See Chapter 3.

The SET WATCH command now permits you to set watchpoints on nonstatic variables (variables allocated on the stack or in registers). The new /INTO, /OVER, and /[NO]STATIC qualifiers are used with nonstatic watchpoints (see Chapter 2).

The STEP and CALL commands may now be entered at exception breakpoints (see Chapter 8).

The new /CALLS qualifier of the SET MODULE command sets all modules that currently have routines on the call stack.

When examining instructions, you can now obtain the address and contents of instruction operands. You can either use the command EXAMINE/OPERANDS or first enter the command SET MODE OPERANDS to establish the default behavior of the EXAMINE command for instructions.

The SPAWN command has been enhanced. The new /INPUT qualifier specifies an input DCL command file to be executed in a subprocess. The new /OUTPUT qualifier specifies an output file to capture the output from a SPAWN operation.

The following enhancements have been made to screen mode (see Appendix C):

- Register displays are now dynamic by default. The register information is automatically reformatted when you resize the window, and the window is automatically resized and reformatted when you change the screen height or width.
- The predefined windows now include eighths of the screen height. These built-in names have the prefix E. For example, window RE123 occupies the top three eighths of the right half of the screen.
- Source display is now available for MACRO programs.

If your program includes any VAX Ada® code, two predefined breakpoints are automatically set at debugger startup. These breakpoints are associated with Ada exception conditions (see Chapter 8 and the VAX Ada documentation).

® Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

The following enhancements have been made to the support for workstations. The new SET MODE [NO]SEPARATE command provides an optional separate window to display debugger input and output. The new /[NO]POP qualifier of the SET PROMPT command controls whether the debugger window is automatically popped over other windows when the debugger prompts for commands.

Part I Using the VMS Debugger

This part contains task oriented and conceptual information about the debugger, organized into eight chapters.

1 Introduction to the VMS Debugger

This chapter is a tutorial introduction to the VMS Debugger (debugger). The following information is provided:

- An overview of the debugger's features (Section 1.1)
- Enough information to get you started, including a sample debugging session (Section 1.2)
- A list of the debugger commands, by function (Section 1.3)

Once you have read this chapter, consult the rest of this manual for additional details.

1.1 Overview of the Debugger

The debugger is a tool that helps you locate run-time programming or logic errors, also known as bugs. You use the debugger with a program that has been compiled and linked successfully but that does not run correctly. For example, the program may give incorrect output, go into an infinite loop, or terminate prematurely.

You locate errors with the debugger by observing and manipulating your program interactively as it executes. By entering debugger commands at the terminal, you can do the following:

- Control the program's execution (start the program, stop at points of interest, resume execution, and so on)
- Trace the execution path of the program
- Monitor changes in variables and other program entities
- Monitor events (for example, exception conditions)
- 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. 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 can also tailor the debugger for your own needs. The next section summarizes the debugger features.

Introduction to the VMS Debugger

1.1 Overview of the Debugger

1.1.1 Functional Features

Programming Language Support

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

Symbolic Debugging

The 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 need 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, by using a type or radix qualifier with the EXAMINE command, you can display the contents of a program location in ASCII, word-integer, or hexadecimal format.

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 enables you to 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 kinds of events, such as exceptions and Ada tasking events.

® Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

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 location 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 enables you to 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 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 and Workstation Support

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

1.1.2 Convenience Features

Online HELP

Online HELP is always available during a debugging session. Online HELP contains information on all debugger commands and selected topics.

Source Code Display

You can display lines of source code for all supported languages during a debugging session.

Introduction to the VMS Debugger

1.1 Overview of the Debugger

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 (if you have a VT52, VT100, or LK201 keyboard). 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 are 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 lengthy commands, address expressions, or values in abbreviated form.

1.2 Getting Started with the Debugger

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 following basic functions that apply to most situations.

- Compiling and linking your program to prepare for debugging
- 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.

Several examples are language specific. However, the general concepts are readily adaptable to all supported languages.

1.2.1 Compiling and Linking to Prepare for Debugging

Before you can use the debugger, you must compile and link the modules (compilation units) of your program as explained in this section. The following example shows how to compile and link a FORTRAN program (consisting of a single compilation unit named FORMS) before using the debugger.

Note: The /DEBUG and /NOOPTIMIZE qualifiers may be compiler command defaults for some languages. These qualifiers are used in the example for emphasis.

```
$ FORTRAN/DEBUG/NOOPTIMIZE FORMS
$ LINK/DEBUG FORMS
$
```

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

Some compilers optimize the object code to reduce the size of the program or make it run faster. In such cases you should compile your program with the /NOOPTIMIZE (or equivalent) command qualifier. Otherwise the contents of some program locations may be inconsistent with what you might expect from viewing the source code.

The /DEBUG qualifier on the LINK command causes the linker to include all symbol information that is contained in FORMS.OBJ in the executable image. The qualifier also causes the 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.)

1.2.2 Starting and Terminating a Debugging Session

After you have compiled and linked your program with the /DEBUG command qualifier, as explained in Section 1.2.1, you can then invoke the debugger by entering the DCL command RUN. The following example shows how the debugger identifies itself after you invoke it:

```
$ RUN FORMS

                    VAX DEBUG Version 5.0

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

The "INITIAL" message indicates that the debugging session is initialized for a FORTRAN program and that the name of the main program unit (the module containing the image transfer address) is FORMS. The initialization sets up any language-dependent debugger parameters.

At this point, execution is suspended at the start of the main program. The DBG> prompt, which is displayed whenever the debugger suspends execution, indicates that you can now enter debugger commands. (Section 1.2.3 explains how to enter commands, and Section 1.2.5 explains how to start and stop execution.)

To terminate a debugging session and return to DCL level any time the DBG> prompt is displayed, type EXIT or press CTRL/Z:

```
DBG> EXIT
$
```

To interrupt a debugging session and return to DCL level, press CTRL/Y. This may be necessary if your program loops or if you want to interrupt a debugger command that is still in progress.

To resume the debugging session after a CTRL/Y interruption, type either the CONTINUE or DEBUG command at DCL level. Use the CONTINUE command to return to the exact point at which you interrupted the debugging session. If you interrupted the session because of an infinite loop, use the DEBUG command instead. The DEBUG command returns 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>
```

1.2.3 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 command lines, and the left arrow and right arrow keys to position the cursor for editing the command line. See Sections 1 and 2 of the command dictionary for complete rules on entering debugger commands.

Entering the HELP command gives online HELP on debugger commands and selected topics. For example, if you enter the command 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.

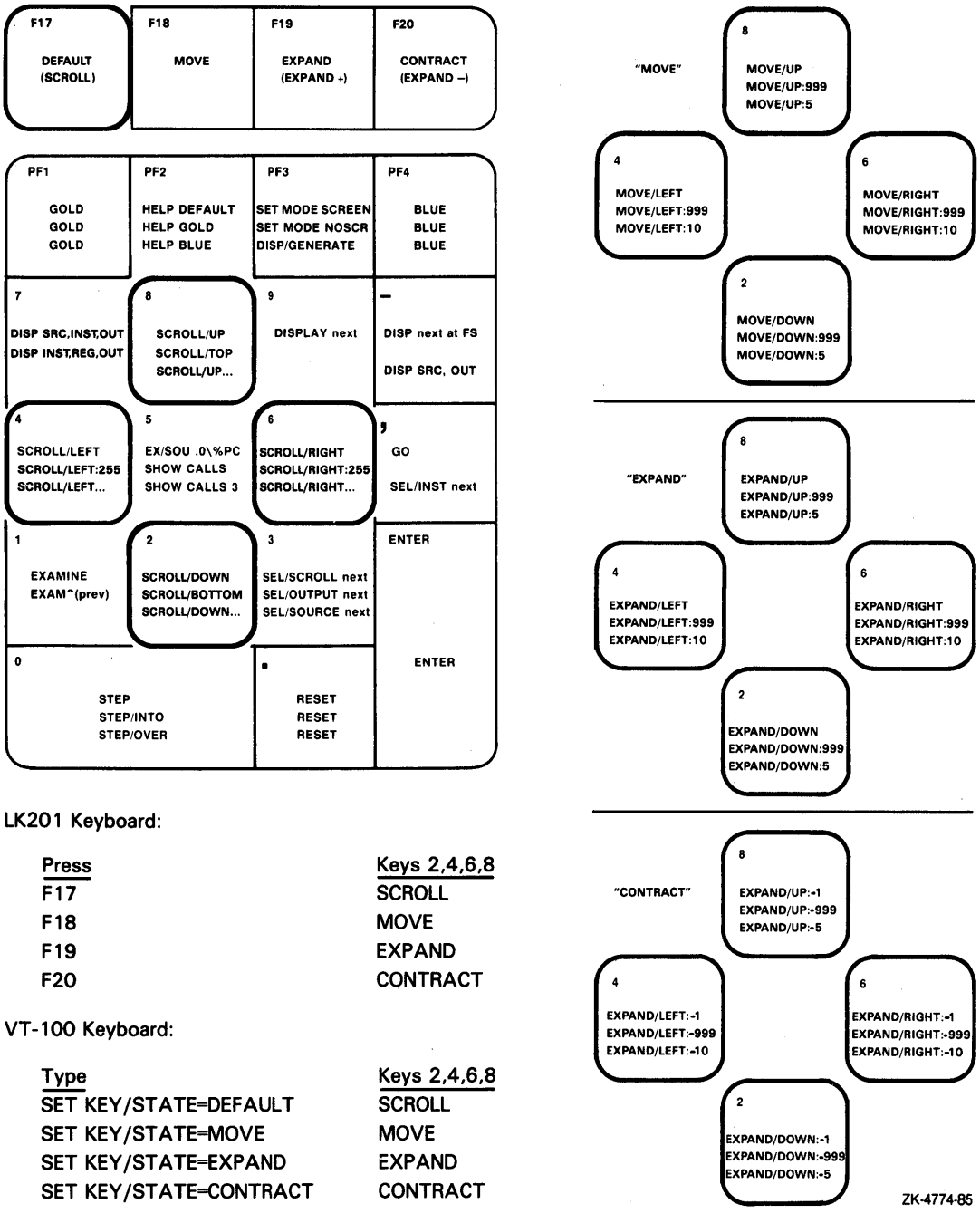
Most keypad keys have three predefined functions—DEFAULT, GOLD, and BLUE. To obtain a key's DEFAULT function, press the key. To obtain its GOLD function, first press the PF1 key, then the given key. To obtain its BLUE function, first press the PF4 key, then the given key. In Figure 1–1, the DEFAULT,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

Normally, keys 2, 4, 6, and 8 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. Enter the command HELP KEYPAD to get HELP on the keypad key definitions.

1.2.4 Viewing Your Source Code

The debugger provides two modes for displaying information: noscreen mode and screen mode. By default, when you invoke the debugger, you are in noscreen mode, but you may find that it is easier to view source code in screen mode. The following sections briefly describe both modes.

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

1.2.4.1 Noscreen Mode

Noscreen mode is the default, line-oriented mode of displaying input and output. The interactive examples throughout this chapter, excluding Section 1.2.4.2, illustrate noscreen mode.

In noscreen mode, use the TYPE command to display one or more source lines. For example, the following command displays line 7 of the module where execution is currently suspended:

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

The display of source lines is independent of program execution. To display source code from a module other than the one where execution is currently suspended, use the TYPE command with 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 Section 1.2.5.1.2.

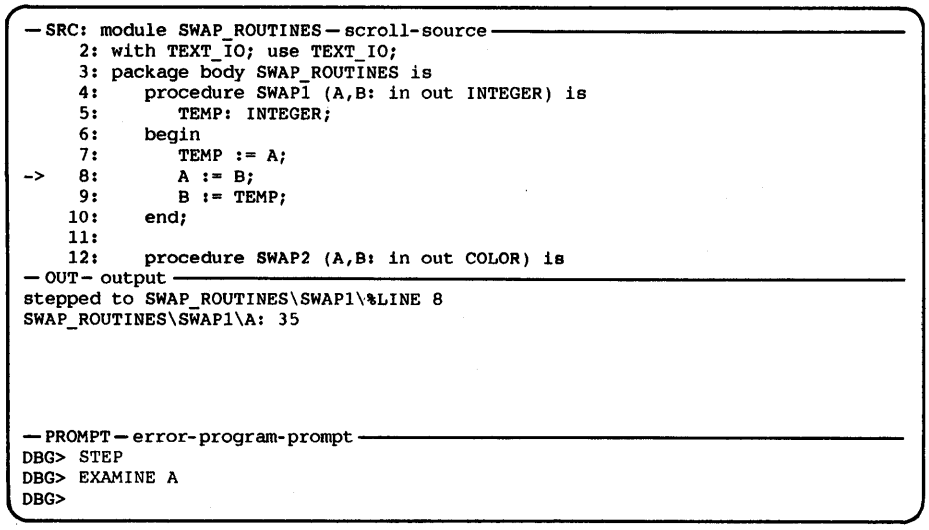
You can also use the EXAMINE/SOURCE command to display the source line for a routine or any other program location that is associated with an instruction.

To invoke noscreen mode from screen mode, press the keypad key sequence GOLD-PF3 (or type SET MODE NOSCREEN). Note that you can use the TYPE and EXAMINE/SOURCE commands in screen mode as well as noscreen mode.

1.2.4.2 Screen Mode

Screen mode provides the easiest way to view your source code. To invoke screen mode, press keypad key PF3 (or type SET MODE SCREEN). In screen mode, by default the debugger splits the screen into three displays named SRC, OUT, and PROMPT, as illustrated in Figure 1–2.

Figure 1–2 Default Screen Mode Display Configuration



ZK-6502-HC

The SRC display shows the source code of the module (compilation unit) where execution is currently suspended. An arrow in the left column points to the source line corresponding to the current value of the program counter, PC (the PC is a VAX register that contains the address of the next instruction to be executed). 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 (by default, SRC is 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 routine where execution is currently suspended, it tries to display source lines for the next routine 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. For example:

- Execution is currently suspended within a routine for which no source code is available (for example, a system or shareable image routine).
- Execution is currently suspended within a routine that was compiled without the /DEBUG compiler command qualifier (or with /NODEBUG).
- Execution is currently suspended within a routine whose module is not "set" (module setting is explained in Section 1.2.7.1).

- The source file was moved to a different directory after it was compiled (the location of source files is embedded in the object modules).

1.2.5 Controlling and Monitoring Program Execution

This section covers the following topics:

- Starting and resuming program execution with the GO and STEP commands
- Determining where execution is currently suspended 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 can pick program locations where you can then test and manipulate the contents of variables as described in Section 1.2.6.

1.2.5.1 Starting and Resuming Program Execution

There are two basic commands for starting and resuming program execution: GO and STEP.

1.2.5.1.1 The GO Command

The GO command starts execution, which continues until forced to stop. Typical uses of the GO command are illustrated in this section.

With most programming languages, when you invoke the debugger, execution is initially suspended directly at the start of the main program. Entering a GO command at this point quickly enables you to test for an exception condition or an infinite loop.

If an exception condition that is not handled by your program occurs, the debugger interrupts execution at that point so that you can enter commands. If you are using screen mode, the pointer in the source display indicates where execution stopped. You can also use the SHOW CALLS command (explained in Section 1.2.5.2) to identify the currently active routine calls (the call stack).

If an infinite loop occurs, the program does not terminate, so the debugger prompt does not reappear. To obtain the prompt, interrupt the program with CTRL/Y and then enter the DCL DEBUG command. You can then look at the source display and a SHOW CALLS display to find where execution is suspended.

In general, the most common use of the GO command is in conjunction with breakpoints, tracepoints, and watchpoints, as described in Sections 1.2.5.3, 1.2.5.4, and 1.2.5.5, respectively. If you set a breakpoint in the path of execution and then enter the GO command, execution is suspended at that breakpoint. Similarly, if you set a tracepoint, execution is monitored through that tracepoint. And if you set a watchpoint, execution is suspended when the value of the "watched" variable changes.

1.2.5.1.2 The STEP Command

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>
```

Execution is now suspended 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. In general, you need to use path names in commands only if the debugger cannot resolve a symbol ambiguity in your program (see Section 1.2.7). Usually the debugger can figure out the symbol you mean from its context.

When using the STEP command, you can also specify a number of lines for the STEP command to execute. In the following example, the STEP command executes three lines:

```
DBG> STEP 3
stepped to TEST\COUNT\%LINE 34
     34:    SWAP(X,Y);
DBG>
```

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. Also, if a line has more than one statement on it, the debugger executes all the statements on that line as part of the single step.

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 entering the SET STEP INTO command, you tell the debugger to suspend execution within called routines as well as within the routine where execution is currently suspended (SET STEP OVER is the default mode).

1.2.5.2 Determining Where Execution Is Currently Suspended

The SHOW CALLS command is useful when you are unsure where execution is suspended during a debugging session (for example, after returning to the debugger following a CTRL/Y interrupt).

The command shows a traceback that lists the sequence of active calls leading to the routine where execution is suspended. 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>
```

This example indicates that execution is suspended 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).

For each routine (beginning with the one where execution is suspended), the debugger displays the following information:

- Name of the module that contains the routine
- Name of the routine
- Line number at which the call was made (or at which execution is suspended, in the case of the current routine)
- Corresponding PC values (the relative PC address from the start of the routine and the absolute PC address of the program)

1.2.5.3 Suspending Program Execution

The SET BREAK command enables you to select locations at which to suspend program execution (breakpoints). You can then enter commands to check the call stack, examine the current values of variables, and so on. You resume execution from a breakpoint with the GO or STEP commands.

The following example shows a typical use of the SET BREAK command:

```
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 use the STEP command to step through routine COUNT and then use the EXAMINE command (discussed in Section 1.2.6.1) 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 interprets the line number as a memory location. For example, the next command sets a breakpoint at line 41 of the module where execution is currently suspended. The breakpoint causes the debugger to suspend further execution when the PC value is at the start of line 41.

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

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 where execution is suspended, you must specify the module's name in a path name. For example:

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

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
DBG>
```

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 command's DO clause displays the value of the variable TEMP whenever the breakpoint is triggered, as shown in the following example:

```
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, enter the command SHOW BREAK:

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

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

1.2.5.4 Tracing Program Execution

The SET TRACE command enables you to 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 path of execution, 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 routine where execution is suspended. 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
DBG>
```

The /SILENT qualifier suppresses the trace message and source code display. This is useful when you want to use the SET TRACE command 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
    .
    .
    .
```

1.2.5.5 Monitoring Changes in Variables

The SET WATCH command enables you to specify program variables that the debugger monitors 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
DBG>
```

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 at SCREEN_IO\%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.

This technique for setting watchpoints always applies to *static* variables. A static variable is associated with the same virtual memory location throughout program execution.

A variable that is allocated on the stack or in a register (a *nonstatic* variable) exists only when its defining routine is active (on the call stack). If you try to set a watchpoint on a nonstatic variable when its defining routine is not active, the debugger issues a warning:

```
DBG> SET WATCH Y
%DEBUG-W-SYMNOTACT, nonstatic variable 'Y' is not active
DBG>
```

A convenient technique for setting a watchpoint on a nonstatic variable is to set a tracepoint on the defining routine, also specifying a DO clause to set the watchpoint whenever execution reaches the tracepoint. In the following example, a watchpoint is set on the nonstatic variable Y in routine ROUT3. After the tracepoint is triggered, the WPTTRACE message indicates that the nonstatic watchpoint is set. And the watchpoint is triggered when the value of Y changes:

```
DBG> SET TRACE/NOSOURCE ROUT3 DO (SET WATCH Y)
DBG> GO
   .
   .
   .
trace at routine MOD4\ROUT3
%DEBUG-I-WPTTRACE, nonstatic watchpoint, tracing every instruction
   .
   .
   .
watch of MOD4\ROUT3\Y at MOD4\ROUT3\%LINE 16
   16:     Y := 4
   old value:    3
   new value:    4
break at MOD4\ROUT3\%LINE 17
   17:     SWAP(X,Y);
DBG>
```

The debugger monitors nonstatic watchpoints by tracing every instruction. Because this slows execution speed compared to monitoring static watchpoints, the debugger informs you (with the WPTTRACE message) when it is monitoring nonstatic watchpoints.

When execution returns to the calling routine, the nonstatic variable is no longer active, so the debugger automatically cancels the watchpoint and issues a message to that effect.

1.2.6 Examining and Manipulating Program Data

This section explains how to use the EXAMINE, DEPOSIT, and EVALUATE commands to display and modify the contents of variables and evaluate expressions. Note that before you can examine or deposit into a nonstatic variable (as defined in Section 1.2.5.5), its defining routine must be active (on the call stack).

1.2.6.1 Displaying the Value of a Variable

To display the current value of a variable, use the EXAMINE command. It has the following form:

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 show some uses of 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.

1.2.6.2 Changing the Value of a Variable

To change the value of a variable, use the DEPOSIT command. It has the following form:

```
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 quotation marks (") or apostrophes ('')):

```
DBG> DEPOSIT PART_NUMBER = "WG-7619.3-84"
DBG>
```

Deposit an integer expression:

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

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'
DBG>
```

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

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

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
DBG>
```

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.

1.2.6.3 Evaluating Expressions

To evaluate a language expression, use the EVALUATE command. It has the following form:

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>
```

1.2.7 Controlling Symbol References

In most cases, the way in which the debugger handles the symbols (variable names, and so on) that you reference in debugger commands is transparent to you. However, the following two areas might require action:

- Module setting
- Multiply-defined symbols

1.2.7.1 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 symbol records of a given module are loaded into the RST at one time.

At debugger startup, only the module containing the image transfer address is set. As your program executes, whenever the debugger interrupts execution it sets the module where execution is suspended. This enables you to reference the symbols that should be visible at the current PC value.

If you try to reference a symbol in a module that has not been set, the debugger warns 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 explicitly:

```
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 entering the command SET MODE NODYNAMIC (SET MODE DYNAMIC enables dynamic module setting).

1.2.7.2 Resolving Multiply-Defined Symbols

The debugger finds the symbols that you reference in commands according to the following conventions. First, it looks in the *PC scope* (also known as scope 0), according to the scope and visibility rules of the currently set language. This means that, typically, the debugger first looks within the block or routine surrounding the current PC value (where execution is currently suspended). If the symbol is not found, the debugger searches the nesting program unit, then its nesting unit, and so on. The precise manner, which depends on the language, guarantees that the correct declaration of a multiply-defined symbol is selected.

The debugger must enable you to reference symbols throughout your program, not just those that are visible in the PC scope as defined by the language. This is necessary so you can set breakpoints in arbitrary areas or examine arbitrary variables, and so on. Therefore, if the symbol is not visible in the PC scope, the debugger continues searching. After the PC scope, the debugger searches the scope of the calling routine (if any), then its caller, and so on. Symbolically, the complete scope search list is denoted *0,1,2, . . . ,n*, where scope 0 is the PC scope and *n* is the number of calls in the call stack. Within each scope, the debugger uses the visibility rules of the language to locate a symbol. If the symbol is not found, the debugger searches the rest of the run-time symbol table.

If the debugger cannot resolve a symbol ambiguity, it issues a message. For example:

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

You can then 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 specify the declaration of Y that is visible in the new scope. 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.

1.2.8 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, you can refer 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 5.0

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

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

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

To check the current values of variables *N* and *K*, enter the following 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 the variables *I* and *K* have the expected values, enter the following command:

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

The variable *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, correct the program by entering 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 appear each time line 11 is executed. The effect of the SET TRACE command has been to patch the program to perform correctly.

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>
```

Now run the program to test the patch. Enter the command GO to execute the program until it reaches 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 works properly with the DEPOSIT K = K + 1 patch. To correct the source code without leaving the debugging session, use the EDIT command. It invokes 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 is as 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 runs correctly:

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

To set a breakpoint at line 12 that displays the values of *I* and *K* automatically, enter the following SET BREAK 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
DBG>
```

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

1.3 Debugger Command Summary

The following sections list all the debugger commands and any related DCL commands in functional groupings, along with brief descriptions. During a debugging session, you can get online HELP on all debugger commands and their qualifiers by typing HELP.

1.3.1 Starting and Terminating a Debugging Session

The following commands are used to start, interrupt, and terminate 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	Has the same effect as CTRL/Y, unless the program has a CTRL/C service routine
($) CONTINUE[1]	Resumes a debugging session after a CTRL/Y interruption
($) DEBUG[1]	Resumes a debugging session after a CTRL/Y interruption but returns you to the debugger prompt
ATTACH	Passes control of your terminal from the current process to another process (similar to the DCL command ATTACH)
SPAWN	Creates a subprocess, enabling you to execute DCL commands without terminating a debugging session or losing your debugging context (similar to the DCL command SPAWN)

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

1.3.2 Controlling and Monitoring Program Execution

The following commands are used to control and monitor 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
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

The following commands are used to examine and manipulate 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 Radix

The following commands are used to control type selection and radix:

(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

1.3.5 Controlling Symbol Lookup and Symbolization

The following commands are used to control symbol lookup and symbolization:

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 and shareable images are set automatically when the debugger interrupts execution
(SET,SHOW,CANCEL) SCOPE	(Establishes, displays, restores) the scope for symbol lookup

SYMBOLIZE	Converts a virtual address to a symbolic address
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

1.3.6 Displaying Source Code

The following commands are used to control the display of source code:

TYPE	Displays lines of source code
EXAMINE/SOURCE	Displays the source code at the location specified by the address expression
SEARCH	Searches the source code for the specified string
(SET,SHOW) SEARCH	(Establishes, displays) the default qualifiers for the SEARCH command
SET STEP SOURCE	Enables the display of source code after a STEP command has been executed or at a breakpoint, tracepoint, or watchpoint
(SET,SHOW) MARGINS	(Establishes, displays) the left and right margin settings for displaying source code
(SET,SHOW,CANCEL) SOURCE	(Creates, displays, cancels) a source directory search list
(SET,SHOW) MAX_SOURCE_FILES	(Establishes, displays) the maximum number of source files that may be kept open at one time

1.3.7 Screen Mode

The following commands are used to control screen mode and screen displays:

SET MODE [NO]SCREEN	Enables/disables screen mode
DISPLAY	Modifies an existing display
SCROLL	Scrolls a display
EXPAND	Expands or contracts a display
MOVE	Moves a display across the screen
(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

SAVE	Saves the current contents of a display into another display
EXTRACT	Saves a display or the current screen state into a file
(SET,SHOW) TERMINAL	(Establishes, displays) the terminal screen height and width that the debugger uses when it formats displays and other output
SET MODE [NO]SCROLL	Controls whether an output display is updated line by line or once per command
CTRL/W,DISPLAY/REFRESH	Refreshes the screen

1.3.8 Source Editing

The following commands are used to control source editing from a debugging session:

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

The following commands are used to define and delete symbols for addresses, commands, or values:

DEFINE	Defines a symbol as an address, command, or value
DELETE	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

The following commands are used to control keypad mode and key definitions:

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

1.3.11 Command Procedures, Log Files, and Initialization Files

The following commands are used with command procedures and log files:

@file-spec	Executes a command procedure
(SET,SHOW) ATSIGN	(Establishes, displays) the default file specification that the debugger uses to search for command procedures
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	Identifies the current output options established by the SET OUTPUT command

1.3.12 Control Structures

The following commands are used to establish conditional and looping structures for debugger commands:

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 Miscellaneous Commands

The following commands are used for miscellaneous purposes:

(DISABLE,ENABLE,SHOW) AST	(Disables, enables) the delivery of ASTs in the program, identifies whether delivery is enabled or disabled
(SET,SHOW) EVENT_FACILITY	(Establishes, identifies) the current run-time facility for language-specific events
(SET,SHOW) LANGUAGE	(Establishes, identifies) the current language
SET MODE [NO]SEPARATE	Controls whether the debugger, when used on a VAXstation, creates a separate window for debugger input and output

1.3 Debugger Command Summary

SET OUTPUT [NO]TERMINAL	Controls whether debugger output, except for diagnostic messages, is displayed or suppressed
SET PROMPT	Specifies the debugger prompt
(SET,SHOW) TASK	Modifies the tasking environment, displays task information
SHOW EXIT_HANDLERS	Identifies the exit handlers declared in the program
SHOW MODE	Identifies the current debugger modes established by the SET MODE command (for example, screen mode, step mode)
SHOW OUTPUT	Identifies the current output options established by the SET OUTPUT command

2 Starting and Controlling Program Execution

This chapter describes the options for invoking the debugger and for starting and controlling program execution while debugging.

2.1 Starting, Terminating, and Interrupting a Debugging Session

This section explains how to do the following:

- Compile and link your program so you can invoke the debugger
- Start, interrupt, resume, and terminate a debugging session

2.1.1 Invoking the Debugger with the DCL RUN Command

The usual way to invoke the debugger is as follows:

1. Compile your program using the /DEBUG and /NOOPTIMIZE (or equivalent) qualifiers with the DCL compiler command (consult your language documentation to determine the compiler command defaults).
2. Link your program using the /DEBUG qualifier with the DCL LINK command.
3. Execute your program using the DCL RUN command. The debugger initially takes control and prompts for commands. Note that you cannot run a program under debugger control over a DECnet link.

The following example illustrates these steps with a simple Pascal program, INVENTORY, that consists of two compilation units whose source code is in two separate files, FORMS.PAS and INVENTORY.PAS. INVENTORY is the main program unit.

```
$ PASCAL/DEBUG/NOOPTIMIZE FORMS, INVENTORY
$ LINK/DEBUG INVENTORY, FORMS
$ RUN INVENTORY

        VAX DEBUG Version *****

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

When the debugger first takes control, it does the following:

- Displays its banner.
- Sets the language-dependent parameters to the language of the main program (the module that contains the image transfer address). The "INITIAL" message identifies the language to which the debugging session is initialized and the name of the main program (Pascal and INVENTORY, respectively, in the previous example). See Sections 3.1.8 and 3.1.9 for more information about language-dependent parameters.
- Executes any user-supplied initialization file (see Section 7.2).

- Suspends execution at the start of the main program. The DBG> prompt, which is displayed whenever the debugger suspends execution, indicates that you can now enter debugger commands.

In some cases the debugger suspends execution before the start of the main program and displays the following additional message:

```
%DEBUG-I-NOTATMAIN, type GO to get to start of main program
```

See Section 8.3 for an explanation of this message.

The effect of the qualifiers used with the compiler command (PASCAL, in this example) and the LINK command is as follows.

The /DEBUG qualifier on the compiler command loads the debugger symbol records associated with each compilation unit into its object module. These symbol records enable you to use, in debugger commands, the names of variables, routines, labels, and other symbols as they appear in the source code. By specifying options with the /DEBUG qualifier, you can control the level of symbolic information provided (see Section 4.1.1). This qualifier does not affect whether the debugger is invoked or how it is invoked.

Most compilers optimize code to reduce the size of the program and make it run faster. For example, invariant expressions are removed from DO loops so that they are evaluated only once at run time; also, some memory locations may be allocated to different variables at different points in the program. The /NOOPTIMIZE (or equivalent) qualifier ensures that the code is not optimized and, therefore, that the contents of all program locations are consistent with what you would expect from looking at the source code. Section 8.1 describes some of the effects of optimization.

Note also another possible cause of unexpected behavior. The debugger and your program share the same address space. In some rare cases, this may cause the debugger to affect how your program executes. Section 2.6 explains how the debugger controls execution and the possible sources of interference.

The /DEBUG qualifier on the LINK command does the following:

- Copies the debugger symbol records from the object modules being linked into the debug symbol table (DST) and puts the DST in the executable image.
- Directs the image activator to pass control to the debugger when you subsequently execute the image with the RUN command.

See Section 4.1.3 for more details on how the LINK command controls symbol information.

Even if you have compiled and linked an image with the /DEBUG command qualifier, you can execute that image normally, without it being under debugger control. To do so, use the /NODEBUG qualifier on the DCL RUN command. For example:

```
$ RUN/NODEBUG INVENTORY
```

This is convenient for checking your program once you think it is error free. But the data required by the debugger still occupies space within the executable image. So, when you think your program is correct, you may want to link your program again without the /DEBUG qualifier. This creates an image with only traceback data in the DST, to use less disk space.

Table 2–1 summarizes how to control debugger activation by means of LINK and RUN command qualifiers. Note that the LINK command qualifiers /[NO]DEBUG and /[NO]TRACEBACK affect not only debugger activation but also the level of symbolic information provided.

Table 2–1 Controlling Debugger Activation with the LINK and RUN Commands

LINK Command Qualifier	To Run Program With Debugger	To Run Program Without Debugger	Maximum Symbolic Information Available[1]
/DEBUG	RUN	RUN/NODEBUG	Full
/TRACEBACK or /NODEBUG[2]	RUN/DEBUG	RUN	Only traceback[3]
/NOTRACEBACK	Cannot	RUN	None

[1]The level of symbolic information available while debugging is controlled both by the compile command qualifier and the LINK command qualifier (see Section 4.1).

[2]LINK/TRACEBACK (or LINK/NODEBUG) is a LINK command default.

[3]Traceback information includes compiler-generated line numbers and the names of routines and modules (compilation units). This symbolic information is used by the VMS traceback condition handler to identify the PC value and the active calls when a run-time error has occurred. The information is also used by the debugger SHOW CALLS command (see Section 1.2.5.2).

2.1.2 Invoking the Debugger with the DCL DEBUG Command

You can invoke the debugger while your program is executing freely—for example, if you suspect that the program may be looping or if you see erroneous output.

To invoke the debugger in this manner, perform the following steps:

1. Compile and link the program with the /DEBUG command qualifier, as described in the previous section (you can also use LINK/TRACEBACK, but only traceback symbols are then available while you debug).
2. Enter the DCL command RUN/NODEBUG to execute the program without debugger control.
3. To interrupt the executing program, press CTRL/Y. Control then passes to the DCL command interpreter.
4. Enter the DCL command DEBUG to activate the debugger. It displays its banner, sets the language-dependent parameters to the language of the module where execution was interrupted, executes any user-defined initialization file, and prompts for commands. Usually you will not know where execution was interrupted. Enter the SHOW CALLS command to identify the current PC value and the sequence of routine calls on the call stack (the SHOW CALLS command is described in Section 1.2.5.2).

For example:

```
$ PASCAL/DEBUG/NOOPTIMIZE FORMS,INVENTORY
$ LINK/DEBUG INVENTORY,FORMS
$ RUN/NODEBUG INVENTORY
    .
    .
    .
CTRL/Y
Interrupt
$ DEBUG

        VAX DEBUG Version *****

%DEBUG-I-INITIAL, language is PASCAL, module set to 'INVENTORY'
DBG> SHOW CALLS
    .
    .
    .
```

Interrupting a running program with CTRL/Y and then invoking the debugger with the DEBUG command is useful under the following conditions:

- Your program is in an infinite loop.
- After entering the RUN/NODEBUG command, you 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. In this case, traceback information is the only symbolic information available for debugging.

2.1.3 Terminating a Debugging Session

To terminate a debugging session in an orderly manner, use the EXIT or QUIT commands, or press CTRL/Z. These commands invoke the debugger exit handlers to close log files, restore the screen and keypad states, and so on.

The EXIT command and CTRL/Z have the same effect. The QUIT command is like the EXIT command or CTRL/Z, except that the EXIT command and CTRL/Z also execute any exit handlers your program may have declared; the QUIT command does not.

You can also terminate a debugging session by pressing CTRL/Y and then entering the DCL commands EXIT or STOP, or any DCL command that executes an image. CTRL/Y followed by EXIT is preferred because all exit handlers are executed. CTRL/Y followed by STOP does not execute the debugger exit handlers. Therefore, the screen and keypad may not get restored to their original state.

2.1.4 Interrupting and Resuming a Debugging Session

There are two basic ways of interrupting a debugging session:

- Press CTRL/Y.
- Create a subprocess or attach to an existing process or subprocess by using the debugger commands SPAWN and ATTACH, respectively.

With each of these techniques, depending on the circumstances, you have several options for resuming your debugging session.

2.1.4.1 Interrupting with CTRL/Y

Pressing CTRL/Y during a debugging session interrupts the session and passes control to the DCL command interpreter.

If, after pressing CTRL/Y, you enter the DCL command CONTINUE, the debugging session resumes at exactly the same point at which you interrupted it.

If, after pressing CTRL/Y, you enter the DCL command DEBUG, control returns to the debugger and the debugger prompt is displayed. You can then enter debugger commands. For example:

```
DBG> GO
    .
    .
    .
CTRL/Y
Interrupt
$ DEBUG
DBG>
```

The CTRL/Y–DEBUG command sequence is useful in the following circumstances. For example:

- Your program goes into an infinite loop when you enter a GO command during a debugging session. After returning to the debugger prompt, you can enter the SHOW CALLS command to determine where execution has been suspended.
- You want to interrupt a debugger command that is taking too long to complete or is generating a large amount of output. In this case, the CTRL/Y–DEBUG sequence causes the debugger to abort the current command when all of the data structures are in a consistent state (entering a second CTRL/Y-DEBUG sequence immediately aborts the command, regardless of whether all data structures are in a consistent state).

Pressing CTRL/C during a debugging session is like pressing CTRL/Y, unless your program has a CTRL/C AST service routine enabled. If a CTRL/C service routine exists, pressing CTRL/C passes control to that routine rather than the DCL command interpreter.

2.1.4.2 Interrupting with the SPAWN and ATTACH Commands

The debugger SPAWN and ATTACH commands enable you to interrupt a debugging session from the debugger prompt, enter DCL commands, and return to the debugger prompt. These commands function essentially like the DCL SPAWN and ATTACH commands.

Use the debugger SPAWN command to create a subprocess. Use the debugger ATTACH command to attach to an existing process or subprocess.

You can enter the SPAWN command with or without specifying a DCL command as parameter. If you specify a DCL command, it is executed in a subprocess (if the DCL command invokes a utility, that utility is invoked in a subprocess). Control returns to the debugging session when the DCL command terminates (or when you exit the utility). The following example illustrates spawning the DCL DIRECTORY command.

```
DBG> SPAWN DIR [JONES.PROJECT2]*.FOR
   .
   .
   .
%DEBUG-I-RETURNED, control returned to process JONES_1
DBG>
```

The next example illustrates spawning the DCL MAIL command, which invokes the MAIL utility:

```
DBG> SPAWN MAIL
MAIL> READ/NEW
   .
   .
   .
MAIL> EXIT
%DEBUG-I-RETURNED, control returned to process JONES_1
DBG>
```

If you enter the SPAWN command without specifying a parameter, a subprocess is created, and you can then enter DCL commands. Either logging out of the subprocess or attaching to the parent process (with the DCL ATTACH command) returns you to the debugging session. For example:

```
DBG> SPAWN
$ RUN PROG2
   .
   .
   .
$ ATTACH JONES_1
%DEBUG-I-RETURNED, control returned to process JONES_1
DBG>
```

If you plan to go back and forth several times between your debugging session and a spawned subprocess (which may be another debugging session), use the debugger ATTACH command to attach to that subprocess. Use the DCL ATTACH command to return to the parent process. Because you do not create a new subprocess every time you leave the debugger, you use system resources more efficiently.

If you are running two debugging sessions simultaneously, you can define a new debugger prompt for one of the sessions with the SET PROMPT command. This helps you to differentiate the sessions.

2.2 Commands That Cause Program Execution

Only four debugger commands can cause your program to execute: GO, STEP, CALL, and EXIT.

As indicated in Section 1.2.5.1, GO and STEP are the basic commands for starting and resuming program execution. The STEP command is discussed further in Section 2.3.

During a debugging session, routines are executed as they are called during the execution of a program. The CALL command enables you to arbitrarily call and execute a routine that was linked with your program. This command is discussed in Section 7.7.

The EXIT command was discussed in Section 2.1.3, in conjunction with terminating a debugging session. Because it executes any exit handlers in your program, it is also useful for debugging exit handlers (see Section 8.5).

When using any of these four commands, keep in mind that program execution may be interrupted or stopped by any of the following events:

- The program terminates.
- A breakpoint is reached.
- A watchpoint is activated.
- An exception is signaled.
- You press CTRL/Y.

2.3 Using the Step Command

The STEP command (probably the most frequently used debugger command) enables you to execute your program in small increments.

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>
```

Execution is now suspended at the first machine code instruction for line 27 of module TEST. Line 27 is in COUNT, a routine within module TEST.

The STEP command can also execute several source lines at a time. If you specify a positive integer as parameter, the STEP command executes that number of lines. In the following example, the STEP command executes the next three lines:

```
DBG> STEP 3
stepped to TEST\COUNT\%LINE 34
     34:    SWAP(X,Y);
DBG>
```

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. Also, if a line has more than one statement on it, the debugger executes all the statements on that line as part of the single step.

Source lines are displayed by default after stepping if they are available for the module being debugged. Source lines are not available if you are stepping in code that has not been compiled or linked with the /DEBUG qualifier (for example, a shareable image routine). If source lines are available, you can control their display with the SET STEP [NO]SOURCE command and the /[NO]SOURCE qualifier of the STEP command. See Chapter 5 for information on how to control the display of source code in general and in particular after stepping.

2.3.1 Changing the STEP Command Behavior

The default behavior of the STEP command may be altered in the following two ways:

- By specifying a STEP command qualifier — for example, STEP/INSTRUCTION.
- By establishing a new default qualifier with the SET STEP command — for example, SET STEP INSTRUCTION.

In the following example, the command STEP/INSTRUCTION executes the next instruction rather than the next line (STEP/LINE is the default behavior). The debugger displays the source line (10) associated with the new PC value (instruction TSTL):

```
DBG> STEP/INSTRUCTION
stepped to SQUARES$MAIN\%LINE 10+4: TSTL      W^-164(R11)[R0]
    10:         IF(INARR(I) .NE. 0) THEN
DBG>
```

After the STEP/INSTRUCTION command executes, subsequent STEP commands revert to the default behavior.

In contrast, the SET STEP command enables you to establish new defaults for the STEP command. These defaults remain in effect until another SET STEP command is entered. For example, the command SET STEP INSTRUCTION causes subsequent STEP commands to behave like STEP/INSTRUCTION (SET STEP LINE causes subsequent STEP commands to behave like STEP/LINE).

There is a SET STEP command parameter for each STEP command qualifier.

You can override the current STEP command defaults for the duration of a single STEP command by specifying other qualifiers. Use the SHOW STEP command to identify the current STEP command defaults.

2.3.2 Stepping into and over Routines

By default, when the PC is at a call statement and you enter the STEP command, the debugger steps "over" the called routine. Although the routine is executed, execution is not suspended within the routine but, rather, on the start of the line that follows the call statement. When stepping by instruction, execution is suspended on the instruction that follows a called routine's RET (return from routine) instruction.

To step into a called routine when the PC is at a call statement, enter the STEP/INTO command. The following example shows how to step into the routine PRODUCT, which is called from routine COUNT of module TEST:

```
DBG> STEP
stepped to TEST\COUNT\%LINE 18
    18:       AREA := PRODUCT(LENGTH, WIDTH);
DBG> STEP/INTO
stepped to routine TEST\PRODUCT
     6:    function PRODUCT(X,Y : INTEGER) return INTEGER is
DBG>
```

To return to the calling routine from any point within the called routine, use the STEP/RETURN command. It causes the debugger to step to the RET instruction of the routine being executed. A subsequent STEP command brings you back to the statement that follows the routine call. For example:

```
DBG> STEP/RETURN
stepped on return from TEST\PRODUCT\%LINE 11 to TEST\PRODUCT\%LINE 15+4
    15:    end PRODUCT;
DBG> STEP
stepped to TEST\COUNT\%LINE 19
    19:       LENGTH := LENGTH + 1;
DBG>
```

To step into several routines, enter the command SET STEP INTO to change the default behavior of the STEP command from STEP/OVER to STEP/INTO:

```
DBG> SET STEP INTO
DBG>
```

As a result of this command, when the PC is at a call statement, a STEP command suspends execution *within* the called routine. If you later want to step over routine calls, enter the command SET STEP OVER.

When SET STEP INTO is in effect, you can qualify the kinds of called routines that the debugger is stepping into by specifying any of the following parameters with the SET STEP command:

- [NO]JSB — controls whether to step into routines called by JSB instructions.
- [NO]SHARE — controls whether to step into called routines in shareable images.
- [NO]SYSTEM — controls whether to step into called system routines.

These parameters make it possible to step into your own (user) routines and automatically step over system routines, and so on. For example, the following command tells the debugger to step into called routines in user space only. The debugger steps over routines in system space and in shareable images.

```
DBG> SET STEP INTO,NOSYSTEM,NOSHARE
DBG>
```

2.4 Suspending and Tracing Execution with Breakpoints and Tracepoints

This section discusses use of the SET BREAK and SET TRACE commands to, respectively, *suspend* and *trace* program execution. The commands are discussed together because of their similarities.

SET BREAK Command Overview

The SET BREAK command enables you to specify program locations or events at which to suspend program execution (breakpoints). After setting a breakpoint, you can start or resume program execution with the GO command, letting the program run until the specified location or condition is reached. When the breakpoint is triggered, the debugger suspends execution, identifies the breakpoint, and displays the DBG> prompt. You can then enter debugger commands—for example, to determine where you are (with the SHOW CALLS command), step into a routine, examine or modify variables, and so on.

The syntax of the SET BREAK command is as follows:

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

The following example shows a typical use of the SET BREAK command and illustrates the general default behavior of the debugger at a breakpoint.

In this 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:

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

SET TRACE Command Overview

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

The debugger's default behavior at a tracepoint is identical to that at a breakpoint, except that program execution continues past a tracepoint. Thus, the DBG> prompt is not displayed when a tracepoint is reached and announced by the debugger.

Except for the command name, the syntax of the SET TRACE command is identical to that of the SET BREAK command:

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

The SET TRACE and SET BREAK commands have the same qualifiers. When using the SET TRACE command, you specify address expressions, qualifiers, and the optional WHEN and DO clauses exactly as with the SET BREAK command.

Unless you use the /TEMPORARY qualifier on the SET BREAK (or SET TRACE) command, breakpoints (and tracepoints) remain in effect until you cancel them or exit the debugging session.

To identify all of the breakpoints (or tracepoints) that are currently set, use the SHOW BREAK (or SHOW TRACE) command. To cancel breakpoints (or tracepoints), use the CANCEL BREAK (or CANCEL TRACE) command (see Section 2.4.6).

The following sections describe how to specify program locations and events with the SET BREAK and SET TRACE commands.

2.4.1 Setting Breakpoints or Tracepoints on Individual Program Locations

To set a breakpoint (or a tracepoint) on a particular program location, specify an address expression with the SET BREAK (or SET TRACE) command.

Fundamentally, an address expression specifies a location in virtual memory or a VAX register. Because the debugger understands the symbols associated with your program, the address expressions you typically use with the SET BREAK (or SET TRACE) command are routine names, labels, or source line numbers rather than virtual memory addresses — the debugger converts these symbols to addresses.

2.4.1.1 Specifying Symbolic Addresses

NOTE: In some cases, when using the SET BREAK or SET TRACE command with a symbolic address expression, you may need to set a module or specify a scope or a path name. Those concepts are described in detail in Chapter 4. The examples in this section assume that all modules are set and that all symbols referenced are uniquely defined, unless otherwise indicated.

The following examples illustrate how to set a breakpoint (or a tracepoint) on a routine (SWAP) and a label (LOOP1):

```
DBG> SET BREAK SWAP
DBG> SET TRACE LOOP1
DBG>
```

The next command sets a breakpoint on the RET (return) instruction of routine SWAP. "Breaking" on the RET instruction of a routine enables you to inspect the local environment before the RET instruction removes the routine's call frame from the call stack.

```
DBG> SET BREAK/RETURN SWAP
DBG>
```

Some languages, for example FORTRAN, use numeric labels. To set a breakpoint (or a tracepoint) on a numeric label, you must precede the number with the built-in symbol %LABEL. Otherwise, the debugger interprets the number as a virtual memory address. For example, the following command sets a tracepoint on label 20.

```
DBG> SET TRACE %LABEL 20
DBG>
```

You can set a breakpoint (or a tracepoint) on a line of source code by specifying the line number preceded by the built-in symbol %LINE. The following command sets a breakpoint on line 14.

```
DBG> SET BREAK %LINE 14
DBG>
```

The preceding breakpoint causes execution to be suspended when the PC value is on the first instruction of line 14. Note that you can set a breakpoint (or a tracepoint) only on lines for which the compiler generated instructions (lines that resulted in executable code). If you specify a line number that is not associated with an instruction, such as a comment line or a statement that declares but does not initialize a variable, the debugger issues a diagnostic message. For example:

```
DBG> SET BREAK %LINE 6
%DEBUG-I-LINEINFO, no line 6, previous line is 5, next line is 8
%DEBUG-E-NOSYMBOL, symbol '%LINE 6' is not in the symbol table
DBG>
```

The preceding messages indicate that the compiler did not generate instructions for lines 6 or 7 in this case.

Some languages, for example BASIC, allow more than one statement on a line. In such cases, you can use statement numbers 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 is as follows:

```
%LINE line-number.statement-number
```

For example, the following command sets a tracepoint on the second statement of line 38:

```
DBG> SET TRACE %LINE 38.2
DBG>
```

When searching for symbols that you reference in commands, the debugger uses the conventions described in Section 4.3.1. That is, it first looks within the module where execution is currently suspended, then in other scopes associated with routines on the call stack, and so on. Therefore, to specify a symbol that is defined in more than one module, such as a line number, you may need to use a path name. For example, the following command sets a tracepoint on line 27 of module MOD4:

```
DBG> SET TRACE MOD4\%LINE 27
DBG>
```

Remember the symbol lookup conventions when specifying a line number in debugger commands. If that line number is not defined in the module where execution is suspended (because it is not associated with an instruction), the debugger uses the symbol lookup conventions to locate another module where the line number is defined.

When specifying address expressions, you can combine symbolic addresses with byte offsets. Thus, you can set a breakpoint (or a tracepoint) on a particular assembly language instruction by specifying its line number and the byte offset from the start of that line to the first byte of the instruction. For example, the next command sets a breakpoint on the address that is five bytes beyond the start of line 23.

```
DBG> SET BREAK %LINE 23+5
DBG>
```

2.4.1.2 Specifying Locations in Virtual Memory

To set a breakpoint (or a tracepoint) on a location in virtual memory, specify its numerical address in the currently set radix The default radix for both data entry and display is decimal for all languages except BLISS and MACRO. It is hexadecimal for BLISS and MACRO. For example, the following command sets a breakpoint at address 2753, decimal (for all languages except BLISS or MACRO), or at address 2753, hexadecimal (for BLISS and MACRO):

```
DBG> SET BREAK 2753
DBG>
```

You can specify a radix when you enter an individual integer literal (such as 2753) by using one of the built-in symbols %BIN, %OCT, %DEC, or %HEX. For example, in the following command line the symbol %HEX specifies that 2753 should be treated as a hexadecimal integer:

```
DBG> SET BREAK %HEX 2753
DBG>
```

Note that when specifying a hexadecimal number that starts with a letter rather than a number, you must add a leading "0". Otherwise, the debugger tries to interpret the entity specified as a symbol declared in your program.

See Section 3.1.9 and Appendix D for additional information on specifying radixes and on the built-in symbols %BIN, %DEC, %HEX, and %OCT.

If a breakpoint (or a tracepoint) was set on a numerical address that corresponds to a symbol in your program, the SHOW BREAK (or SHOW TRACE) command identifies the breakpoint symbolically.

2.4.1.3 Obtaining and Symbolizing Virtual Memory Addresses

Use the EVALUATE/ADDRESS command to determine the virtual memory address associated with a symbolic address expression, such as a line number, routine name, or label. For example:

```
DBG> EVALUATE/ADDRESS SWAP
1536
DBG> EVALUATE/ADDRESS %LINE 26
1629
DBG>
```

The address is displayed in the current radix. You can specify a radix qualifier to display the address in another radix. For example:

```
DBG> EVALUATE/ADDRESS/HEX %LINE 26
0000065D
DBG>
```

The command SYMBOLIZE does the reverse of EVALUATE/ADDRESS. It converts a virtual memory address into its symbolic representation (including its path name) if such a representation is possible. Chapter 4 explains how to control symbolization. See Section 3.1.10 for more information on obtaining and symbolizing addresses.

2.4.2 Setting Breakpoints or Tracepoints on Consecutive Lines or on Classes of Instructions

Several SET BREAK (and SET TRACE) command qualifiers cause the debugger to break on (or trace) every source line or every assembly language instruction of a particular class:

```
/LINE
/BRANCH
/CALL
/INSTRUCTION
/INSTRUCTION=(opcode[, . . . ])
```

When using these qualifiers, do not specify an address expression.

For example, the following command causes the debugger to break on the start of every source line encountered during execution:

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

The instruction-related qualifiers are especially useful for opcode tracing, which is the tracing of all instructions or the tracing of a class of instructions. The next command causes the debugger to trace every branch instruction encountered (for example BEQL, BGTR, and so on):

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

Note that opcode tracing slows program execution.

By default, when you use the qualifiers discussed in this section, the debugger breaks (or traces) within all called routines as well as within the currently executing routine (this is equivalent to specifying SET BREAK/INTO or SET TRACE/INTO). By specifying SET BREAK/OVER or SET TRACE/OVER, you can suppress break (or trace) action within all called routines. Or, you can use the /[NO]JSB, /[NO]SHARE, or /[NO]SYSTEM qualifiers to specify the kinds of called routines where break (or trace) action is to be suppressed. For example, the next command causes the debugger to break on every line except within called routines that are in shareable images or system space:

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

2.4.3 Controlling Debugger Action at Breakpoints or Tracepoints

The SET BREAK and SET TRACE commands provide several options for controlling the behavior of the debugger at breakpoints and tracepoints — the /AFTER, /[NO]SILENT, /[NO]SOURCE, and /TEMPORARY command qualifiers, and the optional WHEN and DO clauses. The following examples illustrate several of these options.

The next command sets a breakpoint on line 14 and specifies that the breakpoint take effect after the fifth time that line 14 is executed:

```
DBG> SET BREAK/AFTER:5 %LINE 14
DBG>
```

The next command sets a tracepoint that is triggered at every line of execution. The DO clause obtains the value of the variable X when each line is executed:

```
DBG> SET TRACE/LINE DO (EXAMINE X)
DBG>
```

The next example illustrates how the WHEN and DO clauses can be used together. The command sets a breakpoint at line 27. The breakpoint is triggered (execution is suspended) only when the value of SUM is greater than 100 (not each time line 27 is executed). The DO clause causes the value of TEST_RESULT to be examined whenever the breakpoint is triggered—that is, whenever the value of SUM is greater than 100. If the value of SUM is not greater than 100 when execution reaches line 27, the breakpoint is not triggered and the DO clause is not executed.

```
DBG> SET BREAK %LINE 27 WHEN (SUM >100) DO (EXAMINE TEST_RESULT)
DBG>
```

See Section 3.1.5 and Section 8.3.2.2 for information about evaluating language expressions, such as the expression "SUM >100".

The /SILENT qualifier suppresses the break or trace message and source code display. This is useful when, for example, you want to use the SET TRACE command only to execute a debugger command at the tracepoint. In the next example, the SET TRACE command is used to examine the value of the boolean variable STATUS at the tracepoint.

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

In the next example, the SET TRACE command is used to count the number of times line 12 is executed. The first DEFINE/VALUE command defines a symbol COUNT and initializes its value to zero. The DO clause of the SET TRACE command causes the value of COUNT to be incremented and evaluated whenever the tracepoint is triggered (whenever execution reaches line 12).

```
DBG> DEFINE/VALUE COUNT=0
DBG> SET TRACE/SILENT %LINE 12 DO (DEF/VAL COUNT=COUNT+1;EVAL COUNT)
DBG>
```

Source lines are displayed by default at eventpoints (breakpoints, tracepoints, and watchpoints) if they are available for the module being debugged. You can also control their display with the SET STEP [NO]SOURCE command and the /[NO]SOURCE qualifier of the SET BREAK, SET TRACE, and SET WATCH commands. See Chapter 5 for information on how to control the display of source code in general and in particular at eventpoints.

2.4.4 Setting Breakpoints or Tracepoints on Exceptions

The SET BREAK/EXCEPTION and SET TRACE/EXCEPTION commands direct the debugger to treat any exception generated by your program as a breakpoint or tracepoint, respectively. The breakpoint (or tracepoint) occurs before any user-declared exception handler is invoked. See Section 8.4 for debugging techniques associated with exceptions and condition handlers.

2.4.5 Setting Breakpoints or Tracepoints on Language-Specific Events

The SET BREAK and SET TRACE commands each have an /EVENT=event-name qualifier. You can use this qualifier to set breakpoints or tracepoints to be triggered by various language-specific events (denoted by event-name keywords).

Note: Currently, event-name keywords are defined only for Ada and SCAN. See the VAX Ada and VAX SCAN documentation for complete information.

When you run a program under debugger control, the appropriate set of event-name keywords is defined during the initialization of language-specific parameters. Use the SHOW EVENT_FACILITY command to identify the event-name keywords that apply to the current language. The SET EVENT_FACILITY command enables you to initialize the debugger for events that are specific to another language.

The following examples briefly illustrate how to set event breakpoints with Ada tasking programs and SCAN programs. When a breakpoint or tracepoint is triggered, the debugger identifies the event that caused it to be triggered and gives additional information.

The following command causes the debugger to break whenever any Ada task enters the TERMINATED state.

```
DBG> SET BREAK/EVENT=TERMINATED
DBG>
```

The next command sets two tracepoints, which are associated with the Ada tasks CHECKIN and RESERVE, respectively. Each tracepoint is triggered whenever its associated task makes a transition to the RUN state.

```
DBG> SET TRACE/EVENT=RUN CHECKIN,RESERVE
DBG>
```

The next command causes the debugger to break whenever a SCAN token is built, for any value.

```
DBG> SET BREAK/EVENT=TOKEN
DBG>
```

See Section 8.3.2 for information on predefined Ada event breakpoints.

2.4.6 Canceling Breakpoints or Tracepoints

Use the CANCEL BREAK and CANCEL TRACE commands to cancel breakpoints and tracepoints, respectively. To cancel a breakpoint (or a tracepoint), specify address expressions and qualifiers exactly as you specified them when setting the breakpoint (or tracepoint).

Thus, to cancel breakpoints (or tracepoints) that were set at specific address expressions, specify those same address expressions. For example:

```
DBG> CANCEL BREAK SWAP,MOD2\LOOP4,2753
DBG>
```

To cancel breakpoints (or tracepoints) that were set with the following command qualifiers, specify those same command qualifiers: /BRANCH, /CALL, /EVENT=event-name, /EXCEPTION, /INSTRUCTION, /INSTRUCTION=(opcode[, . . .]), /LINE. If the qualifier requires one or more keywords, include the keywords associated with the breakpoints or tracepoints that are to be canceled. For example:

```
DBG> CANCEL BREAK/LINE
DBG> CANCEL TRACE/INSTRUCTION=(JSB,CALLS)
DBG> CANCEL TRACE/EVENT=RUN CHECKIN
DBG>
```

2.5 Monitoring Changes in Variables and Other Program Locations

Note: This section describes the general use of the SET WATCH command. Section 2.5.2 gives additional information pertaining to setting watchpoints on nonstatic variables — variables that are allocated on the stack or in registers.

Also, in some cases, when using the SET WATCH command with a variable name (or any other symbolic address expression) you may need to set a module or specify a scope or a path name. Those concepts are described in Chapter 4. The examples in this section assume that all modules are set and that all variable names are uniquely defined.

The SET WATCH command enables you to specify program variables (or arbitrary memory locations) that the debugger monitors as your program executes. This process is called setting watchpoints. If, during execution, the program modifies the value of a "watched" variable (or memory location), the watchpoint is triggered. The debugger then suspends execution, displays information, and prompts for more commands. The debugger monitors watchpoints continuously during program execution.

The syntax of the SET WATCH command is as follows:

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

Although any valid address expression may be specified, usually you specify the name of a variable. The example that follows shows a typical use of the SET WATCH command and illustrates the general default behavior of the debugger at a watchpoint.

```
DBG> SET WATCH COUNT
DBG> GO
   .
   .
   .
watch of MOD2\COUNT at MOD2\%LINE 24
     24:   COUNT := COUNT + 1;
    old value: 27
    new value: 28
break at MOD2\%LINE 25
     25:   END;
DBG>
```

In this example, the SET WATCH command sets a watchpoint on the variable COUNT, and the GO command starts execution. When the program changes the value of COUNT, execution is suspended. The debugger then does the following:

- Announces the event ("watch of MOD2\COUNT . . . "), identifying the location of the instruction that changed the value of the watched variable (" . . . at MOD2\%LINE 24")
- Displays the associated source line (24)
- Displays the old and new values of the variable (27 and 28)
- Announces that execution has been suspended at the start of the next line ("break at MOD2\%LINE 25") and displays that source line
- Prompts for another command.

When the address of the instruction that modified a watched variable is not at the start of a source line, the debugger denotes the instruction's location by displaying the line number plus the byte offset from the start of the line. For example:

```
DBG> SET WATCH K
DBG> GO
   .
   .
   .
watch of TEST\K at TEST\%LINE 19+5
     19:   DO 40 K = 1, J
    old value: 4
    new value: 5
break at TEST\%LINE 19+9
     19:   DO 40 K = 1, J
DBG>
```

In this example, the address of the instruction that modified variable K is 5 bytes beyond the start of line 19. Note that the breakpoint is on the instruction that follows the instruction that modified the variable (not on the start of the next source line as in the preceding example).

You can set watchpoints on aggregates (that is, entire arrays or records). A watchpoint set on an array or record triggers 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. In the following example, the watchpoint is triggered because element 3 of array ARR was modified:

```
DBG> SET WATCH ARR
DBG> GO
    .
    .
    .
watch of SUBR\ARR at SUBR\%LINE 12
    12:     ARR(3) := 28
   old value:
    (1):          7
    (2):          12
    (3):          3
    (4):          0

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

break at SUBR\%LINE 13
DBG>
```

To identify all of the watchpoints that are currently set, use the SHOW WATCH command. To cancel watchpoints, use the CANCEL WATCH command.

Note that the SET BREAK/MODIFY command has the same effect as the SET WATCH command.

2.5.1 Watchpoint Options

The SET WATCH command provides the same options for controlling the behavior of the debugger at watchpoints that the SET BREAK and SET TRACE commands provide for breakpoints and tracepoints — namely the /AFTER, /[NO]SILENT, /[NO]SOURCE, and /TEMPORARY command qualifiers, and the optional WHEN and DO clauses. See Section 2.4.3 for examples.

2.5.2 Watching Nonstatic Variables

Storage for a variable in your program is allocated either statically or nonstatically. A *static variable* is associated with the same virtual memory address throughout execution of the program. A *nonstatic variable* is allocated on the stack or in a register and has a value only when its defining routine is active (on the call stack). As explained in this section, the technique for setting a watchpoint, the watchpoint's behavior, and the speed of program execution are different for the two kinds of variables.

To determine how a variable is allocated, use the EVALUATE/ADDRESS command. A static variable generally has its address in P0 space (0 through 3FFFFFFF, hexadecimal). A nonstatic variable generally has its address in P1 space (40000000 through 7FFFFFFF, hexadecimal) or is in a register. In the following Pascal code example, X is declared as a static variable, whereas Y is a nonstatic variable (by default). The EVALUATE/ADDRESS command, entered while debugging, shows that X is allocated at memory location 512, whereas Y is allocated in register R0:

```
 . . .
VAR
     X: [STATIC] INTEGER;
     Y: INTEGER;
 . . .

DBG> EVALUATE/ADDRESS X
512
DBG> EVALUATE/ADDRESS Y
%R0
DBG>
```

When using the SET WATCH command, note the following distinction. You can set a watchpoint on a *static* variable regardless of the PC value when you enter the command; but you can set a watchpoint on a *nonstatic* variable only when the PC value is within the routine where that variable is defined. Otherwise, the debugger issues a warning. For example:

```
DBG> SET WATCH Y
%DEBUG-W-SYMNOTACT, nonstatic variable 'MOD4\ROUT3\Y' is not active
DBG>
```

Section 2.5.2.2 describes how to set a watchpoint on a nonstatic variable.

2.5.2.1 Execution Speed

When a watchpoint is set, the speed of program execution depends on whether the variable is static or nonstatic. To watch a static variable, the debugger write-protects the page containing the variable. If your program attempts to write to that page (modify the value of that variable), an access violation occurs and the debugger handles the exception. The debugger temporarily unprotects the page to allow the instruction to complete and then determines whether the watched variable was modified. Except when writing to that page, the program executes at full speed.

Because problems arise if the stack or registers are write-protected, the debugger must use another technique to watch a nonstatic variable. It traces every instruction in the variable's defining routine and checks the value of the variable after each instruction has been executed. Because this significantly slows down the execution of the program, the debugger issues the following message when you set a nonstatic watchpoint:

```
DBG> SET WATCH Y
%DEBUG-I-WPTTRACE, nonstatic watchpoint, tracing every instruction
DBG>
```

2.5.2.2 Setting a Watchpoint on a Nonstatic Variable

To set a watchpoint on a nonstatic variable, make sure that the PC value is within the defining routine. A convenient technique is to set a tracepoint on that routine, also specifying a DO clause to set the watchpoint. Thus, whenever the routine is called, the tracepoint is triggered and the watchpoint is automatically set on the local variable. In the following example, the WPTTRACE message indicates that a watchpoint has been set on Y, a nonstatic variable that is local to routine ROUT3:

```
DBG> SET TRACE/NOSOURCE ROUT3 DO (SET WATCH Y)
DBG> GO
    .
    .
    .
trace at routine MOD4\ROUT3
%DEBUG-I-WPTTRACE, nonstatic watchpoint, tracing every instruction
    .
    .
    .
watch of MOD4\ROUT3\Y at MOD4\ROUT3\%LINE 16
   16:     Y := 4
   old value:    3
   new value:    4
break at MOD4\ROUT3\%LINE 17
   17:     SWAP(X,Y);
DBG>
```

When execution returns to the caller of routine ROUT3, variable Y is no longer active. Therefore, the debugger automatically cancels the watchpoint and issues the following messages:

```
%DEBUG-I-WATCHVAR, watched variable MOD4\ROUT3\Y has gone out of scope
%DEBUG-I-WATCHCAN, watchpoint now cancelled
```

2.5.2.3 Options for Watching Nonstatic Variables

The SET WATCH command qualifiers /OVER, /INTO, and /[NO]STATIC provide options for watching nonstatic variables.

When you set a watchpoint on a nonstatic variable, you can direct the debugger to do one of two things at a routine call:

- Step over the called routine — executing it at full speed — and resume instruction tracing after returning. This is the default (SET WATCH/OVER).
- Trace instructions within the called routine, thereby monitoring the variable instruction-by-instruction within the routine (SET WATCH/INTO).

Using the SET WATCH/OVER command results in better performance. But it also means that, if the called routine modifies the watched variable, the watchpoint is triggered only after execution returns from that routine. The SET WATCH/INTO command slows down program execution but enables you to monitor watchpoints more precisely within called routines.

The debugger determines whether a variable is static or nonstatic by looking at its address (P0 space, P1 space, or register). When entering a SET WATCH command, you can override this decision with the /[NO]STATIC qualifier. For example, if you have allocated nonstack storage in P1 space, use the SET WATCH/STATIC command to tell the debugger that a particular variable is static even though it is in P1 space. Conversely, if you have allocated your own stack in P0 space, use the SET WATCH/NOSTATIC command to tell the debugger that a particular variable is nonstatic even though it is in P0 space.

2.6 How the Debugger Controls Program Execution

This section is for readers who are interested in how the debugger functions.

The debugger controls and monitors execution by causing exceptions to occur at points of interest in your program. For example, it may put a breakpoint fault instruction (BPT) in your code, causing a breakpoint exception to occur when that instruction is executed. The debugger may also set the trace enable bit (T bit) in the processor status longword (PSL), thus causing a trace trap at the end of each instruction.

When you run your program with the debugger, the debugger is the primary exception handler. Any exception resulting from the execution of your program, whether or not it is caused by the debugger, is first handled by the debugger. If the debugger did not cause the exception, it resignals the exception (refer to Section 8.4 for information and debugging techniques related to exceptions and condition handlers). If the debugger caused the exception, it takes appropriate action. For example, in the case of a tracepoint the debugger identifies the tracepoint and returns control to the program. In the case of a breakpoint, the debugger maintains control by identifying the breakpoint and then prompting for commands.

The following paragraphs illustrate the functioning of the debugger with some typical commands — SET BREAK and STEP. See also Sections 2.5.2 and 8.4 for implementation information on the SET WATCH and SET BREAK/EXCEPTION commands, respectively.

When you set a breakpoint, specifying a particular address expression, the debugger removes the opcode at that address and replaces it with the BPT instruction. When execution reaches that address, the BPT instruction causes a breakpoint fault, which gives control to the debugger:

1. The debugger announces the breakpoint and prompts for commands. When you resume execution, the debugger performs the following steps.
2. The debugger replaces the original opcode and sets the T bit of the saved PSL on the stack, so that a trace trap occurs when the current instruction is executed.
3. The instruction is executed.
4. When the trace trap occurs, the debugger replaces the BPT instruction at the original breakpoint address, so that the break fault occurs whenever execution again reaches that address.

When you enter a STEP/INSTRUCTION command, the debugger sets the T bit of the saved PSL, executes the next instruction, then, when the trace trap occurs, issues a message and prompts for commands.

The STEP/LINE command is implemented similarly, except that the debugger keeps track of line boundaries by correlating the low and high PC values of each line with data stored in the symbol table. The debugger completes the step and prompts for commands when you leave the current line.

When you set a breakpoint on a class of instructions and then start execution, the debugger traces (traps on) every instruction by setting the T bit of the saved PSL. If the next instruction is of the desired class, the debugger suspends execution on that instruction, announces the breakpoint, and prompts for commands. If the instruction is not of the desired class, the debugger continues to trace and execute instructions.

When you enter a STEP/OVER command at a routine call, the debugger does the following:

1. Steps into the routine, then sets a reserved bit in the saved PSL.
2. Lets the program run. The routine is executed, but the RET instruction causes a reserved-operand exception when it tries to restore the modified PSL.
3. Lets the RET instruction complete but sets the T bit to suspend execution after the RET instruction (in the calling routine) on the instruction that follows the original call.

STEP/RETURN is also implemented by setting a reserved bit in the saved PSL.

Because the debugger and your program share the same address space, in some rare cases they may interfere with each other, causing unexpected behavior. The following paragraphs highlight possible sources of interference.

Effect of Debugger on Uninitialized Variables

Because the debugger acts as an exception handler, it uses the stack. This may cause uninitialized variables saved on the stack to be modified by the debugger.

If your program references an uninitialized variable that is in this state, the execution of the program may be affected.

Effect of Debugger on Memory Usage

Another source of possible interference between the debugger and your program is that they share virtual memory. If your program is sensitive to changes in memory usage, the execution of the program may be affected.

3 Examining and Manipulating Program Data

This chapter explains how to use the EXAMINE and DEPOSIT commands to display and modify the values of symbols declared in your program as well as the contents of arbitrary program locations. The chapter also explains how to use the EVALUATE and other commands that evaluate language expressions.

The topics covered in this chapter are organized as follows:

- General concepts related to using the EXAMINE, DEPOSIT, and EVALUATE commands.
- Use of the commands with symbolic names — for example, the names of variables and routines declared in your program. Such symbolic address expressions are associated with compiler generated types.
- Use of the commands with program locations (virtual memory addresses or registers) that do not have symbolic names. Such address expressions are not associated with compiler generated types.
- Specifying a type to override the type associated with an address expression.

The examples in this chapter do not cover all language-dependent behavior. When debugging in any language, be sure to consult the documentation supplied with that language. The chapter devoted to debugging in the user's guide contains all language-dependent information for that language. The following sections of this manual also contain language-related information:

- Appendix E tabulates the constructs and operators that are supported by the debugger for each language.
- Section 8.3 highlights some important differences between languages that you should be aware of when debugging multilanguage programs.

3.1 General Concepts

This section introduces the EXAMINE, DEPOSIT, and EVALUATE commands and discusses concepts that are common to those commands.

3.1.1 Accessing Variables While Debugging

Before you try to examine or deposit into a nonstatic (stack-local or register) variable, its defining routine must be active (on the call stack). That is, program execution must be suspended somewhere within the defining routine. See Section 2.5.2 for more information about nonstatic variables.

You can examine a static variable at any time during program execution, and you can examine a nonstatic variable as soon as execution reaches its defining routine. However, before you examine any variable, you should step or otherwise execute the program beyond the point where the variable

is declared and initialized. The value contained in any uninitialized variable should be considered invalid.

Many compilers optimize code to make the program run faster. If the code that you are debugging has been optimized, some program locations may not match what you might expect from looking at the source code. In particular, some optimization techniques eliminate certain variables, so that you no longer have access to them while debugging.

Section 8.1 explains the effect of several optimization techniques on the executable code. When first debugging a program, it is best to disable optimization, if possible, with the /NOOPTIMIZE (or equivalent) compiler command qualifier.

Note that, in some cases, when using the EXAMINE or DEPOSIT command with a variable name (or any other symbolic address expression) you may need to set a module or specify a scope or a path name. Those concepts are described in Chapter 4. The examples in this chapter assume that all modules are set and that all variable names are uniquely defined.

3.1.2 Using the EXAMINE Command

For high-level language programs, the EXAMINE command is used mostly to display the current value of variables, and it has the following form:

```
EXAMINE variable-name[, . . . ]
```

Thus, for example, the following command displays the current value of the integer variable X:

```
DBG> EXAMINE X
MOD3\X:   17
DBG>
```

When displaying the value, the debugger prefixes the variable name with its path name — in this case, the name of the module where variable X is declared (see Section 4.3.2).

More generally, the EXAMINE command displays the current value of the entity denoted by an address expression, in the type associated with that location (for example, integer, real, array, record, and so on). The basic format of the EXAMINE command is as follows:

```
EXAMINE address-expression[, . . . ]
```

When you enter an EXAMINE command, the debugger evaluates the address expression to yield a program location (a virtual memory address or a register). The debugger then displays the value stored at that location as follows:

- If the location has a symbolic name, the debugger formats the value according to the compiler generated type associated with that symbol.
- If the location does not have a symbolic name, the debugger formats the value in the type longword integer, by default.

See Section 3.1.4 for more information on the types associated with symbolic and nonsymbolic address expressions.

By default, when displaying the value, the debugger identifies the address expression and its path name symbolically if symbol information is available. See Section 3.1.10 for additional information about symbolization of addresses.

3.1.3 Using the DEPOSIT Command

For high-level languages, the DEPOSIT command is used mostly to assign a new value to a variable. The command is like an assignment statement in most programming languages, and it has the following form:

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

Thus, for example, the following DEPOSIT command assigns the value 23 to the integer variable X:

```
DBG> EXAMINE X
MOD3\X:   17
DBG> DEPOSIT X = 23
DBG> EXAMINE X
MOD3\X:   23
DBG>
```

More generally, the DEPOSIT command evaluates a language expression and deposits the resulting value into a program location denoted by an address expression. The basic format of the DEPOSIT command is as follows:

```
DEPOSIT address-expression = language-expression
```

When you enter a DEPOSIT command, the debugger does the following:

- It evaluates the address expression to yield a program location.
- If the program location has a symbolic name, the debugger associates the location with the symbol's compiler generated type. If the location does not have a symbolic name, the debugger associates the location with the type longword integer, by default (see Section 3.1.4).
- It evaluates the language expression in the syntax of the current language and in the current radix to yield a value. This behavior is identical to that of the EVALUATE command (see Section 3.1.5).
- It checks that the value and type of the language expression is consistent with the type of the address expression. If you try to deposit a value that is incompatible with the type of the address expression, the debugger issues a diagnostic message. If the value is compatible, the debugger deposits the value into the location denoted by the address expression.

Note that the debugger may do type conversion during a deposit operation if the language rules allow it. For example, assume X is an integer variable. In the following example, the real value 2.0 is converted to the integer value 2, which is then assigned to X:

```
DBG> DEPOSIT X = 2.0
DBG> EXAMINE X
MOD3\X:   2
```

In general, the debugger tries to follow the assignment rules for the current language.

3.1.4 Address Expressions and Their Associated Types

The symbols that are declared in your program (variable names, routine names, and so on) are symbolic address expressions. They denote locations in virtual memory or in registers. Symbolic address expressions (also called symbolic names in this chapter) have compiler generated types, and the debugger knows the type and location that are associated with symbolic names. Section 3.1.10 explains how to obtain memory addresses and register names from symbolic names and how to symbolize program locations.

Symbolic names include the following categories:

- Variables. The associated program locations contain the current values of variables. Techniques for examining and depositing into variables are described in Section 3.2.
- Routines, labels, and line numbers. The associated program locations contain VAX assembly-language instructions. Techniques for examining and depositing VAX instructions are described in Section 3.3.

Program locations that do not have a symbolic name are not associated with a compiler generated type. To enable you to examine and deposit into such locations, the debugger associates them with the default type *longword integer*. This means that, if you specify a location that does not have a symbolic name, the EXAMINE command displays the contents of 4 bytes starting at the address specified and formats the displayed information as an integer value. In the following example, the virtual memory address 926 is not associated with a symbolic name (note that the address is not symbolized when the EXAMINE command is executed). Therefore, the EXAMINE command displays the value at that address as a longword integer:

```
DBG> EXAMINE 926
926:  749404624
DBG>
```

Similarly, by default you can deposit up to 4 bytes of integer data into a program location that does not have a symbolic name. And this data is formatted as a longword integer. For example:

```
DBG> DEPOSIT 926 = 84
DBG> EXAMINE 926
926:  84
DBG>
```

Techniques for examining and depositing into locations that do not have a symbolic name are described in Section 3.5.

The EXAMINE and DEPOSIT commands accept type qualifiers (/ASCII:n, /BYTE, and so on) that enable you to override the type associated with a program location. This is useful if you want the contents of the location to be interpreted and displayed in another type, or if you want to deposit some value of a particular type into a location that is associated with another type. Techniques for overriding a type are described in Section 3.5.

3.1.5 Evaluating Language Expressions

A language expression consists of any combination of one or more symbols, literals, and operators that is evaluated to a single value in the syntax of the current language and in the current radix. (The current language and current radix are defined in Section 3.1.8 and Section 3.1.9, respectively.) Several debugger commands and constructs evaluate language expressions:

- The EVALUATE and DEPOSIT commands, which are described in this section and in Section 3.1.3, respectively.
- The IF, FOR, REPEAT, and WHILE commands (see Section 7.6).
- WHEN clauses, which are used with the SET BREAK, SET TRACE, and SET WATCH commands (see Section 2.4.3).

Although this discussion applies to all commands and constructs that evaluate language expressions, it focuses on the use of the EVALUATE command.

The EVALUATE command evaluates one or more language expressions in the syntax of the current language and in the current radix and displays the resulting values. The command has the following form:

```
EVALUATE language-expression[, . . . ]
```

One use of the EVALUATE command is as a calculator, to perform arithmetic calculations that may be unrelated to your program. For example:

```
DBG> EVALUATE (8+12)*6/4
30
DBG>
```

The debugger uses the rules of operator precedence of the current language when evaluating language expressions.

You can also evaluate language expressions that include variables and other constructs. For example, the following EVALUATE command subtracts 3 from the current value of the integer variable X, multiplies the result by 4, and displays the resulting value:

```
DBG> DEPOSIT X = 23
DBG> EVALUATE (X - 3) * 4
80
DBG>
```

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 the types are incompatible, a diagnostic message is issued. Debugger support for operators and other constructs in language expressions is tabulated in Appendix E for each language. You can also obtain information by typing "HELP LANGUAGE *language-name*".

The built-in symbol %CURVAL denotes the *current value* — the value last displayed by an EVALUATE or EXAMINE command, or deposited by a DEPOSIT command. The backslash (\) also denotes the current value when used in that context. For example:

```
DBG> EXAMINE X
MOD3\X:  23
DBG> EVALUATE %CURVAL
23
DBG> DEPOSIT Y = 47
DBG> EVALUATE \
47
DBG>
```

3.1.5.1 Using Variables in Language Expressions

You can use variables in language expressions in much the same way that you use them in the source code of your program.

Thus, the debugger generally interprets a variable used in a language expression as the current *value* of that variable, not the address of the variable. For example (X is an integer variable):

```
DBG> DEPOSIT X = 12      ! Assign the value 12 to X
DBG> EXAMINE X           ! Display the value of X
MOD4\X:  12
DBG> EVALUATE X          ! Evaluate and display the value of X
12
DBG> EVALUATE X + 4      ! Add the value of X to 4
16
DBG> DEPOSIT X = X/2     ! Divide the value of X by 2 and assign
                         ! the resulting value to X
DBG> EXAMINE X           ! Display the new value of X
MOD4\X:   6
DBG>
```

Note that the use of a variable in a language expression as illustrated in the previous examples is generally limited to single-valued, nonstructured variables. Typically, you can specify a multi-valued, structured variable (like an array or record) in a language expression only if the syntax indicates that you are referencing only a single value (a single element of the structure). For example, if ARR is the name of an array of integers, the following command is invalid:

```
DBG> EVALUATE ARR
%DEBUG-W-NOVALUE, reference does not have a value
DBG>
```

However, the following commands are valid because only a single element of the array is referenced:

```
DBG> EVALUATE ARR(2)          ! Evaluate element 2 of array ARR
37
DBG> DEPOSIT K = 5 + ARR(2)   ! Deposit the sum of two integer values
DBG>                          ! into an integer variable
```

Note also that, if the current language is BLISS, the debugger interprets a variable in a language expression as the *address* of that variable. To denote the *value* stored in a variable, you must use the contents-of operator (period (.)). For example, when the language is set to BLISS:

```
DBG> EXAMINE Y          ! Display the value of Y.
MOD4\Y:  3
DBG> EVALUATE Y         ! Display the address of Y.
02475B
DBG> EVALUATE .Y        ! Display the value of Y.
3
DBG> EVALUATE Y + 4     ! Add 4 to the address of Y and display
02475F                  ! the resulting value.
DBG> EVALUATE .Y + 4    ! Add 4 to the value of Y and display
7                       ! the resulting value.
```

For all languages, to obtain the address of a variable, use the EVALUATE/ADDRESS command, as described in Section 3.1.10. The EVALUATE and EVALUATE/ADDRESS commands both display the address of an address expression when the language is set to BLISS.

3.1.5.2 Numeric Type Conversion by the Debugger

When evaluating language expressions involving numeric types of different precision, the debugger first converts lower-precision types to higher-precision types before performing the evaluation. In the following example, the debugger converts the integer 1 to the real 1.0 before doing the addition.

```
DBG> EVALUATE 1.5 + 1
2.5
DBG>
```

The basic rules are as follows. If integer and real types are mixed, the integer type is converted to the real type. If integer types of different sizes are mixed (for example, byte-integer and word-integer), the one with the smaller size is converted to the larger size. If real types of different sizes are mixed (for example, G_float and H_float), the one with the smaller size is converted to the larger size.

In general, the debugger allows more numeric type conversion than the programming language. In addition, the hardware type used for a debugger calculation (word, longword, G_float, and so on) may differ from that chosen by the compiler. Because the debugger is not as strongly typed or as precise as some languages, the evaluation of an expression by the EVALUATE command may differ from the result that would be calculated by compiler generated code and obtained with the EXAMINE command.

3.1.6 Address Expressions Compared to Language Expressions

Do not confuse address expressions with language expressions. An address expression specifies a *program location*, whereas a language expression specifies a *value*. In particular, the EXAMINE command expects an address expression as its parameter, and the EVALUATE command expects a language expression as its parameter. These points are illustrated in the next examples.

In the following example, the value 12 is deposited into the variable X. This is confirmed by the EXAMINE command. The EVALUATE command computes and displays the sum of the current *value* of X and the integer literal 6:

```
DBG> DEPOSIT X = 12
DBG> EXAMINE X
MOD3\X: 12
DBG> EVALUATE X + 6
18
DBG>
```

In the next example, the EXAMINE command displays the value currently stored at the virtual memory location that is 6 bytes beyond the *address* of X.

```
DBG> EXAMINE X + 6
MOD3\X+6: 274903
DBG>
```

In this case the location is not associated with a compiler generated type. Therefore, the debugger interprets and displays the value stored at that location in the type longword integer (see Section 3.1.4).

In the next example, the value of X + 6 (that is, 18) is deposited into the location that is 6 bytes beyond the address of X. This is confirmed by the last EXAMINE command.

```
DBG> EXAMINE X
MOD3\X: 12
DBG> DEPOSIT X + 6 = X + 6
DBG> EXAMINE X
MOD3\X: 12
DBG> EXAMINE X + 6
MOD3\X+6: 18
```

3.1.7 Specifying the Current, Previous, and Next Entity

When using the EXAMINE and DEPOSIT commands, you can use three special built-in symbols (address expressions) to refer quickly to the current, previous, and next data locations (logical entities). These are the period (.), the circumflex (^), and the RETURN key.

The period (.), when used by itself with an EXAMINE or DEPOSIT command, denotes the current entity — that is, the program location most recently referenced by an EXAMINE or DEPOSIT command. For example:

```
DBG> EXAMINE X
SIZE\X: 7
DBG> DEPOSIT . = 12
DBG> EXAMINE .
SIZE\X: 12
DBG>
```

The circumflex (^) and RETURN key denote, respectively, the previous and next logical data locations relative to the last EXAMINE or DEPOSIT command (the logical predecessor and successor, respectively). The circumflex and RETURN key are useful for referring to consecutive indexed components of an array. The following example illustrates the use of these operators with an array of integers, ARR:

```
DBG> EXAMINE ARR(5)      ! Examine element 5 of array ARR
MAIN\ARR(5): 448670
DBG> EXAMINE ^           ! Examine the previous element (4)
MAIN\ARR(4): 79280
DBG> EXAMINE [RET]       ! Examine the next element (5)
MAIN\ARR(5): 448670
DBG> EXAMINE [RET]       ! Examine the next element (6)
MAIN\ARR(6): 891236
DBG>
```

The debugger uses the type associated with the current entity to determine logical successors and predecessors.

You can also use the built-in symbols %CURLOC, %PREVLOC, and %NEXTLOC to achieve the same purpose as the period, circumflex, and RETURN key, respectively. These symbols are useful in command procedures and also if your program uses the circumflex for other purposes. Moreover, 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.

See Appendix D for more information on built-in symbols.

The previous example illustrates the use of the built-in symbols after referencing a symbolic name with the EXAMINE or DEPOSIT command. If you examine or deposit into a virtual memory address, that location may or may not be associated with a compiler generated type. When you reference a virtual memory address, the debugger uses the following convention to determine logical predecessors and successors:

- If the address has a symbolic name (the name of a variable, component of a structured variable, routine, and so on), the debugger uses the associated compiler generated type.
- If the address does not have a symbolic name, the debugger uses the type longword integer by default.

As the current entity is reset with new examine or deposit operations, the debugger associates each new location with a type in the manner indicated to determine logical successors and predecessors. This is illustrated in the next examples.

Assume that your program has declared three variables, ARY, FLT, and BTE:

- ARY is an array of three word integers (2 bytes each).
- FLT is an F_floating type (4 bytes).
- BTE is a byte integer (1 byte).

Assume that storage for these variables has been allocated at consecutive addresses in memory, starting with 1000. For example:

```
1000: ARY(1)
1002: ARY(2)
1004: ARY(3)
1006: FLT
1010: BTE
1011: undefined
   .
   .
   .
```

Then, examining successive logical data locations would give the following results:

```
DBG> EXAMINE 1000          ! Examine ARY(1), associated with 1000.
MOD3\ARY(1):  13           ! Current entity is now ARY(1).
DBG> EXAMINE [RET]         ! Examine next location, ARY(2),
MOD3\ARY(2):  7            ! using type of ARY(1) as reference.
DBG> EXAMINE [RET]         ! Examine next location, ARY(3).
MOD3\ARY(3):  19           ! Current entity is now ARY(3).
DBG> EXAMINE [RET]         ! Examine entity at 1006 (FLT).
MOD3\FLT:  1.9117807E+07   ! Current entity is now FLT.
DBG> EXAMINE [RET]         ! Examine entity at 1010 (BTE).
MOD3\BTE:  43              ! Current entity is now BTE.
DBG> EXAMINE [RET]         ! Examine entity at 1011 (undefined).
1011: 17694732             ! Interpret data as longword integer.
DBG>                       ! Location is not symbolized.
```

The same principles apply when you use type qualifiers with the EXAMINE and DEPOSIT commands (see Section 3.5.2). The type specified by the qualifier determines the data boundary of an entity and, therefore, any logical successors and predecessors.

3.1.8 Language Dependencies and the Current Language

The debugger enables you to set your debugging context to any one of several VAX-supported languages. The setting of the current language determines how the debugger parses and interprets the names, numbers, operators, and expressions you specify in debugger commands, and how it displays data.

By default, the current language is the language of the module containing the main program, and it is identified when you invoke the debugger. For example:

```
$ PASCAL/NOOPTIMIZE/DEBUG FORMS
$ LINK/DEBUG FORMS
$ RUN FORMS

                    VAX DEBUG Version 5.0

%DEBUG-I-INITIAL, language is PASCAL, module set to 'FORMS'
DBG>
```

When debugging modules whose code is written in other languages, you can use the SET LANGUAGE command to establish a new language dependent context. Section 8.3 highlights some important language differences. Appendix E identifies the operators and language constructs that are supported for each language (these are also identified if you type HELP LANGUAGE *language-name* at the debugger prompt). Be sure to consult the user's guide of your language documentation for details on debugger support for that language.

3.1.9 Specifying a Radix for Entering or Displaying Integer Data

The debugger can interpret and display integer data in any one of four radixes: decimal, hexadecimal, octal, and binary. The default radix is decimal for all languages except BLISS and MACRO, and it is hexadecimal for BLISS and MACRO.

You can control the radix for the following kinds of integer data:

- Data that you specify in address expressions or language expressions.

- Data that is displayed by the EVALUATE and EXAMINE commands.

You cannot control the radix for other kinds of integer data. For example, addresses are always displayed in hexadecimal radix in a SHOW CALLS display. Or, when specifying an integer *n* with various command qualifiers (/AFTER:*n*, /UP:*n*, and so on) you must use decimal radix.

The technique you use to control radix depends on your objective. To establish a new radix for all subsequent commands, use the SET RADIX command. For example:

```
DBG> SET RADIX HEXADECIMAL
DBG>
```

After this command is executed, all integer data that you enter in address or language expressions is interpreted as being hexadecimal. Also, all integer data displayed by EVALUATE and EXAMINE commands is given in hexadecimal radix.

The SHOW RADIX command identifies the *current radix* (which is either the default radix, or the radix last established by a SET RADIX command). For example:

```
DBG> SHOW RADIX
input radix: hexadecimal
output radix: hexadecimal
DBG>
```

The SHOW RADIX command identifies both the *input radix* (for data entry) and the *output radix* (for data display). The SET RADIX command qualifiers /INPUT and /OUTPUT enable you to specify different radixes for data entry and display. See the command dictionary for additional information about the SET RADIX command.

Use the CANCEL RADIX command to restore the default radix.

The examples that follow show several techniques for displaying or entering integer data in another radix without changing the current radix.

To convert some integer data to another radix without changing the current radix, use the EVALUATE command with a radix qualifier (/BINARY, /DECIMAL, /HEXADECIMAL, /OCTAL). For example:

```
DBG> SHOW RADIX
input radix: decimal
output radix: decimal
DBG> EVALUATE 18 + 5
23                          ! 23 is decimal integer.
DBG> EVALUATE/HEX 18 + 5
00000017                    ! 17 is hexadecimal integer.
DBG>
```

The radix qualifiers do not affect the radix for data entry.

To display the current value of an integer variable (or the contents of a program location that has an integer type) in another radix, use the EXAMINE command with a radix qualifier. For example:

```
DBG> EXAMINE X
MOD4\X: 4398                ! 4398 is a decimal integer.
DBG> EXAMINE/OCTAL .        ! X is the current entity.
MOD4\X: 00000010456         ! 10456 is an octal integer.
DBG>
```

To *enter* one or more integer literals in another radix without changing the current radix, use one of the radix built-in symbols %BIN, %DEC, %HEX, or %OCT. A radix built-in symbol directs the debugger to treat an integer literal that follows (or all numeric literals in a parenthesized expression that follows) as a binary, decimal, hexadecimal, or octal number, respectively. These symbols do not affect the radix for data display. For example:

```
DBG> SHOW RADIX
input radix: decimal
output radix: decimal
DBG> EVAL %BIN 10           ! Evaluate the binary integer 10.
2                           ! 2 is a decimal integer.
DBG> EVAL %HEX (10 + 10)    ! Evaluate the hexadecimal integer 20.
32                          ! 32 is a decimal integer.
DBG> EVAL %HEX 20 + 33      ! Treat 20 as hexadecimal, 33 as decimal.
65                          ! 65 is a decimal integer.
DBG> EVAL/HEX %OCT 4672     ! Treat 4672 as octal and display in hex.
000009BA                    ! 9BA is a hexadecimal number.
DBG> EXAMINE X + %DEC 12    ! Examine the location 12 decimal bytes
MOD3\X+12:  493847          ! beyond the address of X.
DBG> DEPOS J = %OCT 7777777 ! Deposit an octal value.
DBG> EXAMINE .              ! Display that value in decimal radix.
MOD3\J:  2097151
DBG> EXAMINE/OCTAL .        ! Display that value in octal radix.
MOD3\J:  00007777777
DBG> EXAMINE %HEX 0A34D     ! Examine location A34D, hexadecimal.
SHARE$LIBRTL+4941:  344938193   ! 344938193 is a decimal integer.
DBG>
```

NOTE: **When specifying a hexadecimal integer that starts with a letter rather than a number (for example, A34D in the last example), add a leading "0". Otherwise, the debugger tries to interpret the integer as a symbol declared in your program.**

See Appendix D for more examples showing the use of the radix built-in symbols.

3.1.10 Obtaining and Symbolizing Virtual Memory Addresses

Use the EVALUATE/ADDRESS command to determine the virtual memory address or the register name associated with a symbolic address expression, such as a variable name, line number, routine name, or label. For example:

```
DBG> EVALUATE/ADDRESS X       ! A variable name
2476
DBG> EVALUATE/ADDRESS SWAP    ! A routine name
1536
DBG> EVALUATE/ADDRESS %LINE 26
1629
DBG>
```

The address is displayed in the current radix (as defined in Section 3.1.9). You can specify a radix qualifier to display the address in another radix. For example:

```
DBG> EVALUATE/ADDRESS/HEX X
000009AC
DBG>
```

If a variable is associated with a register instead of a virtual memory address, the EVALUATE/ADDRESS command displays the name of the register, regardless of whether a radix qualifier is used. The following command indicates that variable K (a nonstatic variable) is associated with register R2:

```
DBG> EVALUATE/ADDRESS K
%R2
```

The command SYMBOLIZE does the reverse of EVALUATE/ADDRESS. It converts a virtual memory address or a register name into its symbolic representation (including its path name) if such a representation is possible (Chapter 4 explains how to control symbolization). For example, the following command shows that variable K is associated with register R2:

```
DBG> SYMBOLIZE %R2
address MOD3\%R2:
    MOD3\K
DBG>
```

By default, symbolic mode is in effect (SET MODE SYMBOLIC). Therefore the debugger displays all addresses symbolically, if symbols are available for the addresses. For example, if you specify a numeric address with the EXAMINE command, the address is displayed in symbolic form if symbolic information is available:

```
DBG> EVALUATE/ADDRESS X
2476
DBG> EXAMINE 2476
MOD3\X:  16
```

However, if you specify a register that is associated with a variable, the EXAMINE command does not convert the register name to the variable name. For example:

```
DBG> EVALUATE/ADDRESS K
%R2
DBG> EXAMINE %R2
MOD3\%R2:  78
```

By entering the command SET MODE NOSYMBOLIC, you disable symbolic mode and cause the debugger to display numeric addresses rather than their symbolic names. When symbolization is disabled, the debugger may process commands somewhat faster because it does not need to convert numbers to names. The EXAMINE command has a /[NO]SYMBOLIC qualifier that enables you to control symbolization for a single EXAMINE command. For example:

```
DBG> EVALUATE/ADDRESS Y
512
DBG> EXAMINE 512
MOD3\Y:  28
DBG> EXAMINE/NOSYMBOLIC 512
512:  28
DBG>
```

Symbolic mode also affects the display of instructions. For example:

```
DBG> EXAMINE/INSTRUCTION .%PC
MOD5\%LINE 14+2: MOVAL   L^MOD4\X,R11
DBG> EXAMINE/NOSYMBOL/INSTRUCTION .%PC
1538:   MOVAL   L^1080,R11
DBG>
```

3.2 Examining and Depositing into Variables

The examples in this section illustrate how to use the EXAMINE and DEPOSIT commands with variables.

Languages differ in the types of variables they use, the names for these types, and the degree to which different types can be intermixed in expressions. The following generic types are discussed in this section.

- Scalars (such as integer, real, character, or boolean)
- Strings
- Arrays
- Records
- Pointers (access types)

The most important consideration when examining and manipulating variables in high-level language programs is that the debugger recognizes the names, syntax, type constraints, and scoping rules of the variables in your program. Therefore, when specifying a variable with the EXAMINE or DEPOSIT command, you use the same syntax that is used in the source code. The debugger processes and displays the data accordingly. Similarly, when assigning a value to a variable, the debugger follows the typing rules of the language. It issues a diagnostic message if you try to deposit an incompatible value. The examples in this section show some of these invalid operations and the resulting diagnostics.

When using the DEPOSIT command (or any other command), note the following behavior. If the debugger issues a diagnostic message with a severity level of I (informational), the command is still executed (the deposit is made in this case). The debugger aborts an illegal command line only when the severity level of the message is W (warning) or greater.

See your language documentation for additional examples and for information concerning any language features that are not supported by the debugger.

3.2.1 Scalar Types

The following examples illustrate use of the EXAMINE, DEPOSIT, and EVALUATE commands with some integer, real, and boolean types.

Examine a list of three integer variables:

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

Deposit an integer expression:

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

The debugger checks that a value to be assigned is compatible with the data type and dimensional constraints of the variable. The following example shows an attempt to 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
DBG>
```

If you try to mix numeric types (integer and real of varying precision) in a language expression, the debugger generally follows the rules of the language. Strongly typed languages do not allow much if any mixing. With some languages, you can deposit a real value into an integer variable. However, the real value is converted into an integer. For example:

```
DBG> DEPOSIT I = 12345
DBG> EXAMINE I
MOD3\I:  12345
DBG> DEPOSIT I = 123.45
DBG> EXAMINE I
MOD3\I:  123
DBG>
```

Note that, if numeric types are mixed in an expression, the debugger performs type conversion as discussed in Section 3.1.5.2. For example:

```
DBG> DEPOSIT Y = 2.356      ! Y is of type D_floating point.
DBG> EXAMINE Y
MOD3\Y: 2.35600000000000
DBG> EVALUATE Y + 3
5.35600000000000

DBG> DEPOSIT R = 5.35E3     ! R is of type F_floating point.
DBG> EXAMINE R
MOD3\R:  5350.000
DBG> EVALUATE R*50
   267500.0
DBG> DEPOSIT I = 22222
DBG> EVALUATE R/I
   0.2407524
DBG>
```

The next example shows some operations with boolean variables. The values TRUE and FALSE are assigned to the 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>
```

3.2.2 ASCII String Types

When displaying an ASCII string value, the debugger encloses it within quotation marks (") or apostrophes ('), depending on the language syntax. For example:

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

To deposit a string value (including a single character) into a string variable, you must enclose the value in quotation marks (") or apostrophes (')). For example:

```
DBG> DEPOSIT PART_NUMBER = "WG-7619.3-84"
DBG>
```

If the string has more ASCII characters (1 byte each) than can fit into the location denoted by the address expression, the debugger truncates the extra characters from the right and issues the following message:

```
%DEBUG-I-ISTRTRU, string truncated at or near DEPOSIT
```

If the string has fewer characters, the debugger pads the remaining characters to the right of the string by inserting ASCII space characters.

3.2.3 Array Types

You can examine an entire array aggregate, a single indexed element, or a slice (a range of elements). But you can deposit into only one element at a time. The following examples show typical operations with arrays.

The following command displays the values of all the elements of the array variable ARRX, a one-dimensional array of integers:

```
DBG> EXAMINE ARRX
MOD3\ARRX
    (1):     42
    (2):     17
    (3):    278
    (4):     56
    (5):    113
    (6):    149
DBG>
```

The following command displays the value of element 4 of array ARRX (depending on the language, parentheses or brackets are used to denote indexed elements):

```
DBG> EXAMINE ARRX(4)
MOD3\ARRX(4):    56
DBG>
```

The following command displays the values of all the elements in a slice of ARRX. This slice consists of the range of elements from element 2 through element 5:

```
DBG> EXAMINE ARRX(2:5)
MOD3\ARRX
    (2):     17
    (3):    278
    (4):     56
    (5):    113
DBG>
```

In general, a range of values to be examined is denoted by two values separated by a colon (*value1*:*value2*). Depending on the language, two periods (..) may be used instead of a colon.

You can deposit a value to only a single array element at a time (you cannot deposit to an array slice or an entire array aggregate with a single DEPOSIT command). For example, the following command deposits the value 53 into element 2 of ARRX:

```
DBG> DEPOSIT ARRX(2) = 53
DBG>
```

The following command displays the values of all the elements of array REAL_ARRAY, 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>
```

The debugger issues a diagnostic message if you try to deposit to an index value that is out of bounds. For example:

```
DBG> DEPOSIT REAL_ARRAY(1,4) = 26.13
%DEBUG-I-SUBOUTBND, subscript 2 is out of bounds, value is 4, bounds are 1..3
DBG>
```

Note that, in the previous example the deposit operation was executed because the diagnostic message is of I level. This means that the value of some array element adjacent to (1,3), possibly (2,1) may have been affected by the out-of-bounds deposit operation.

To deposit the same value to several components of an array, you can use a looping command, such as FOR or REPEAT. For example, assign the value RED to elements 1 through 4 of the array COLOR_ARRAY:

```
DBG> FOR I = 1 TO 4 DO (DEPOSIT COLOR_ARRAY(I) = RED)
DBG>
```

You can also use the built-in symbols (.) and (^) and the RETURN key to step through array elements, as explained in Section 3.1.7.

3.2.4 Record Types

You can examine an entire record aggregate, a single record component, or several components. But you can deposit into only one component at a time. The following examples show typical operations with records.

The following command displays the values of all the components of the record variable PART:

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

The following command displays the value of component IN_STOCK of record PART (general syntax):

```
DBG> EXAMINE PART.IN_STOCK
INVENTORY\PART.IN_STOCK: 24
DBG>
```

The following command displays the value of the same record component, using COBOL syntax (the language must be set to COBOL):

```
DBG> EXAMINE IN_STOCK OF PART
INVENTORY\IN_STOCK of PART:
    IN_STOCK:  24
DBG>
```

The following command displays the values of two components of record PART:

```
DBG> EXAMINE PART.ITEM, PART.IN_STOCK
INVENTORY\PART.ITEM:      "WF-1247"
INVENTORY\PART.IN_STOCK:  24
DBG>
```

The following command deposits a value into record component IN_STOCK:

```
DBG> DEPOSIT PART.IN_STOCK = 17
DBG>
```

3.2.5 Pointer (Access) Types

You can examine the entity designated (pointed to) by a pointer variable and deposit a value into that entity. You can also examine a pointer variable.

For example, the following Pascal code declares a pointer variable *A* that designates a value of type real:

```
 . . .
TYPE
     T = ^REAL;
VAR
     A : T;
 . . .
```

The following command displays the value of the entity designated by the pointer variable *A*:

```
DBG> EXAMINE A^
MOD3\A^:  1.7
```

In the follo· ing example, the value 3.9 is deposited into the entity designated by *A*:

```
DBG> DEPOSIT A^ = 3.9
DBG> EXAMINE A^
MOD3\A^:  3.9
```

When you specify the name of a pointer variable with the EXAMINE command, the debugger displays the virtual memory address of the object it designates. For example:

```
DBG> EXAMINE/HEXADECIMAL A
SAMPLE\A: 0000B2A4
DBG>
```

3.3 Examining and Depositing VAX Instructions

The debugger recognizes address expressions that are associated with VAX assembly language instructions. This enables you to examine and deposit instructions using the same basic techniques as with variables.

When stepping through your program to examine or deposit instructions, you may find it convenient to first enter the following command. It sets the default step mode to stepping by instruction:

```
DBG> SET STEP INSTRUCTION
DBG>
```

There are other step modes that enable you to step to specific kinds of instructions (CALL, BRANCH, and so on).

3.3.1 Examining VAX Instructions

If you specify an address expression that is associated with an instruction in an EXAMINE command (for example, a line number), the debugger displays the first instruction at that location. You can then use the period (.), RETURN key, and circumflex character (^) to display the current, next, and previous instruction (logical entity), as described in Section 3.1.7. For example:

```
DBG> EXAMINE %LINE 12
MOD3\%LINE 12:      MOVL    (R11),B^16(R11)
DBG> EXAMINE RET
MODE\%LINE 12+4:    MOVL    S^#1,B^4(R11)   ! Next instruction.
DBG> EXAMINE RET
MOD3\%LINE 12+8:    TSTL    B^16(R11)       ! Next instruction.
DBG> EXAMINE ^
MODE\%LINE 12+4:    MOVL    S^#1,B^4(R11)   ! Previous instruction.
DBG>
```

Line numbers, routine names, and labels are symbolic address expressions that are associated with instructions. In addition, instructions may be stored at various other memory addresses and in certain registers during the execution of your program.

The program counter (PC) is the register that contains the address of the next instruction to be executed by your program. The command EXAMINE .%PC displays that instruction. The period (.), when used directly in front of an address expression, denotes the "contents of" operator — that is, the contents of the location designated by the address expression. Note the following distinction:

- EXAMINE %PC displays the current PC value, namely the *address of the next instruction* to be executed.
- EXAMINE .%PC displays the contents of that address, namely the *next instruction* to be executed by the program.

When you enter the command EXAMINE .%PC, you can control the amount of information displayed by using the /OPERANDS qualifier. For example:

```
DBG> EXAMINE .%PC
MOD3\%LINE 12:       MOVL    B^12(R11),R1
DBG> EXAMINE/OPERANDS .%PC
MOD3\%LINE 12:       MOVL    B^12(R11),R1
     B^12(R11)  MOD3\K (address 1196) contains 1
     R1         R1 contains 8
DBG> EXAMINE/OPERANDS=FULL .%PC
MOD3\%LINE 12:       MOVL    B^12(R11),R1
     B^12(R11)  R11 contains MOD3\N (address 1184), B^12(1184) evaluates to
                MOD3\K (address 1196), which contains 1
     R1         R1 contains 8
DBG>
```

Use the /OPERANDS qualifier only when examining the current PC instruction. The information may not be reliable if you specify other locations. The command SET MODE [NO]OPERANDS enables you to control the default behavior of the command EXAMINE .%PC.

As shown in the previous examples, the debugger knows whether an address expression is associated with an instruction. If it is, the EXAMINE command displays that instruction (you do not need to use the /INSTRUCTION qualifier). You use the /INSTRUCTION qualifier to display the contents of an arbitrary program location as a VAX instruction — that is, the command EXAMINE/INSTRUCTION causes the debugger to interpret and format the contents of *any* program location as a VAX instruction (see Section 3.5.2).

Note that, when you examine consecutive instructions in a MACRO program, the debugger may misinterpret data as instructions if storage for the data is allocated in the middle of a stream of instructions. The following example shows some MACRO code with two longwords of data storage allocated directly after the BRB instruction at line 7 (line numbers have been added to the example for clarity):

```
module TEST
     1:          .TITLE  TEST
     2:
     3: TEST$START::
     4:          .WORD   0
     5:
     6:          MOVL    #2,R2
     7:          BRB     LABEL_2
     8:
     9:          .LONG   ^X12345
    10:          .LONG   ^X14465
    11:
    12: LABEL_2:
    13:          MOVL    #5,R5
    14:
    15:          .END    TEST$START
```

The following examine command displays the instruction at the start of line 6:

```
DBG> EXAMINE %LINE 6
TEST\TEST$START\%LINE 6:  MOVL    S^#02,R2
```

The following examine command correctly interprets and displays the logical successor entity as an instruction, at line 7:

```
DBG> EXAMINE [RET]
TEST\TEST$START\%LINE 7:  BRB     TEST\TEST$START\LABEL_2
```

However, the following three examine commands incorrectly interpret the three logical successors as instructions:

```
DBG> EXAMINE [RET]
TEST\TEST$START\%LINE 7+2:        MULF3   S^#11.00000,S^#0.5625000,S^#0.5000000
DBG> EXAMINE [RET]
%DEBUG-W-ADDRESSMODE, instruction uses illegal or undefined addressing modes
TEST\TEST$START\%LINE 7+6:        MULD3   S^#0.5625000[R4],S^#0.5000000,@W^5505(R0)
DBG> EXAMINE [RET]
TEST$START+12:    HALT
```

3.3.2 Depositing VAX Instructions

When depositing a VAX instruction, use the following command format:

```
DEPOSIT/INSTRUCTION address-expression = "VAX instruction"
```

You must enclose the instruction in either quotation marks or apostrophes. You must also use the /INSTRUCTION qualifier with the DEPOSIT command, to indicate that the delimited string is an instruction and not an ASCII string. Or, if you plan to deposit several instructions, you can first enter the command SET TYPE/OVERRIDE INSTRUCTION (see Section 3.5.2). You then do not need to use the /INSTRUCTION qualifier on the DEPOSIT command.

VAX instructions occupy different numbers of bytes, depending on their operands. When depositing VAX instructions of arbitrary lengths into successive memory locations, use the logical successor operator (RETURN key) to establish the next unoccupied location where an instruction can be deposited. The following example illustrates the technique.

```
DBG> SET TYPE/OVERRIDE/INST         ! Set the default type to instruction.
DBG> DEPOSIT 730 = "MOVB #77, R1"   ! Deposit an instruction beginning at address 730.
DBG> EXAMINE .                      ! Examine the current entity to verify the deposit.
730:  MOVB  #77,R1
DBG> EXAMINE [RET]            ! Make the logical successor the new current entity.
734:  HALT
DBG> DEPOSIT . = "MOVB #66, R2"     ! Deposit the next instruction.
DBG> EXAMINE .                      ! Display and verify the deposit.
734:  MOVB  #66,R2
DBG>
```

When you *replace* an instruction, be sure that the new instruction, including operands, is the same length in bytes as the old instruction. If the new instruction is longer, you cannot deposit it without overwriting, and thereby destroying, the next instruction. If the new instruction occupies fewer bytes of memory than the old one, 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.

The following example illustrates how to replace an instruction with an instruction of equal length.

```
DBG> SET STEP INSTRUCTION          ! Step by instruction.
DBG> STEP
stepped to 1584: PUSHAL  (R11)
DBG> STEP
stepped to 1586: CALLS  #1,L^2224  ! Instruction to be replaced.
DBG> EXAMINE .%PC
1586: CALLS  #1,L^2224
DBG> EXAMINE [RET]                 ! Determine start of next
1593:  CALLS   #0,L^2216           ! instruction (1593).
DBG> DEPOSIT/INST 1586 = "CALLS  #2,L^2224"
                                   ! Deposit new instruction.
DBG> EXAMINE .                     ! Verify that instruction
1586:  CALLS   #2,L^2224           ! is deposited.
DBG> EXAMINE [RET]                 ! Verify that the next
1593:  CALLS   #0,L^2216           ! instruction is unchanged.
DBG>
```

3.4 Examining and Depositing Register Values

The VAX architecture provides 16 general registers, some of which are used for temporary address and data storage. When referencing a register in a debugger command, use the following built-in symbols (the register name preceded by a percent sign (%)):

Symbol	Description
%R0 . . . %R11	General purpose registers R0 . . . R11
%AP (R12)	Argument pointer
%FP (R13)	Frame pointer
%SP (R14)	Stack pointer
%PC (R15)	Program counter
%PSL	Processor status longword

You can omit the % prefix if your program has not declared a symbol with the same name.

You can examine the contents of all the registers. You can deposit values into all the registers except for SP. Use caution when depositing values into FP.

The following examples show how to examine and deposit into registers.

```
DBG> SHOW TYPE          ! Show type for locations without
type: long integer      ! a compiler generated type.
DBG> SHOW RADIX         ! Identify the current radix.
input radix: decimal
output radix: decimal
DBG> EXAMINE %R11       ! Display the value in R11.
MOD3\%R11:  1024
DBG> DEPOSIT %R11 = 444 ! Deposit a new value into R11
DBG> EXAMINE %R11       ! Check the new value
R11:  444
DBG> EXAMINE %SP        ! Display the value in the stack pointer.
0\%SP:  2147278720
DBG>
```

See Section 3.3.1 for specific information about the PC.

3.4.1 The Processor Status Longword (PSL)

The PSL is a 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) contain 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 cannot be altered by a user-mode program.

The following example shows how to examine the contents of the PSL:

```
DBG>EXAMINE %PSL
MOD3\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
DBG>
```

See the *VAX Architecture Handbook* for complete information on the PSL, including the values of the various bits.

You can also display the information in the PSL in other formats. For example:

```
DBG> EXAMINE/LONG/HEX PSL
MOD3\%PSL:        03C00010
DBG> EXAMINE/LONG/BIN PSL
MOD3\%PSL:        00000011 11000000 00000000 00010000
DBG>
```

The command EXAMINE/PSL displays the value at any location in PSL format. This is useful for examining saved PSLs on the stack.

To disable all conditions in the PSL, clear bits 0 through 15 with the following DEPOSIT command:

```
DBG>DEPOSIT/WORD PSL = 0
DBG>EXAMINE PSL
MOD3\PSL:
      CMP TP FPD IS CURMOD PRVMOD IPL DV FU IV T N Z V C
       0   0  0   0  USER   USER   0   0  0  0 0 0 0 0 0
DBG>
```

3.5 Specifying a Type When Examining and Depositing

The preceding sections explain how to use the EXAMINE and DEPOSIT commands with program locations that have a symbolic name and, therefore, are associated with a compiler generated type.

Section 3.5.1 describes how the debugger formats (types) data for program locations that do not have a symbolic name and explains how you can control the type for those locations.

Section 3.5.2 explains how to override the type associated with any program location, including a location that has a symbolic name.

3.5.1 Defining a Type for Locations Without a Symbolic Name

Program locations that do not have a symbolic name and, therefore, are not associated with a compiler generated type have the type longword integer by default. Section 3.1.4 explains how to examine and deposit into such locations using the default type.

The SET TYPE command enables you to change the default type. This is useful if you want to examine and display the contents of a location in another type, or if you want to deposit a value of some particular type into a location that is associated with another type. The possible type keywords are as follows:

ASCIC	CONDITION_VALUE	INSTRUCTION	QUADWORD
ASCID	D_FLOAT	LONGWORD	TYPE=(*type-expression*)
ASCII:n	DATE_TIME	OCTAWORD	WORD
ASCIW	FLOAT	PACKED	
ASCIZ	G_FLOAT	PSL	
BYTE	H_FLOAT	PSW	

For example, the following commands set the type for locations without a symbolic name to, respectively, byte integer, G_float, and ASCII with 6 bytes of ASCII data. Each successive SET TYPE command resets the type:

```
DBG> SET TYPE BYTE
DBG> SET TYPE G_FLOAT
DBG> SET TYPE ASCII:6
```

Note that the SET TYPE command, when used without the /OVERRIDE qualifier, does not affect the type for program locations that have a symbolic name (locations associated with a compiler generated type).

The SHOW TYPE command identifies the current type for locations without a symbolic name. To restore the default type for such locations, enter the command SET TYPE LONGWORD.

3.5.2 Overriding the Current Type

The SET TYPE/OVERRIDE command enables you to change the type associated with *any* program location, thereby overriding any compiler generated type. For example, after the following command is executed, an unqualified EXAMINE command displays the contents of only the first byte of the location specified and interprets the contents as byte integer data. An unqualified DEPOSIT command modifies only the first byte of the location specified and formats the data deposited as byte integer data.

```
DBG> SET TYPE/OVERRIDE BYTE
DBG>
```

To identify the current override type, enter the command SHOW TYPE/OVERRIDE. To cancel the current override type and restore the normal interpretation of locations that have a symbolic name, enter the command CANCEL TYPE/OVERRIDE.

Type qualifiers, used with the EXAMINE and DEPOSIT commands, enable you to override the type currently associated with a program location for the duration of a single EXAMINE or DEPOSIT command. The type qualifiers are as follows:

/ASCIC	/CONDITION_VALUE	/INSTRUCTION	/QUADWORD
/ASCID	/D_FLOAT	/LONGWORD	/TASK
/ASCII:n	/DATE_TIME	/OCTAWORD	/TYPE=(*type-expression*)
/ASCIW	/FLOAT	/PACKED	/WORD
/ASCIZ	/G_FLOAT	/PSL	
/BYTE	/H_FLOAT	/PSW	

These qualifiers override any previous SET TYPE or SET TYPE/OVERRIDE command as well as any compiler generated type.

When used with a type qualifier, the EXAMINE command displays the entity specified by the address expression in that type. For example:

```
DBG> EXAMINE %LINE 15               ! Display line 15 in compiler
MOD3\%LINE 15 :  MOVL  #1,B^44(R11) ! generated type: instruction.
DBG> EXAMINE/BYTE .                 ! Type is byte integer.
MOD3\%LINE 15 :  -48
DBG> EXAMINE/WORD .                 ! Type is word integer.
MOD3\%LINE 15 :  464
DBG> EXAMINE/LONG .                 ! Type is longword integer.
MOD3\%LINE 15 :  749404624
DBG> EXAMINE/QUAD .                 ! Type is quadword integer.
MOD3%LINE 15 :  +0130653502894178768
DBG> EXAMINE/FLOAT .                ! Type is F_floating.
MOD3%LINE 15 :   1.9117807E-38
DBG> EXAMINE/G_FLOAT .              ! Type is G_floating.
MOD3%LINE 15 :   1.509506018605227E-300
DBG> EXAMINE/INSTRUCTION .          ! Type is VAX instruction.
MOD3\%LINE 15 :  MOVL  #1,B^44(R11)
DBG> EXAMINE/ASCII .                ! Type is ASCII string.
MOD3\%LINE 15 :  ".."
DBG>
```

When used with a type qualifier, the DEPOSIT command deposits a value of that type into the location specified by the address expression, overriding the type associated with the address expression.

The remaining sections provide examples of specifying integer, string, and user-declared types with type qualifiers and the SET TYPE command.

3.5.2.1 Integer Types

The following examples illustrate the use of the EXAMINE and DEPOSIT commands with integer type qualifiers (/BYTE, /WORD, /LONGWORD). These qualifiers enable you to deposit a value of a particular integer type into an arbitrary program location.

```
DBG> SHOW TYPE              ! Show type for locations without
type:  long integer         ! a compiler generated type.
DBG> EVALU/ADDR .           ! Current location is 724.
724
DBG> DEPO/BYTE . = 1        ! Deposit the value 1 into one byte
                            ! of memory at address 724.
DBG> EXAM .                 ! By default, 4 bytes are examined.
724:  1280461057
DBG> EXAM/BYTE .            ! Examine one byte only.
724:  1
DBG> DEPO/WORD . = 2        ! Deposit the value 2 into first two
                            ! bytes (word) of current entity.
DBG> EXAM/WORD .            ! Examine a word of the current entity.
724:  2
DBG> DEPO/LONG 724 = 999    ! Deposit the value 999 into 4 bytes
                            ! (a longword) beginning at address 724.
DBG> EXAM/LONG 724          ! Examine 4 bytes (longword)
724:  999                   ! beginning at address 724.
DBG>
```

3.5.2.2 ASCII String Type

The following examples illustrate the use of the EXAMINE and DEPOSIT commands with the /ASCII:*n* type qualifier.

When used with the DEPOSIT command, this qualifier enables you to deposit an ASCII string of length *n* into an arbitrary program location. In the example, the location has a symbolic name (I) and, therefore, is associated with a compiler generated integer type. The command format is as follows:

```
DEPOSIT/ASCII:n address-expression = "ASCII string of length n"
```

The default value of *n* is 4 bytes.

```
DBG> DEPOSIT I = "abcde"      ! I has compiler generated integer type.
%DEBUG-W-INVNUMBER, invalid numeric string 'abcde'
                              ! So, cannot deposit string into I.
DBG> DEP/ASCII:5 I = "abcde"  ! /ASCII qualifier overrides integer
                              ! type to deposit 5 bytes of
                              ! ASCII data.
DBG> EXAMINE .                ! Display value of I in compiler
MOD3\I:  1146048327           ! generated integer type.
DBG> EXAM/ASCII:5 .           ! Display value of I as 5-byte
MOD3\I:  "abcde"              ! ASCII string.
DBG>
```

If you want to enter several DEPOSIT/ASCII commands, you can establish an override ASCII type with the SET TYPE/OVERRIDE command. Subsequent EXAMINE and DEPOSIT commands then have the effect of specifying the /ASCII qualifier with these commands. For example:

```
DBG> SET TYPE/OVER ASCII:5 ! Establish ASCII:5 as override type.
DBG> DEPOSIT I = "abcde"   ! Can now deposit 5-byte string into I.
DBG> EXAMINE I             ! Display value of I as 5-byte
MOD3\I:  "abcde"! ASCII string.)
DBG> CANCEL TYPE/OVERRIDE  ! Cancel ASCII override type.
DBG> EXAMINE I             ! Display I in compiler generated type.
MOD3\I:  1146048327
DBG>
```

3.5.2.3 User-Declared Types

The following examples illustrate the use of the EXAMINE and DEPOSIT commands with the /TYPE=(*type-expression*) qualifier. The qualifier enables you to specify a user-declared override type when examining or depositing.

For example, assume that a Pascal program contains the following code, which declares the enumeration type COLOR with the three values RED, GREEN, and BLUE:

```
. . .
TYPE
    COLOR = (RED,GREEN,BLUE);
. . .
```

During the debugging session, the SHOW SYMBOL/TYPE command identifies the type COLOR as it is known to the debugger:

```
DBG> SHOW SYMBOL/TYPE COLOR
data MOD3\COLOR
    enumeration type (COLOR, 3 elements), size: 1 byte
DBG>
```

The next command displays the value at address 1000, which is not associated with a symbolic name. Therefore, the value *0* is displayed in the type longword integer, by default:

```
DBG> EXAMINE 1000
1000:   0
```

The next command displays the value at address 1000 in the type COLOR. The preceding SHOW SYMBOL/TYPE command indicates that each enumeration element is stored in 1 byte. Therefore, the debugger converts the first byte of the longword integer value 0 at address 1000 to the equivalent enumeration value, RED (the first of the three enumeration values):

```
DBG> EXAMINE/TYPE=(COLOR) 1000
1000:   RED
DBG>
```

The following DEPOSIT command deposits the value GREEN into address 1000 with the override type COLOR. The EXAMINE command displays the value at address 1000 in the default type, longword integer:

```
DBG> DEPOSIT/TYPE=(COLOR) 1000 = GREEN
DBG> EXAMINE 1000
1000:   1
DBG>
```

The following SET TYPE command establishes the type COLOR for locations, such as address 1000, that do not have a symbolic name. The EXAMINE command now displays the value at 1000 in the type COLOR:

```
DBG> SET TYPE TYPE=(COLOR)
DBG> EXAMINE 1000
1000:   GREEN
DBG>
```

4 Controlling Symbol Lookup

Symbolic debugging enables you to specify variable names, routine names, and so on, in debugger commands, precisely as they appear in your source code. You do not need to use virtual memory addresses or registers when referring to program locations (although you can, if you want). Also, the debugger knows about the conventions of the particular source language regarding things like data types, expressions, scope, and visibility of entities. Therefore, you can use symbols (names, operators, and so on) in the context that is appropriate to the source program and language.

In most cases, the way in which symbol information is passed from your source program to the debugger and is processed by the debugger is transparent to you. Certain cases might require some action, however. For example, when you try to examine a variable X in a debugger command, the debugger may display the following diagnostic message:

```
DBG> EXAMINE X
%DEBUG-E-NOSYMBOL, symbol 'X' is not in the symbol table
DBG>
```

Also, the debugger may display the following message if X is multiply defined — that is, if the same symbol X is defined (declared) in more than one module, routine, or other program unit:

```
DBG> EXAMINE X
%DEBUG-E-NOUNIQUE, symbol 'X' is not unique
DBG>
```

This chapter explains how to handle these and other situations related to symbol lookup:

- Controlling the level of symbolic information passed to the debugger when you compile and link your program.
- Module setting, the means by which symbolic information stored in your program's executable image is made available efficiently during a debugging session.
- Resolving multiply-defined symbols that the debugger cannot resolve automatically.
- Applying and extending these concepts when debugging shareable images.

Note that this chapter discusses only the symbols (typically address expressions) that are derived from your source program, for example:

- The names of various entities that you have declared in your source code, such as the names of variables, routines, labels, array elements, or record components.
- The names of modules (compilation units) and shareable images that are linked with your program.

- Elements that the debugger uses to identify source code — for example, the specifications of source files, and source line numbers as they appear in a listing file or when the debugger displays source code.

The following types of symbols are discussed in other chapters:

- The symbols you create during a debugging session with the DEFINE command are covered in Section 7.4.
- The debugger's built-in symbols, such as the period (.), %PC, and %SOURCE_SCOPE are tabulated in Appendix D and discussed throughout this manual in the appropriate context.

Also, see Section 3.1.10 for information on how to obtain the virtual memory addresses and register names associated with symbolic address expressions and how to symbolize program locations.

4.1 Controlling Symbol Information When Compiling and Linking

To take full advantage of symbolic debugging, you must compile and link your program with the /DEBUG qualifier. The following example illustrates these steps with a simple Pascal program, INVENTORY, that consists of two compilation units whose source code is in two separate files, FORMS.PAS and INVENTORY.PAS. INVENTORY is the main program unit:

```
$ PASCAL/NOOPTIMIZE/DEBUG FORMS, INVENTORY
$ LINK/DEBUG INVENTORY, FORMS
$
```

Note that the /NOOPTIMIZE qualifier is used with the compiler command (PASCAL, in this example). If the compiler optimizes code by default, it is best to disable this feature by specifying /NOOPTIMIZE (or the equivalent qualifier, if any, for your compiler). Otherwise, the resulting object code is optimized, possibly causing the contents of some program locations to be inconsistent with what you might expect from looking at the source code. (Section 8.1 describes some of the effects of optimization.)

The next sections describe how symbol information is created and passed to the debugger when compiling and linking.

4.1.1 Compiling

When you compile a source file using the /DEBUG qualifier, the compiler creates symbol records for the debug symbol table (DST records) and includes them in the object module being generated (such as the compiler output file FORMS.OBJ, in the previous example).

DST records provide not only the names of symbols but also all relevant information about their use. For example:

- Data types, ranges, and constraints associated with variables.
- Parameter names and parameter types associated with functions and procedures.
- Source line correlation records, which associate source lines with line numbers and source files.

4.1 Controlling Symbol Information When Compiling and Linking

Most compilers allow you to vary the amount of DST information put in an object module by specifying different options with the /DEBUG qualifier. Table 4–1 identifies the options for most compilers (refer to the documentation supplied with your compiler for complete information).

Table 4–1 Compiler Options for DST Symbol Information

Compiler Command	DST Information
/DEBUG[1]	Full
/DEBUG=TRACEBACK[2]	Traceback only (module names, routine names, and line numbers)
/NODEBUG[3]	None

[1] /DEBUG, /DEBUG=ALL, and /DEBUG=(SYMBOLS,TRACEBACK) are equivalent.

[2] /DEBUG=TRACEBACK and DEBUG=(NOSYMBOLS,TRACEBACK) are equivalent.

[3] /NODEBUG, /DEBUG=NONE, and /DEBUG=(NOSYMBOLS,NOTRACEBACK) are equivalent.

The TRACEBACK option is a default for most compilers. That is, if you omit the /DEBUG qualifier, most compilers assume /DEBUG=TRACEBACK. The TRACEBACK option enables the VMS traceback condition handler to translate virtual addresses into routine names and line numbers so that it can give a symbolic traceback if a run-time error has occurred. For example:

```
$ RUN INVENTORY
   .
   .
   .
%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
INVENTORY       INVENTORY         59       00000020    000005A1
$
```

Traceback information is also used by the debugger's SHOW CALLS command.

4.1.2 Local and Global Symbols

DST records contain information about all of the symbols that are defined in your program. These are either *local* or *global* symbols.

Typically, local symbols are symbols that are referenced only within the module where they are defined; global symbols are symbols such as routine names, procedure entry points, and global data names, that are defined in one module but referenced in other modules.

Compilers handle local and global symbols differently. Generally, the compiler resolves references to local symbols, and the linker resolves references to global symbols.

The distinction between local and global symbols is discussed in this chapter in connection with symbol lookup and with shareable images and universal symbols.

4.1.3 Linking

When you enter the command LINK/DEBUG to link object modules and produce an executable image, the linker performs several functions that affect debugging:

- It builds a debug symbol table (DST) from the DST records contained in the object modules being linked. The DST is the primary source of symbol information during a debugging session.
- It resolves references to global symbols and builds a global symbol table (GST). The GST duplicates some of the global symbol information already contained in the DST, but the GST is used by the debugger for symbol lookup under certain circumstances.
- It puts the DST and GST in the executable image.
- It sets flags in the executable image that cause the image activator to pass control to the debugger when you enter the RUN command.

Table 4–2 summarizes the level of DST and GST information passed to the debugger depending on the compiler or LINK command option. The compiler command qualifier controls the level of DST and GST information passed to the linker. The LINK command qualifier controls not only how much of that information is passed to the debugger but also how (or if) you can invoke the debugger.

Table 4–2 Effect of Compiler and Linker on DST and GST Symbol Information

Compiler Command Qualifier[1]	DST Data in Object Module	LINK Command Qualifier	Invoke Debugger	DST Data Passed to Debugger	GST Data Passed to Debugger
/DEBUG	Full	/DEBUG	RUN	Full	Full
/DEBUG=TRACE	Traceback only	/DEBUG	RUN	Traceback only	Full
/NODEBUG	None	/DEBUG	RUN	None	Full
/DEBUG	Full	/TRACE[2]	RUN/DEBUG	Traceback only	Only universal symbols[3]
/DEBUG=TRACE	Traceback only	/TRACE	RUN/DEBUG	Traceback only	Only universal symbols
/NODEBUG	None	/TRACE	RUN/DEBUG	None	Only universal symbols
/DEBUG	Full	/NOTRACE	Cannot		

[1]See Table 4–1 for additional information.

[2]LINK/TRACEBACK and LINK/NODEBUG are equivalent. This is the default for the LINK command.

[3]A universal symbol is a symbol that is defined in one image and referenced in another. A universal symbol must be defined as such at link time. See Section 4.4 for information on universal symbols and shareable images.

If you specify /NODEBUG with the compiler command and subsequently link and execute the image, the debugger issues the following message when it is invoked:

```
%DEBUG-I-NOLOCALS, image does not contain local symbols
```

The preceding message, which occurs whether you linked with the /TRACEBACK or /DEBUG qualifier, indicates that no DST has been created for that image. Therefore, you have access only to global symbols contained in the GST.

If you do not specify /DEBUG with the LINK command, the debugger issues the following message when it is invoked:

```
%DEBUG-I-NOGLOBALS, some or all global symbols not accessible
```

The preceding message indicates that the only global symbol information available during the debugging session is the following:

- Information about global symbols that is stored in the DST.
- Information about universal symbols that is stored in the GST.

These concepts are discussed in later sections. In particular, see Section 4.4 for additional information related to debugging shareable images.

4.1.4 Controlling Symbol Information in Debugged Images

Symbol records occupy space within the executable image. After you have debugged your program, you may want to link it again without using the /DEBUG qualifier, to make the executable image smaller. This creates an image with only traceback data in the DST.

The command LINK/NOTRACEBACK enables you to secure the contents of an image from users once it has been debugged. Use this command for images that are to be installed with privileges (see the *Guide to VMS System Security* and the *Guide to Setting Up a VMS System*). When you enter LINK/NOTRACEBACK, no symbolic information (including traceback data) is passed to the image. Moreover, the debugger cannot be invoked, either by the RUN/DEBUG command, or by a CTRL/Y—DEBUG sequence while the program is running.

4.2 Setting and Canceling Modules

The preceding sections explain how symbol information derived from your program is passed to the debugger when you compile and link the program. This section explains how that information is made available during a debugging session. The material covered will help you take appropriate action when the debugger is unable to locate a symbol you have specified in a command. For example:

```
DBG> EXAMINE X
%DEBUG-E-NOSYMBOL, symbol 'X' is not in the symbol table
DBG>
```

When you invoke the debugger, symbol information is contained in the DST and GST, within the executable image. The DST contains detailed information about local and global symbols. The GST duplicates some of the global symbol information contained in the DST.

Controlling Symbol Lookup

4.2 Setting and Canceling Modules

To facilitate symbol searches, the debugger loads symbol records from the DST and GST into a run-time symbol table (RST), which is structured for efficient symbol lookup. Unless a symbol record is in the RST, the debugger cannot recognize or use the symbol.

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

Symbol records are loaded into the RST as follows. At debugger startup, all GST records are loaded into the RST because global symbols must be accessible throughout the debugging session. Also, the debugger sets the module which contains the main program (the routine specified by the image transfer address, where execution is suspended at the start of a debugging session). You therefore have access to all global symbols and to any local symbols that should be visible within the main program.

As the program executes, whenever the debugger interrupts execution it sets the module containing the routine where execution is suspended. Therefore you can always reference the local symbols that should be visible at the current PC value (in addition to the global symbols). This default mode of operation is called "dynamic mode".

If you try to reference a local symbol that is defined in a module that has not been set, the debugger warns you that the symbol is not in the RST. You must then use the SET MODULE command to set the module containing that symbol explicitly. For example:

```
DBG> EXAMINE X
%DEBUG-E-NOSYMBOL, symbol 'X' is not in the symbol table
DBG> SET MODULE MOD3
DBG> EXAMINE X
MOD3\ROUT2\X: 26
DBG>
```

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

When a module is set, the debugger automatically allocates memory as needed by the RST. This may eventually slow down the debugger 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, thereby automatically releasing memory. Or you can disable dynamic mode by entering the command SET MODE NODYNAMIC. When dynamic mode is disabled, the debugger does not set modules automatically. Use the SHOW MODE command to determine whether dynamic mode is enabled or disabled.

Section 4.4 explains how to set images and modules when debugging shareable images.

4.3 Resolving Multiply-Defined Symbols

When you reference a multiply-defined symbol in a debugger command, the debugger may not be able to determine the particular declaration of the symbol that you intended. For example:

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

Also, the debugger may reference the declaration that is visible in the current scope, not the one you want.

To resolve such problems, you must specify a scope where the debugger should search for a particular declaration of the symbol. In the following example, the path name COUNTER\X uniquely specifies a particular declaration of X:

```
DBG> EXAMINE COUNTER\X
COUNTER\X: 14
DBG>
```

The next sections discuss scope concepts and explain how to work with multiply-defined symbols.

4.3.1 Scope and Symbol Lookup Conventions

You can specify symbols in debugger commands by using either a path name or the exact symbol.

If you specify a path name, the debugger looks for the symbol in the scope denoted by the path name. Section 4.3.2 explains the technique.

If you do not specify a path name, by default, the debugger searches the RST as follows (you can modify this default behavior with the SET SCOPE command, as explained in Section 4.3.3).

First, the debugger looks for symbols in the *PC scope* (also known as scope 0), according to the scope and visibility rules of the currently set language. This means that, typically, the debugger first looks within the block or routine surrounding the current PC value (where execution is currently suspended). If the symbol is not found, the debugger searches the nesting program unit, then its nesting unit, and so on. The precise manner, which depends on the language, guarantees that the correct declaration of a multiply-defined symbol is selected.

The debugger must enable you to reference symbols throughout your program, not just those that are visible in the PC scope as defined by the language. This is necessary so you can set breakpoints in arbitrary areas or examine arbitrary variables, and so on. Therefore, if the symbol is not visible in the PC scope, the debugger continues searching as follows.

After the PC scope, the debugger searches the scope of the calling routine (if any), then its caller, and so on. Symbolically, the complete *scope search list* is denoted *0,1,2, . . . ,n*, where scope 0 is the PC scope and *n* is the number of calls in the call stack. Within each scope, the debugger uses the visibility rules of the language to locate a symbol.

If the symbol is still not found, the debugger searches the rest of the RST (the other set modules and the GST). At this point the debugger does not attempt to resolve multiply-defined symbols. Instead, if more than one occurrence of the symbol is found, the debugger issues the "symbol not unique" message. For example:

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

4.3.2 Using SHOW SYMBOL and Path Names to Specify Symbols Uniquely

If the debugger indicates that a symbol reference is "not unique", use the SHOW SYMBOL command to obtain all possible path names for that symbol, then specify a path name to reference the symbol uniquely. For example:

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

DBG> SHOW SYMBOL COUNT
data MOD7\ROUT3\BLOCK1\COUNT
data MOD4\ROUT2\COUNT
routine MOD2\ROUT1\ROUT3\COUNT

DBG> EXAMINE MOD4\ROUT2\COUNT
MOD4\ROUT2\COUNT: 12
DBG>
```

The command SHOW SYMBOL COUNT lists all declarations of the symbol COUNT that exist in the RST. The first two declarations of COUNT are variables (data). The last declaration listed is a routine. Each declaration is shown with its path name prefix, which indicates the path (search scope) the debugger must follow to reach that particular declaration. For example, MOD4\ROUT2\COUNT denotes the declaration of the symbol COUNT in routine ROUT2 of module MOD4.

The path name format is as follows. The leftmost element of a path name identifies the module containing the symbol. Moving toward the right, the path name lists the successively nested routines and blocks that lead to the particular declaration of the symbol (which is the rightmost element).

Although the debugger always displays symbols with their path names, you need to use path names in debugger commands only to resolve an ambiguity.

The debugger looks up line numbers like any other symbols you specify (by default, it first looks in the module where execution is suspended). A common use of path names is for specifying a line number in an arbitrary module. For example:

```
DBG> SET BREAK QUEUE_MANAGER\%LINE 26
DBG>
```

Note that the SHOW SYMBOL command identifies global symbols twice, because global symbols are included both in the DST and in the GST. For example:

```
DBG> SHOW SYMBOL X
data ALPHA\X                ! global X
data ALPHA\BETA\X           ! local X
data X (global)             ! same as ALPHA\X
DBG>
```

4.3.2.1 Simplifying Path Names

Path names are often long. You can simplify the process of specifying path names in three ways:

- Abbreviate a path name.
- Define a brief symbol for a path name.
- Set a new search scope so you do not have to use a path name.

To abbreviate a path name, delete the names of nesting program units starting from the left, leaving enough of the path name to specify it uniquely. For example, ROUT3\COUNT is a valid abbreviated path name for the routine in the first example of Section 4.3.2.

To define a symbol for a path name, use the DEFINE command. For example:

```
DBG> DEFINE INTX = INT_STACK\CHECK\X
DBG> EXAMINE INTX
```

To set a new search scope, use the SET SCOPE command, which is described in Section 4.3.3.

4.3.2.2 Specifying Symbols in the Call Stack

You can use a numeric path name to specify the scope associated with a routine on the call stack (as identified in a SHOW CALLS display). The path name prefix "0\" denotes the PC scope, the path name prefix "1\" denotes scope 1 (the scope of the caller routine), and so on.

For example, the following commands display the current values of two distinct declarations of Y, which are visible in scope 0 and scope 2, respectively.

```
DBG> EXAMINE 0\Y
DBG> EXAMINE 2\Y
```

By default, the command EXAMINE Y signifies EXAMINE 0\Y.

4.3.2.3 Specifying Global Symbols

To specify a global symbol uniquely, use a backslash (\) as a prefix to the symbol. For example, the following command displays the value of the global symbol X:

```
DBG> EXAMINE \X
```

4.3.2.4 Specifying Routine Invocations

When a routine is called recursively, you may need to distinguish among several calls to the same routine, all of which generate new symbols with identical names.

You can include an invocation number in a path name to indicate a particular call to a routine. The number must be a nonnegative integer and must follow the name of the rightmost routine in the path name. 0 denotes the most recent invocation; 1 denotes the previous invocation, and so on. For example, if PROG calls COMPUTE and COMPUTE calls itself recursively, and each call creates a new variable SUM, the following command displays the value of SUM for the most recent call to COMPUTE:

```
DBG> EXAMINE PROG\COMPUTE 0\SUM
```

To refer to the variable SUM that was generated in the previous call to COMPUTE, you would express the path name with a 1 in place of the 0.

When you do not include an invocation number, the debugger assumes that the reference is to the most recent call to the routine (the default invocation number is 0).

4.3.3 Using SET SCOPE to Specify a Symbol Search Scope

By default, the debugger looks up symbols that you specify without a path name prefix by using the scope search list described in Section 4.3.1.

The SET SCOPE command enables you to establish a new scope for symbol lookup, so that you do not have to use a path name when referencing symbols in that scope.

In the following example, the SET SCOPE command establishes the path name MOD4\ROUT2 as the new scope for symbol lookup. Then, references to Y without a path name prefix specify the declaration of Y that is visible in the new scope.

```
DBG> EXAMINE Y
%DEBUG-E-NOUNIQUE, symbol 'Y' is not unique
DBG> SHOW SYMBOL Y
data MOD7\ROUT3\BLOCK1\Y
data MOD4\ROUT2\Y

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

After you have entered a SET SCOPE command, the debugger applies the path name you specified in the command to all references that are not individually qualified with path names.

You can specify numeric path names with SET SCOPE (see Section 4.3.2.2). For example, the following command sets the current scope to be three calls down from the PC scope.

```
DBG> SET SCOPE 3
DBG>
```

You can also define a scope search list to specify the order in which the debugger should search for symbols. For example, the following command causes the debugger to look for symbols first in the PC scope (scope 0) and then in the scope denoted by routine ROUT2 of module MOD4:

```
DBG> SET SCOPE 0, MOD4\ROUT2
DBG>
```

The debugger's default scope search list is equivalent to entering the following command (if it existed):

```
DBG> SET SCOPE 0,1,2,3, . . .  ,n
DBG>
```

Here the debugger searches successively down the call stack to find a symbol.

To display the current scope for symbol lookup, use the SHOW SCOPE command. To restore the default scope search list (see Section 4.3.1), use the CANCEL SCOPE command.

4.4 Debugging Shareable Images

By default, your program may be linked with several DIGITAL-supplied shareable images (for example, the run-time library image MTHRTL.EXE). This section explains how to extend the concepts described in the previous sections when debugging user-defined shareable images.

A shareable image is not intended to be directly executed. A shareable image must first be included as input in the linking of an executable image, and then the shareable image is loaded at run time when the executable image is run. You do not have to install a shareable image to debug it. Instead, you can debug your own private copy by assigning a logical name to it.

See the *VMS Linker Utility Manual* for detailed information on linking shareable images.

4.4.1 Compiling and Linking Shareable Images for Debugging

The basic steps in compiling and linking a shareable image for debugging are as follows:

1. Compile the source files for the main image and for the shareable image, using the /DEBUG qualifier.
2. Link the shareable image with the /SHAREABLE and /DEBUG command qualifiers, declaring any universal symbols for that image using the UNIVERSAL linker option. (A *universal symbol* is a symbol, for example a routine name, that is defined in a shareable image and referenced in another image.)
3. Link the shareable image against the main image, specifying the shareable image with the /SHAREABLE file qualifier as a linker option. Also specify the /DEBUG command qualifier.
4. Define a logical name to point to the local copy of the shareable image. You must specify the device and directory as well as the image name. Otherwise the VMS image activator looks for an image of that name in the system default shareable image library (SYS$LIBRARY:IMAGELIB.OLB).
5. Execute the main image to invoke the debugger. The shareable image is loaded at run time.

These steps are illustrated next with a simple example. In the example, MAIN.FOR and SUB1.FOR are the source files for the main image (the executable image that you specify with the RUN command); SHR1.FOR and SHR2.FOR are the source files for the shareable image that is to be debugged.

You compile the source files for each image as described in Section 4.1:

```
$ FORTRAN/NOOPT/DEBUG MAIN,SUB1
$ FORTRAN/NOOPT/DEBUG SHR1,SHR2
$
```

You then use the LINK command to create the shareable image, also specifying any universal symbols:

```
$ LINK/SHAREABLE/DEBUG SHR1,SHR2,SYS$INPUT:/OPTIONS
UNIVERSAL=SHR_ROUT CTRL/Z
$
```

In the preceding example,

- The /SHAREABLE command qualifier creates the shareable image SHR1.EXE from the object files SHR1.OBJ and SHR2.OBJ.
- The /OPTIONS qualifier appended to SYS$INPUT: enables you to specify the global symbol SHR_ROUT as a universal symbol interactively.
- The /DEBUG command qualifier builds a DST and a GST for SHR1.EXE and puts them in that image. The GST contains the universal symbol SHR_ROUT. Note that the linker puts universal symbols in the GST unless you specify LINK/NOTRACEBACK, because universal symbols must be global symbols as well.

You have now built the shareable image SHR1.EXE in your current default directory. Because SHR1.EXE is a shareable image, you do not execute it directly with the RUN command. Instead you link SHR1.EXE against the main (executable) image:

```
$ LINK/DEBUG MAIN,SUB1,SYS$INPUT:/OPTION
SHR1.EXE/SHAREABLE [CTRL/Z]
$
```

In the preceding example,

- The LINK command creates the executable image MAIN.EXE from MAIN.OBJ and SUB1.OBJ.
- The /DEBUG qualifier builds a DST and a GST for MAIN.EXE and puts them in that image.
- The /SHAREABLE qualifier appended to SHR1.EXE specifies that SHR1.EXE is to be linked against MAIN.EXE as a shareable image.

When you execute the resulting main image, MAIN.EXE, any shareable images linked against it are loaded at run time. However, by default the VMS image activator looks for shareable images in the system default shareable image library (SYS$LIBRARY:IMAGELIB.OLB). Therefore, you must define the logical name SHR1 to point to SHR1.EXE in your current default directory. Be sure to specify the device and directory:

```
$ DEFINE SHR1 SYS$DISK:[]SHR1.EXE
$
```

You can now invoke the debugger to debug both MAIN and SHR1 by entering the following command:

```
$ RUN MAIN
```

4.4.2 Accessing Symbols in Shareable Images

All the concepts covered in Sections 4.1 through 4.3 apply to the modules of a single image, namely the main (executable) image. This section provides additional information that is specific to debugging shareable images.

When you link shareable images for debugging as explained in the previous section, the linker builds a DST and a GST for each image. To conserve memory, the debugger builds an RST for an image only when that image is "set", either dynamically by the debugger or when you enter a SET IMAGE command.

The SHOW IMAGE command identifies all shareable images that are linked with your program, shows which images are set, and identifies the current image (see Section 4.4.2.2 for a definition of the current image). Only the main image is set initially when you invoke the debugger.

The following sections explain how the debugger sets images dynamically during program execution and how you can access symbols in arbitrary images independently of execution.

4.4.2.1 Accessing Symbols in the PC Scope (Dynamic Mode)

By default, dynamic mode is enabled. Therefore, whenever the debugger interrupts execution, the debugger sets the image and module where execution is suspended, if they are not already set (unless the image was linked with the /NOTRACEBACK qualifier).

Dynamic mode gives you the following access to symbols automatically:

- You can reference symbols defined in all set modules in the image where execution is suspended.
- You can reference symbols in the GST for that image, including any universal symbols defined for that image.
- By setting other modules in that image, you can reference any symbol defined in the image.

Once an image is set, it remains set until you cancel it with the CANCEL IMAGE command. If the debugger slows down as more images and modules are set, use the CANCEL IMAGE command. You can also enter the command SET MODE NODYNAMIC to disable dynamic mode.

4.4.2.2 Accessing Symbols in Arbitrary Images

The last image that you or the debugger sets is the *current image*. The current image is the debugging context for symbol lookup. Therefore, when using the following commands, you can reference only the symbols that are defined in the current image:

- (SET, SHOW, CANCEL) MODULE
- SHOW SYMBOL
- EXAMINE, DEPOSIT, EVALUATE
- TYPE
- (SET, CANCEL) BREAK
- (SET, CANCEL) TRACE
- (SET, CANCEL) WATCH

- DEFINE/ADDRESS, DEFINE/VALUE

However, note that the commands SHOW BREAK, SHOW TRACE, and SHOW WATCH identify any breakpoints, tracepoints, or watchpoints that have been set in *all* images.

To reference a symbol in another image, use the SET IMAGE command to make the specified image the current image, then use the SET MODULE command to set the module where that symbol is defined (the SET IMAGE command does not set any modules). The following example illustrates these concepts.

The sample program consists of a main image PROG1 and a shareable image SHR1. Assume that you have just invoked the debugger and that execution is suspended in image PROG1, within the main program. Now, suppose you want to set a breakpoint on routine ROUT2, which is defined in some module in image SHR1.

If you try to set a breakpoint on ROUT2, the debugger looks for ROUT2 in the current image, PROG1:

```
DBG> SET BREAK ROUT2
%DEBUG-E-NOSYMBOL, symbol 'ROUT2' is not in symbol table
DBG>
```

The SHOW IMAGE command shows that image SHR1 needs to be set:

```
DBG> SHOW IMAGE
 image name                      set    base address    end address

*PROG1                           yes    00000200        000009FF
 SHR1                            no     00001000        00001FFF

 total images: 2                 bytes allocated: 32856
DBG> SET IMAGE SHR1

DBG> SHOW IMAGE
 image name                      set    base address    end address

 PROG1                           yes    00000200        000009FF
*SHR1                            yes    00001000        00001FFF

 total images: 2                 bytes allocated: 41948
 DBG>
```

SHR1 is now set and is the current image. However, because the SET IMAGE command does not set any modules, you must set the module where ROUT2 is defined before you can set the breakpoint:

```
DBG> SET BREAK ROUT2
%DEBUG-E-NOSYMBOL, symbol 'ROUT2' is not in symbol table
DBG> SET MODULE/ALL
DBG> SET BREAK ROUT2
DBG> GO
break at routine ROUT2
10:     SUBROUTINE ROUT2(A,B)
DBG>
```

Now that you have set image SHR1 and all its modules and have reached the breakpoint at ROUT2, you can debug using the normal method (for example, step through the routine, examine variables, and so on).

Once you have set an image and set modules within that image, the image and modules remain set even if you establish a new current image. However, you have access to symbols only in the current image at any one time.

5 Controlling the Display of Source Code

The term *source code* refers to statements in a programming language as they appear in a source file. Each line of source code is also called a source line.

This chapter covers the following topics:

- How the debugger obtains information about source files and source lines.
- Directing the debugger to a source file that has been moved to another directory after it was compiled.
- Displaying source lines by specifying line numbers, address expressions, and search strings.
- Controlling the display of source code at debugger eventpoints (breakpoints, tracepoints, watchpoints) and after a STEP command has been executed.
- Using the SET MARGINS command to improve the display of source lines under certain circumstances.

The techniques described in this chapter apply to screen mode as well as line (noscreen) mode. Any difference in behavior between line mode and screen mode is identified in this chapter and in the command dictionary for the commands discussed. (Screen mode is described fully in Chapter 6.)

If your program has been optimized by the compiler, the code that is executing as you debug may not always match your source code. See Section 8.1 for information on that subject.

5.1 How the Debugger Obtains Source Code Information

When a compiler processes source files to generate object modules, it assigns a line number to each source line in sequential order. For most languages, each compilation unit (module) starts with line 1. For others like Ada, each source file, which may represent several compilation units, starts with line 1.

Line numbers appear in a source listing obtained with the /LIST compile-command qualifier. They also appear whenever the debugger displays source code, either in line mode or screen mode. Moreover, line numbers are used in several debugger commands (for example, TYPE, SET BREAK) to specify source lines.

The debugger displays source lines only if you have specified the /DEBUG command with both the compile command and the LINK command. Under these conditions, the symbol information created by the compiler and passed to the debug symbol table (DST) includes source-line correlation records. For a given module, source-line correlation records contain the full VMS file specification of each source file that contributes to that module. In addition, they associate source records (symbols, types, and so on) with source files and line numbers in the module.

5.2 Specifying the Location of Source Files

The debug symbol table (DST) contains the full VMS file specification of each source file when it was compiled. Thus, by default, the debugger expects a source file to be in the same directory it was in at compile time. If a source file is moved to a different directory after it is compiled, the debugger does not find it and displays a warning such as the following when attempting to display source code from that file:

```
%DEBUG-W-UNAOPNSRC, unable to open source file DISK:[JONES.WORK]PRG.FOR;2
```

In such cases, use the SET SOURCE command to direct the debugger to the new directory. The command may be applied to all source files for your program or to only the source files for specific modules.

For example, after the following command line is entered, the debugger looks for *all* source files in WORK$:[JONES.PROG3]:

```
DBG> SET SOURCE WORK$:[JONES.PROG3]
DBG>
```

You can specify a directory search list with the SET SOURCE command. For example, after the following command line is entered, the debugger looks for source files first in the current default directory ([]) and then in WORK$:[JONES.PROG3]:

```
DBG> SET SOURCE [], WORK$:[JONES.PROG3]
DBG>
```

If you want to apply the SET SOURCE command only to the source files for a given module, use the /MODULE=module-name qualifier and specify that module. For example, the following command line tells the debugger to find the source files for module SCREEN_IO in the directory DISK2:[SMITH.SHARE] (the search of source files for other modules is not affected by this command):

```
DBG> SET SOURCE/MODULE=SCREEN_IO DISK2:[SMITH.SHARE]
DBG>
```

In summary, 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.

Use the SHOW SOURCE command to display all source directory search lists currently in effect. The command displays the search lists for specific modules (as previously established by one or more SET SOURCE/MODULE commands) and the search list for all other modules (as previously established by a SET SOURCE command). For example:

```
DBG> SET SOURCE [PROJA],[PROJB],USER$:[PETER.PROJC]
DBG> SET SOURCE/MODULE=COBOLTEST [], DISK$2:[PROJD]
DBG> SHOW SOURCE
source directory search list for COBOLTEST:
        []
        DISK$2:[PROJD]
source directory search list for all other modules:
        [PROJA]
        [PROJB]
        USER$:[PETER.PROJC]
DBG>
```

If no SET SOURCE or SET SOURCE/MODULE command has been entered, the SHOW SOURCE command indicates that no search list is currently in effect.

Use the CANCEL SOURCE command to cancel the effect of a previous SET SOURCE command. Use the CANCEL SOURCE/MODULE command to cancel the effect of a previous SET SOURCE/MODULE command (specifying the same module name).

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.

See the description of the SET SOURCE command in the command dictionary for additional information about how the debugger locates source files that have been moved to another directory after compile time.

Opening a source file requires the use of an I/O channel, a limited system resource. Like the debugger, your program may need to open files. To ensure that the debugger does not use all available I/O channels and thus cause the program to fail, by default the debugger can keep a maximum of 5 source files open at one time. To specify a different limit, use the SET MAX_SOURCE_FILES command. For example, the following command line sets the limit to 7 source files:

```
DBG> SET MAXIMUM_SOURCE_FILES 7
DBG>
```

Note that the value specified limits only the number of source files that may be kept open *at any one time*. If the debugger reaches this limit, it closes a file in order to open another one. Note also that setting the limit 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 one time.

5.3 Displaying Source Code by Specifying Line Numbers

The TYPE command enables you to display source lines by specifying compiler-assigned line numbers, where each line number designates a line of source code.

For example, the following command displays line 160 and lines 22 through 24 of the module being debugged:

```
DBG> TYPE 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.
DBG>
```

You can display all the source lines of a module 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 module.

After displaying a source line, you can display the next line in that module by entering a TYPE command without a line number — that is, by entering a TYPE command and then pressing the RETURN key. For example:

```
DBG>TYPE 160
module COBOLTEST
   160: START-IT-PARA.
DBG>TYPE
module COBOLTEST
   161:          MOVE SC1 TO ESO.
DBG>
```

You can then display the next line and successive lines by entering the TYPE command repeatedly, in this way reading through your code one line at a time.

To display source lines in an arbitrary module of your program, specify the module name with the line numbers. Use standard path name notation — that is, first specify the module name, then a backslash (\), and finally the line numbers (or the range of line numbers), without intervening spaces. For example, the following command displays line 16 of module TEST:

```
DBG> TYPE TEST\16
```

If you do not specify a module name with the TYPE command, the debugger displays source lines for the module where execution is currently suspended, by default — that is, the module associated with the PC scope. If you have specified another scope with the SET SCOPE command the debugger displays source lines in the module associated with the specified scope.

In screen mode, the output of a TYPE command is directed at the current source display, not at an output or DO display. The source display shows the lines specified and any surrounding lines that fit in the display window (see Chapter 6).

5.4 Displaying Source Code by Specifying Address Expressions

The EXAMINE/SOURCE command enables you to display the source line corresponding to an address expression. The address expression must denote the address of a machine code instruction and, therefore, must be one of the following:

- A line number
- A label
- A routine name
- The virtual address of an instruction

You cannot specify a variable name with the EXAMINE/SOURCE command, because a variable name is associated with data, not with instructions.

When you use the EXAMINE/SOURCE command, the debugger evaluates the address expression to obtain a virtual address, determines which compiler-assigned line number corresponds to that address, and then displays the source line designated by the line number.

For example, the following command line displays the source line associated with the address (declaration) of routine SWAP:

```
DBG> EXAMINE/SOURCE SWAP
module MAIN
    47:  procedure SWAP(X,Y: in out INTEGER) is
DBG>
```

If you specify a line number that is not associated with an instruction, the debugger issues a diagnostic message. For example:

```
DBG> EXAMINE/SOURCE %LINE 6
%DEBUG-I-LINEINFO, no line 6, previous line is 5, next line is 8
%DEBUG-E-NOSYMBOL, symbol '%LINE 6' is not in the symbol table
DBG>
```

The command EXAMINE/SOURCE .%PC displays the source line corresponding to the current PC value (the line that is about to be executed). For example:

```
DBG> EXAMINE/SOURCE .%PC
module COBOLTEST
   162:         DISPLAY ES0.
DBG>
```

Note the use of the "contents-of" operator (.), which specifies the contents of the entity that follows the period. If you do not use the contents-of operator, the debugger tries to find a source line for the address of the PC rather than for the value stored in the PC:

```
DBG> EXAMINE/SOURCE %PC
!%DEBUG-W-NOSRCLIN, no source line for address 7FFF005C
DBG>
```

The same kind of warning is issued if you specify a valid address expression with EXAMINE/SOURCE, but the module that contains the address expression is not set.

The following example shows the use of a numeric path name (1\) to display the source line at the PC value one level down the call stack (at the call to the routine where execution is suspended):

```
DBG> EXAMINE/SOURCE .1\%PC
```

In screen mode, the output of an EXAMINE/SOURCE command is directed at the current source display, not at an output or DO display. The arrow in the source display points to the line associated with the address expression specified. The predefined source display SRC is an automatically updated display that executes the following command every time the debugger prompts for commands (see Chapter 6 and Appendix C):

```
EXAMINE/SOURCE .%SOURCE_SCOPE\%PC
```

5.5 Displaying Source Code by Searching for Strings

The SEARCH command enables you to display any source lines that contain an occurrence of a specified string.

The syntax of the SEARCH command is as follows:

SEARCH[/qualifier[, . . .]] [range] [string]

The range parameter may be a module name, a range of line numbers, or a combination of both. If you do not specify a module name, the debugger uses the current scope to find source lines, as with the TYPE command (see Section 5.3).

By default, the SEARCH command displays the source line that contains the first (next) occurrence of a string in a specified range (SEARCH/NEXT). The command SEARCH/ALL displays all source lines that contain an occurrence of a string in a specified range. For example, the following command line displays the source line that contains the first occurrence of the string "pro" in module SCREEN_IO:

```
DBG> SEARCH SCREEN_IO pro
```

The remaining examples use source lines from one COBOL module, in the current scope, to illustrate various aspects of the SEARCH command.

The following command line displays all source lines within lines 40 through 50 that contain an occurrence of the string "D".

```
DBG>SEARCH/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.
DBG>
```

Once you have found an occurrence of a string in a particular module, you can enter the SEARCH command with no parameters to display the source line containing the next occurrence of the same string in the same module. This is analogous to using the TYPE command without a parameter to display the next source line. For example:

```
DBG> SEARCH 42:50 D
module COBOLTEST
    42: 02      DN      COMP-2 VALUE -222222.333333.
DBG> SEARCH
module COBOLTEST
    47: 02      DR0     COMP-2 VALUE  0.1.
DBG>
```

By default, the debugger searches for a string as specified and does not interpret the context surrounding an occurrence of the string (this is the behavior of SEARCH/STRING). If you want to locate occurrences of a string that is an identifier in your program (for example, a variable name) and exclude other occurrences of the string, use the /IDENTIFIER qualifier. The command SEARCH/IDENTIFIER displays only those occurrences of the string that are bounded on either side by a character that cannot be part of an identifier in the current language.

The default qualifiers for the SEARCH command are /NEXT and /STRING. If you want to establish different default qualifiers, use the SET SEARCH command. For example, after the following command is executed, the SEARCH command behaves like SEARCH/IDENTIFIER:

```
DBG> SET SEARCH IDENTIFIER
DBG>
```

Use the SHOW SEARCH command to display the default qualifiers currently in effect for the SEARCH command. For example:

```
DBG> SHOW SEARCH
search settings: search for next occurrence, as an identifier
DBG>
```

5.6 Controlling Source Display After Stepping and at Eventpoints

By default, whenever the debugger interrupts the execution of your program, it displays the source line at which execution is suspended. The debugger interrupts execution following a STEP command and when an eventpoint is triggered. Eventpoints are breakpoints, tracepoints, and watchpoints.

When you enter a STEP command, by default the debugger displays the source line at which execution is suspended after the step. This is the next line to be executed if you enter a STEP or GO command. For example:

```
DBG> STEP
stepped to MAIN\%LINE 16
    16:         RANGE := 500;
DBG>
```

When an eventpoint is triggered, by default the debugger displays the source line at which execution is suspended. For example:

```
DBG> SET BREAK SWAP
DBG> GO
   .
   .
   .
break at MAIN\SWAP
    47: procedure SWAP(X,Y: in out INTEGER) is
DBG>
```

In the case of a breakpoint or tracepoint, the debugger displays the source line at the location of the eventpoint. In the case of a watchpoint, the debugger displays the source line corresponding to the instruction that caused the watchpoint to be triggered.

The SET STEP [NO]SOURCE command enables you to control the display of source code after a step and at eventpoints. SET STEP SOURCE, the default, enables source display. SET STEP NOSOURCE suppresses source display. For example:

```
DBG> SET STEP NOSOURCE
DBG> STEP
stepped to MAIN\%LINE 16
DBG> SET BREAK SWAP
DBG> GO
    .
    .
    .
break at MAIN\SWAP
DBG>
```

You can selectively override the effect of a SET STEP SOURCE command or a SET STEP NOSOURCE command by using the qualifiers /SOURCE and /NOSOURCE with the STEP, SET BREAK, SET TRACE, and SET WATCH commands.

The command STEP/SOURCE overrides the effect of the command SET STEP NOSOURCE, but only for the duration of that STEP command (similarly, STEP/NOSOURCE overrides the effect of SET STEP SOURCE for the duration of that STEP command). For example:

```
DBG> SET STEP NOSOURCE
DBG> STEP/SOURCE
stepped to MAIN\%LINE 16
    16:         RANGE := 500;
DBG>
```

The command SET BREAK/SOURCE overrides the effect of the command SET STEP NOSOURCE, but only for the eventpoint set with that SET BREAK command (similarly, SET BREAK/NOSOURCE overrides the effect of SET STEP SOURCE for the eventpoint set with that SET BREAK command). The same conventions apply to the SET TRACE and SET WATCH commands. For example:

```
DBG> SET STEP SOURCE
DBG> SET BREAK/NOSOURCE SWAP
DBG> GO
    .
    .
    .
break at MAIN\SWAP
DBG>
```

5.7 Setting Margins for Source Display

The SET MARGINS command enables you to specify the leftmost and rightmost source-line character positions at which to begin and end the display of a source line (respectively, the left and right margins). This is useful for controlling the display of source code when, for example, the code is deeply indented or long lines wrap at the right margin. In such cases, you can set the left margin to eliminate indented space in the source display, and you can decrease the right margin setting to truncate lines and prevent them from wrapping.

For example, the following command line sets the left margin to column 20 and the right margin to column 35.

```
DBG> SET MARGINS 20:35
DBG>
```

Subsequently, only that portion of the source code that is between columns 20 and 35 is displayed when you enter commands that display source lines (for example, TYPE, SEARCH, STEP). Use the SHOW MARGINS command to identify the current margin settings for the display of source lines.

Note that the SET MARGINS command affects only the display of source lines. It does not affect the display of other debugger output, as from an EXAMINE command.

The SET MARGINS command is useful mostly in line (noscreen) mode. In screen mode, the SET MARGINS command has no effect on the display of source lines in a source display, such as the predefined display SRC.

6 Using Screen Mode

Screen mode enables you to 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). In screen mode, to recreate the default layout of various windows, press the keypad-key sequence BLUE-MINUS (PF4 followed by the MINUS key (-)).

Screen mode output is best displayed on VT100, VT200, or VT300 series terminals and VAXstations. The larger screen of VAXstations is particularly suitable to using a number of displays for different purposes. You can use screen mode with VT52 terminals, but they are less suited to the formatted screen displays because they do not support the scrolling regions used in screen mode.

This chapter covers the following topics:

- Screen mode concepts and terminology used throughout the chapter.
- The predefined displays SRC, OUT, PROMPT, INST, and REG, which are automatically available when you enter screen mode.
- Scrolling, hiding, deleting, moving, and resizing a display.
- Creating a new display.
- Specifying a display window.
- The different kinds of displays and how to use them.
- Directing various types of debugger output to different displays by assigning display attributes.
- A sample display configuration that illustrates a possible use of screen mode.
- Saving the current state of your screen displays.
- Changing your terminal screen's height and width during a debugging session and the effect on display windows.

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.

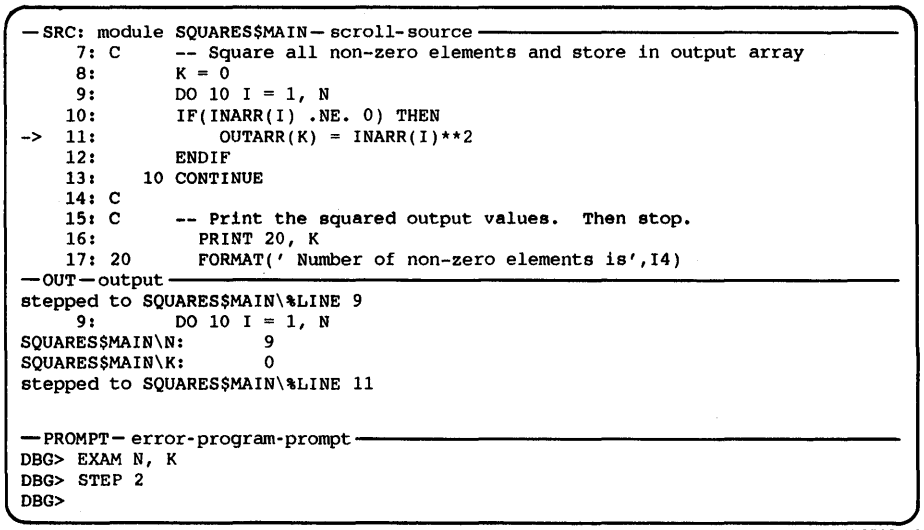
6.1 Concepts and Terminology

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.

Figure 6–1 is an example of screen mode that 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.

Figure 6–1 Default Screen Mode Display Configuration



ZK-6503-HC

- Display SRC is a source code display (it is displaying FORTRAN code in the example shown in Figure 6–1). 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, SQUARES$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.

Figure 6–1 is the default display configuration that is established when you first invoke screen mode. 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.

6.1 Concepts and Terminology

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 enable you to 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 can 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 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.

The contents of a display are generated in two ways. 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 enter 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 Figure 6–1 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.

6.2 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, 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 Figure 6–1. If, after rearranging displays and windows, you ever want to recreate this default configuration, press the keypad-key sequence BLUE-MINUS (PF4 followed by the MINUS (-) key).

Each of the predefined displays is discussed in the next sections.

6.2.1 The Predefined Source Display SRC

Note: See Chapter 5 for general information about source lines and how to control their display. See also the description of the SET SOURCE command in the command dictionary for related information.

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 source line where execution is suspended. 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 source line that corresponds to the new PC value.

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 routine where execution is suspended (because, for example, that 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 module where execution is suspended, 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.
```

Figure 6-2 illustrates this feature. In the source display, the arrow indicates that the PC value is at a call to 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 routine where execution is suspended and the call sequence leading to that routine.

Figure 6–2 Screen Mode Source Display When Source Code Is Not Available

```
—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>
```

ZK-6504-HC

6.2.2 The Predefined Output Display OUT

Figures 6–1 and 6-2 illustrate some typical debugger output in the predefined display OUT.

By default, 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). You can assign attributes to OUT so that it captures debugger input and diagnostics as well as normal output (see Section 6.7).

6.2.3 The Predefined Prompt Display PROMPT

The predefined display PROMPT is where the debugger prompts for input. Figures 6–1 and 6-2 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, by default, 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 such that it is hidden by PROMPT.

6.2.4 The Predefined Instruction Display INST

The predefined assembly-language instruction display INST displays the instruction stream of the routine being debugged (see Figure 6–3). The instructions displayed are decoded from the image being debugged.

Figure 6–3 Screen Mode Instruction Display

```
—INST: routine SQUARES$MAIN———————————————————————————————————————
          : TSTL    B^16(R11)
          : BLEQ    SQUARES$MAIN\%LINE 16
Line    10: MOVL    B^4(R11),R0
          : TSTL    W^-164(R11)[R0]
          : BEQL    SQUARES$MAIN\%LINE 13
->ne    11: MOVL    B^12(R11),R1
          : MOVL    B^4(R11),R0
          : MULL3   W^-164(R11)[R0],W^-164(R11)[R0],B^-84(R11)[R1]
Line    13: AOBLEQ  B^16(R11),B^4(R11),SQUARES$MAIN\%LINE 10
Line    16: PUSHAL  L^525
          : MNEGL   S^#1,-(SP)
—OUT—output———————————————————————————————————————————————————————
stepped to SQUARES$MAIN\%LINE 9
     9:         DO 10 I = 1, N
SQUARES$MAIN\N:       3
SQUARES$MAIN\K:       0
stepped to SQUARES$MAIN\%LINE 11
SQUARES$MAIN\I:       1
SQUARES$MAIN\K:       0
—PROMPT—error-program-prompt—————————————————————————————————————
DBG> STEP
DG> EXAMINE I,K
DBG>
```

ZK-6505-HC

This type of display is useful when debugging code that has been optimized. In such cases some of the code being executed may not match the source code that is shown in a source display. See Section 8.1 for information on the effects of optimization.

In display INST, the numbers to the left of the instructions are line numbers. The arrow points to the instruction at the current PC value. Whenever the debugger gains control from your program, the arrow position is updated and the display is centered around the current PC value.

By default, INST is marked as *removed* (see Section 6.3.2) from the display pasteboard and is not visible. You must use the DISPLAY command (or keypad keys 7 or GOLD-7) to show the INST display in such cases.

If you assign the instruction attribute to INST with the command SELECT /INSTRUCTION INST, then the output of an EXAMINE/INSTRUCTION command is directed to the instruction display.

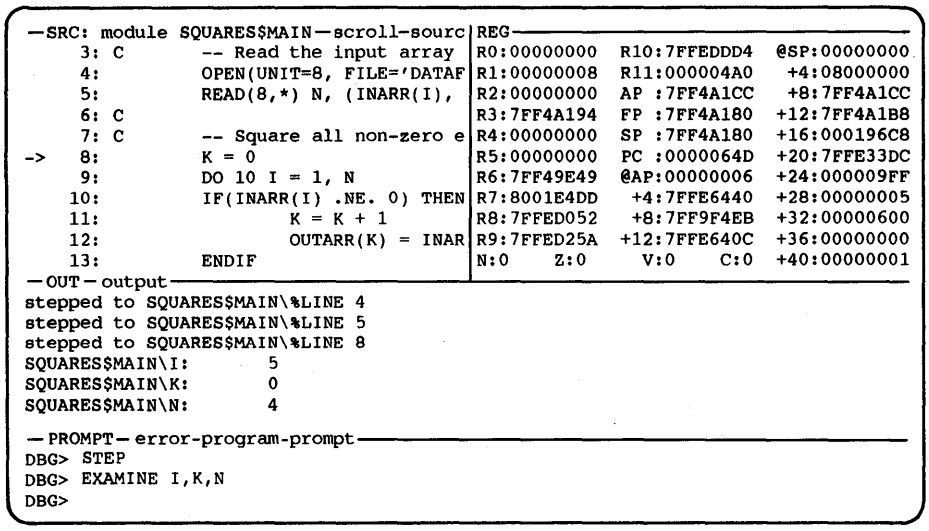
6.2.5 The Predefined Register Display REG

The predefined register display REG automatically shows the current values of all VAX machine registers (see Figure 6–4). 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 6.3.2) from the display pasteboard and is not visible. You must use the DISPLAY command (or the keypad key sequence GOLD-7) to show the REG display.

If the register window is made larger, the debugger fills the remaining space with information (in hexadecimal format) contained in the user stack (see Appendix C).

Figure 6–4 Screen Mode Register Display



ZK-6506-HC

6.3 Manipulating Existing Displays

This section explains how to

- Use the SELECT and SCROLL commands to scroll a display (Section 6.3.1).
- 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 6.3.2).

- Use the MOVE command to move a display across the screen (Section 6.3.3).
- Use the EXPAND command to expand or contract a display (Section 6.3.4).

Note also that Sections 6.5 and 6.6 discuss more advanced techniques for modifying existing displays with the DISPLAY command — how to change the display window and the type of information displayed.

6.3.1 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 enables you to 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 is 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 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 display. 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).

6.3.2 Showing, Hiding, Removing, and Canceling a Display

The DISPLAY command is the most versatile command for manipulating existing displays. In its simplest form, the command 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
```

Pressing keypad key 9, which is bound to the command DISPLAY %NEXTDISP, enables you to achieve this effect conveniently. The built-in function %NEXTDISP signifies the next display in the display list (Appendix D identifies all screen-related built-in functions). Each time you press key 9, the next display in the list 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 hides 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 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 6.4.

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. Note also that the DISPLAY command accepts optional parameters that enable you to modify other characteristics of existing displays, namely the display window and the type of information displayed. The techniques are discussed in Sections 6.5 and 6.6.

6.3.3 Moving a Display Across the Screen

Use the MOVE command to 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).

6.3.4 Expanding or Contracting a Display

Use the EXPAND command to 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.

6.4 Creating a New Display

To create a new screen display, use the SET DISPLAY command. The basic syntax is as follows:

```
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 6.5 explains the options 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 6.6 explains the options 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.

6.5 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, occludes 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 expands or contracts in proportion to the new screen height or width.

6.5.1 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.

6.5.2 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> DISPLAY ZIP AT RH1
```

The SHOW WINDOW command identifies all predefined window definitions, as well as those you create with the SET WINDOW command.

6.5.3 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.

6.6 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 you use 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 you use 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 6.2).

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 enter 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 6.7. 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 are paged in as necessary as you scroll through the module. In the case of an instruction display, the instructions are 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 holds. 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.

6.6.1 DO (command[; . . .]) Display Kind

A DO display is an automatically updated display. The commands in the command list are executed in the order listed 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)
```

6.6.2 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 updates the display.

6.6.3 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 is updated like a simple instruction display by an interactive EXAMINE/INSTRUCTION command (in addition to being updated by its built-in command).

6.6.4 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 6.7 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, because 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 now collects any debugger output that is not directed to another display. For example:

- The output of a SHOW CALLS command goes to OUT2.

- If no instruction display has been selected as the current instruction display, the output of an EXAMINE/INSTRUCTION command goes to OUT2.
- By default, debugger diagnostic messages are directed to the PROMPT display. They may be directed to OUT2 with the SELECT/ERROR command.

6.6.5 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.

See Appendix C for information on the behavior of register displays when expanded and contracted.

6.6.6 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 updates the display.

6.6.7 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 is updated like a simple source display by an interactive EXAMINE/SOURCE or TYPE command (in addition to being updated by its built-in command).

6.6.8 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 6.2.3).

6.7 Assigning Display Attributes

In screen mode, the output from commands you enter 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:

- SRC has the source 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" disappears from the top border of the predefined output display OUT and appears instead on display ZIP, and all debugger output formerly directed to OUT is now 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.
/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.

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 are logged in ZIP in the proper sequence as they occur, and ZIP is 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.

6.8 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 shows the source code of the module where execution is suspended.

The next command creates a source display named SRC2 in RH1 that shows the PC value at scope 1 (one level down the call stack, at the call to the routine where execution is suspended):

```
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)
```

Because 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.

6.9 Saving Displays and the Screen State

The SAVE command enables you to 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 enables you to 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 enables you to create a command procedure that may later be invoked during a debugging session to recreate 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.

6.10 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 VAXstation, 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 VAXstation.

In this example, assume you are using a VAXstation 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 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 enables you to control whether or not a display is dynamic. If a display is not dynamic, it does not change its window coordinates after you enter 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.

7 Additional Convenience Features

This chapter describes the following debugger convenience features not described elsewhere in Part I of this manual:

- Using debugger command procedures
- Using an initialization file for a debugging session
- Logging a debugging session into a file
- Defining symbols to represent commands, address expressions, or values
- Assigning debugger commands to function keys
- Using control structures to enter commands
- Calling arbitrary routines linked with your program

7.1 Using Debugger Command Procedures

A debugger command procedure is a sequence of commands contained in a file. It is an efficient way to enter precisely the same sequence of commands several times, for example, commands to set a series of breakpoints in your program. As with DCL command procedures, you execute a debugger command procedure by preceding its file specification with an at sign (@). The @ is the execute procedure command.

Debugger command procedures are especially useful when you regularly perform a number of standard set-up debugger commands, as specified in a debugger initialization file (see Section 7.2). You can also use a debugger log file as a command procedure (see Section 7.3).

7.1.1 Basic Conventions

The following is a sample debugger command procedure named BREAK7.COM:

```
! ***** Debugger Command Procedure BREAK7.COM *****
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
```

When you execute this command procedure with the execute procedure (@) command, the commands listed in the procedure are executed in the order they appear.

The rules for creating command procedures are listed in Sections 1 and 2 of the command dictionary.

You can pass parameters to a command procedure. See Section 7.1.2 for conventions on passing parameters.

You can enter the @ command like any other debugger command — that is, directly from the terminal, from within another command procedure, from within a DO clause in a command such as SET BREAK, or from within a DO clause in a screen display definition.

If you do not supply a full file specification with the @ command, the debugger assumes SYS$DISK:[]DEBUG.COM as the default file specification for command procedures. For example, you would enter the following command line to execute command procedure BREAK7.COM, located in your current default directory:

```
DBG> @BREAK7
```

The SET ATSIGN command enables you to change any or all fields of the default file specification, SYS$DISK:[]DEBUG.COM. The command SHOW ATSIGN identifies the default file specification for command procedures.

By default, commands read from a command procedure are not echoed. If you enter the command 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). Use the SHOW OUTPUT command to determine whether commands read from a command procedure will be echoed or not.

If the execution of a command in a command procedure results in a diagnostic of severity "warning" or higher, the command is aborted, but execution of the command procedure continues at the next command line.

7.1.2 Passing Parameters to Command Procedures

As with DCL command procedures, you can pass parameters to debugger command procedures. However, the technique is different in several respects.

Subject to the conventions described here, you can pass as many parameters as you want to a debugger command procedure. The parameters may be address expressions, commands, or value expressions in the current language. You must surround command strings in quotation marks ("), and you must separate parameters by commas (,).

A debugger command procedure to which you pass parameters must contain a DECLARE command line that binds each actual (passed) parameter to a formal parameter (a symbol) declared within the command procedure.

The DECLARE command is valid only within a command procedure. Its format is as follows:

DECLARE p-name:p-kind [,p-name:p-kind [,...]]

Each p-name:p-kind pair associates a formal parameter (p-name) with a parameter kind (p-kind). The valid p-kind keywords are as follows:

ADDRESS	Causes the actual parameter to be interpreted as an address expression.
COMMAND	Causes the actual parameter to be interpreted as a command.
VALUE	Causes the actual parameter to be interpreted as a value expression in the current language.

The following example illustrates what happens when a parameter is passed to a command procedure. The command DECLARE K:ADDRESS, within command procedure EXAM.COM, declares the formal parameter K. The actual parameter passed to EXAM.COM is interpreted as an address expression. The command EXAMINE K displays the value of that address expression. The command SET OUTPUT VERIFY causes the commands to echo when they are read by the debugger.

```
! ***** Debugger Command Procedure EXAM.COM *****
SET OUTPUT VERIFY
DECLARE K:ADDRESS
EXAMINE K
```

The next command line executes EXAM.COM, passing the actual parameter ARR4. Within EXAM.COM, ARR4 is interpreted as an address expression (an array variable, in this case).

```
DBG> @EXAM ARR4
%DEBUG-I-VERIFYIC, entering command procedure EXAM
 DECLARE K:ADDRESS
 EXAMINE K
PROG_8\ARR4
    (1):        18
    (2):        1
    (3):        0
    (4):        1
%DEBUG-I-VERIFYIC, exiting command procedure EXAM
```

Each p-name:p-kind pair specified by a DECLARE command binds one parameter. So, for instance, if you want to pass five parameters to a command procedure, you need five corresponding p-name:p-kind pairs. The pairs are always processed in the order in which you specify them.

For example, the next command procedure, EXAM_GO.COM accepts two parameters, an address expression (L) and a command string (M). The address expression is then examined and the command is executed:

```
! ***** Debugger Command Procedure EXAM_GO.COM *****
DECLARE L:ADDRESS, M:COMMAND
EXAMINE L; M
```

The following example shows how you could execute EXAM_GO.COM, passing a variable X to be examined and a command @DUMP.COM to be executed:

```
DBG> @EXAM_GO X, "@DUMP"
```

The %PARCNT built-in symbol, which can be used only within a command procedure, enables you to pass a variable number of parameters to a command procedure. The value of %PARCNT is the number of actual parameters passed to the command procedure.

The %PARCNT built-in symbol is illustrated in the following example. The command procedure, VAR.DBG, contains the following lines:

```
! ***** Debugger Command Procedure VAR.DBG *****
SET OUTPUT VERIFY
! Display the number of parameters passed:
EVALUATE %PARCNT
! Loop as needed to bind all passed parameters and obtain their values:
FOR I = 1 TO %PARCNT DO (DECLARE X:VALUE; EVALUATE X)
```

The following command line executes VAR.DBG, passing the parameters 12, 37, and 45:

```
DBG> @VAR.DBG 12,37,45
%DEBUG-I-VERIFYIC, entering command procedure VAR.DBG
 ! Display the number of parameters passed:
 EVALUATE %PARCNT
3
 ! Loop as needed to bind all passed parameters and obtain their values:
 FOR I = 1 TO %PARCNT DO (DECLARE X:VALUE; EVALUATE X)
12
37
45
%DEBUG-I-VERIFYIC, exiting command procedure VAR.DBG
```

When VAR.DBG is executed, %PARCNT has the value 3. Therefore, the FOR loop within VAR.DBG is repeated 3 times. The FOR loop causes the DECLARE command to bind each of the three actual parameters (starting with 12) to a new declaration of X. Each actual parameter is interpreted as a value expression in the current language, and the command EVALUATE X displays that value.

7.2 Using a Debugger Initialization File

A debugger initialization file is a command procedure, assigned the logical name DBG$INIT, that the debugger automatically executes at debugger start up. Every time you invoke the debugger, the commands contained in the file are automatically executed.

An initialization file contains any command lines you might always enter at the start of a debugging session to either tailor your debugging environment or control the execution of your program in a predetermined way from run to run.

For example, you might have a file DEBUG_START4.COM containing the following commands:

```
! ***** Debugger Initialization File DEBUG_START4.COM *****
! Log debugging session into default log file (SYS$DISK:[]DEBUG.LOG)
SET OUTPUT LOG
! Echo commands as they are read from command procedures:
SET OUTPUT VERIFY
! If source files are not in current default directory, use [SMITH.SHARE]
SET SOURCE [],[SMITH.SHARE]
! Set all modules:
SET MODULE/ALL
! Invoke screen mode:
SET MODE SCREEN
! Define the symbol SB as the command SET BREAK:
DEFINE/COMMAND SB = "SET BREAK"
! Assign the command SHOW MODULE * to keypad key 7:
DEFINE/KEY/TERMINATE KP7 "SHOW MODULE *"
```

To make this file a debugger initialization file, use the DCL command DEFINE. For example:

```
$ DEFINE DBG$INIT WORK:[JONES.DBGCOMFILES]DEBUG_START4.COM
```

7.3 Logging a Debugging Session into a File

A debugger log file maintains a history of a debugging session. During the debugging session, each command entered and the resulting debugger output are stored in the file.

The following is an example of a debugger log file.

```
SHOW OUTPUT
!noverify, terminal, noscreen_log, logging to DSK2:[JONES.P7]DEBUG.LOG;1
SET STEP NOSOURCE
SET TRACE %LINE 30
SET BREAK %LINE 60
SHOW TRACE
!tracepoint at PROG4\%LINE 30
GO
!trace at PROG4\%LINE 30
!break at PROG4\%LINE 60
    .
    .
    .
```

The DBG> prompt is not recorded, and the debugger output is commented out with exclamation points so the file can be used as a debugger command procedure without modification. Thus, if a lengthy debugging session is interrupted, you can execute the log file as you would any other debugger command procedure. Executing the log file restores the debugging session to the point at which it was previously terminated.

To create a debugger log file, use the command SET OUTPUT LOG. By default, the debugger writes the log to SYS$DISK:[]DEBUG.LOG. To name a debugger log file, use the SET LOG command. You can override any field of the default file specification. For example, after you enter the following commands, the debugger logs the session to the file [JONES.WORK2]MONITOR.LOG:

```
DBG> SET OUTPUT LOG
DBG> SET LOG [JONES.WORK2]MONITOR
```

You may want to enter the SET OUTPUT LOG command in your debugger initialization file (see Section 7.2).

The SHOW LOG command reports whether the debugger is writing to a log file and identifies the current log file. The SHOW OUTPUT command identifies all current output options.

If you are debugging in screen mode, the SET OUTPUT SCREEN_LOG command enables you to log the screen contents as the screen is updated. To use this command, you must already be logging your debugging session — that is, the command SET OUTPUT SCREEN_LOG is valid only after you have entered the command SET OUTPUT LOG. Note that using SET OUTPUT SCREEN_LOG is not desirable for a long debugging session, because storing screen information in this manner results in a big log file. For other techniques on saving screen-mode information, see also the descriptions of the commands SAVE and EXTRACT in Chapter 6 and in the command dictionary.

If you plan to use a log file as a command procedure, you should first enter the command SET OUTPUT VERIFY so that debugger commands are echoed as they are read.

7.4 Defining Symbols for Commands, Address Expressions, and Values

The DEFINE command enables you to create a symbol for a lengthy or often-repeated command sequence or address expression and to store the value of a language expression in a symbol.

You specify the kind of symbol you want to define by the command qualifier you use with the DEFINE command (/COMMAND, /ADDRESS, or /VALUE). The default qualifier is /ADDRESS. If you plan to enter 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). The SHOW DEFINE command identifies the default qualifier currently in effect.

Use the SHOW SYMBOL/DEFINED command to identify symbols you have defined with the DEFINE command. Note that the SHOW SYMBOL command without the /DEFINED qualifier identifies only the symbols that are defined in your program, such as the names of routines and variables.

Use the DELETE command to DELETE symbol definitions created with the DEFINE command.

When defining a symbol within a command procedure, use the /LOCAL qualifier to confine the symbol definition to that command procedure.

7.4.1 Defining Symbols for Commands

Use the DEFINE/COMMAND command to equate one or more commands (actually, strings) to a shorter symbol. The basic syntax is illustrated in the following example.

```
DBG> DEFINE/COMMAND SB = "SET BREAK"
DBG> SB PARSER
```

In the example, the DEFINE/COMMAND command equates the symbol SB to the string SET BREAK (note the use of the quotation marks to delimit the command string). When the command line SB PARSER is executed, the debugger substitutes the string SET BREAK for the symbol SB and then executes the SET BREAK command.

In the following example, the DEFINE/COMMAND command equates the symbol BT to the string consisting of the command SHOW BREAK followed by the command SHOW TRACE (use semicolons to separate multiple command strings):

```
DBG> DEFINE/COMMAND BT = "SHOW BREAK;SHOW TRACE"
```

The SHOW SYMBOL/DEFINED command identifies the symbol BT as follows:

```
DBG> SHOW SYM/DEFINED BT
defined BT
    bound to: "SHOW BREAK;SHOW TRACE"
    was defined /command
```

To define complex commands, you may need to use command procedures with parameters (see Section 7.1.2 for information on passing parameters to command procedures). For example:

```
DBG> DEFINE/COMMAND DUMP =  "@DUMP_PROG2.COM"
```

7.4.2 Defining Symbols for Address Expressions

Use the DEFINE/ADDRESS command to equate an address expression to a symbol. Although /ADDRESS is the default qualifier for the DEFINE command, it is used in the following examples for emphasis.

In the following example, the symbol B1 is equated to the address of line 378; the command SET BREAK B1 then sets a breakpoint on line 378.

```
DBG> DEFINE/ADDRESS B1 = %LINE 378
DBG> SET BREAK B1
```

The DEFINE/ADDRESS command is useful when you need to specify a long path name repeatedly to reference the name of a multiply-defined variable or routine. In the next example, the symbol UX is equated to the path name SCREEN_IO\UPDATE\X; the abbreviated command line EXAMINE UX can then be used to obtain the value of X in routine UPDATE of module SCREEN_IO.

```
DBG> DEFINE UX = SCREEN_IO\UPDATE\X
DBG> EXAMINE UX
```

7.4.3 Defining Symbols for Values

Use the DEFINE/VALUE command to equate the current value of a language expression to a symbol (the current value is the value at the time the DEFINE/VALUE command was entered).

The following example illustrates how the DEFINE/VALUE command may be used to count the number of calls to a routine.

```
DBG> DEFINE/VALUE COUNT = 0
DBG> SET TRACE/SILENT ROUT DO (DEFINE/VALUE COUNT = COUNT + 1)
DBG> GO
    .
    .
    .
DBG> EVALUATE COUNT
14
```

In the example, the first DEFINE/VALUE command initializes the value of the symbol COUNT to 0. The SET TRACE command sets a silent tracepoint on routine ROUT and (through the DO clause) increments the value of COUNT by 1 every time ROUT is called. After execution is resumed and eventually suspended, the EVALUATE command obtains the current value of COUNT (the number of times that ROUT was called).

7.5 Assigning Commands to Function Keys

To facilitate entering commonly used commands, the function keys on the keypad have predefined debugger functions that are established when you invoke the debugger. These predefined functions are identified in detail in Appendix B. You can modify the functions of the keypad keys to suit your individual needs. If you have a VT200- or VT300-series terminal or a workstation, you can also bind commands to the additional function keys on the LK201 keyboard.

The debugger commands DEFINE/KEY, SHOW KEY, and DELETE/KEY enable you to assign, identify, 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 enables you to use the predefined functions of the keypad keys.

If you want to use the keypad keys to enter numbers rather than debugger commands, enter the command SET MODE NOKEYPAD.

7.5.1 Basic Conventions

The debugger DEFINE/KEY command, which is similar to the DCL DEFINE/KEY command, enables you to assign a string to a function key. In the following example, the DEFINE/KEY command defines keypad key 7 to enter and execute the command SHOW MODULE *:

```
DBG> DEFINE/KEY/TERMINATE KP7 "SHOW MODULE *"
%DEBUG-I-DEFKEY, DEFAULT key KP7 has been defined
DBG>
```

The /TERMINATE qualifier indicates that pressing key 7 executes the command. You do not have to press RETURN after pressing key 7.

KP7 is the *key name* that you must use with the commands DEFINE/KEY, SHOW KEY, and DELETE/KEY. The valid key names that you can use with these commands are listed in the command dictionary for VT52 and VT100-series terminals and for LK201 keyboards (see the command descriptions).

The same function key can be assigned any number of definitions as long as each definition is associated with a different state. The predefined states (DEFAULT, GOLD, BLUE, and so on) are identified in Appendix B. In the preceding example, the informational message indicates that key 7 has been defined for the DEFAULT state (which is the default key state).

You can enter key definitions in a debugger initialization file (see Section 7.2) so that these definitions are available whenever you invoke the debugger.

To display a key definition in the current state, enter the command SHOW KEY. For example:

```
DBG> SHOW KEY KP7

DEFAULT keypad definitions:
  KP7 = "SHOW MODULE *" (echo,terminate,nolock)
DBG>
```

To display a key definition in a state other than the current state, specify that state with the /STATE qualifier when entering the SHOW KEY command. To see all key definitions in the current state, enter the command SHOW KEY/ALL.

To delete a key definition, use the DELETE/KEY command. To delete a key definition in a state other than the current state, specify that state with the /STATE qualifier. For example:

```
DBG> DELETE/KEY/STATE=GOLD KP7
%DEBUG-I-DELKEY, GOLD key KP7 has been deleted
```

7.5.2 More Advanced Techniques

This section illustrates more advanced techniques for defining keys, particularly techniques related to the use of state keys.

The following command line assigns the unterminated command string "SET BREAK %LINE" to keypad key 9, for the BLUE state.

```
DBG> DEFINE/KEY/IF_STATE=BLUE KP9 "SET BREAK %LINE"
```

The predefined DEFAULT key state is established by default. The predefined BLUE key state is established by pressing keypad key PF4. You would enter the command line assigned in the preceding example (SET BREAK %LINE . . .) by pressing key PF4 then key 9, then entering a line number, then pressing the RETURN key to terminate and process the command line.

The SET KEY command enables you to change the default state for key definitions. For example, after entering the command SET KEY/STATE=BLUE, you would not need to press PF4 to enter the command line in the previous example. Also, the SHOW KEY command would show key definitions in the BLUE state, by default, and the DELETE /KEY command would delete key definitions in the BLUE state, by default.

You can create additional key states. For example:

```
DBG> SET KEY/STATE=DEFAULT
DBG> DEFINE/KEY/SET_STATE=RED/LOCK_STATE F12 ""
```

In this example, the SET KEY command establishes DEFAULT as the current state. The DEFINE/KEY command makes key F12 (LK201 keyboard) a state key. As a result, 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 enter "RED" commands by pressing keys that have definitions associated with the RED state.

7.6 Using Control Structures to Enter Commands

The FOR, IF, REPEAT, and WHILE commands enable you to create looping and conditional constructs for entering debugger commands. The associated command EXITLOOP is used to exit a FOR, REPEAT, or WHILE loop.

See Section 3.1.5 and Section 8.3.2.2 for information about evaluating language expressions.

7.6.1 FOR Command

The FOR command executes a sequence of commands repetitively for a specified number of times. It has the following format:

```
FOR name=expression1 TO expression2 [BY expression3] DO(command[; . . . ])
```

For example, the following command line sets up a loop that initializes the first 10 elements of an array to zero:

```
DBG> FOR I = 1 TO 10 DO (DEPOSIT A(I) = 0)
```

7.6.2 IF Command

The IF command executes a sequence of commands if a language expression (boolean expression) is evaluated as TRUE. It has the following format:

IF boolean-expression THEN (command[; . . .]) [ELSE (command[; . . .])]

The following FORTRAN example sets up a condition that issues the command EXAMINE X2 if X1 is not equal to -9.9, and issues the command EXAMINE Y1 otherwise:

```
DBG> IF X1 .NE. -9.9 THEN (EXAMINE X2) ELSE (EXAMINE Y1)
```

The following Pascal example combines a FOR loop and a condition test. The STEP command is issued if X1 is not equal to -9.9. The test is made four times:

```
DBG> FOR COUNT = 1 TO 4 DO (IF X1 <> -9.9 THEN (STEP))
```

7.6.3 REPEAT Command

The REPEAT command executes a sequence of commands repetitively for a specified number of times. It has the following format:

REPEAT language-expression DO (command[; . . .])

For example, the following command line sets up a loop that issues a sequence of two commands (EXAMINE Y then STEP) 10 times:

```
DBG> REPEAT 10 DO (EXAMINE Y; STEP)
```

7.6.4 WHILE Command

The WHILE command executes a sequence of commands repetitively until the language expression (boolean expression) you have specified evaluates as FALSE. It has the following format:

WHILE boolean-expression DO (command[; . . .])

The following Pascal example sets up a loop that tests X1 and X2 repetitively and issues the two commands EXAMINE X2 and STEP if X2 is less than X1:

```
DBG> WHILE X2 < X1 DO (EX X2;STEP)
```

7.6.5 EXITLOOP Command

The EXITLOOP command exits one or more enclosing FOR, REPEAT, or WHILE loops. It has the following format:

EXITLOOP [n]

The integer *n* specifies the number of nested loops to exit from.

The following Pascal example sets up an endless loop that issues 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.

```
DBG> WHILE TRUE DO (STEP; IF X > 3 THEN EXITLOOP)
```

7.7 Calling Routines Linked with Your Program

The CALL command enables you to execute a routine independently of the normal execution of your program. It is one of the four debugger commands that can cause your program to execute (the others are GO, STEP, and EXIT).

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. Thus you can use the CALL command to execute routines for any purpose (for example, to debug a routine out of the context of program execution, invoke a run-time library procedure, execute a routine that dumps debugging information, and so on).

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.

The following example shows how you could use the CALL command to display some process statistics without having to include the necessary code in your program. The example consists of calls to run-time library routines that initialize a timer and counts (LIB$INIT_TIMER) and display the elapsed time and counts (LIB$SHOW_TIMER). (Note that the presence of the debugger affects the timings and counts):

```
DBG> SET MODULE SHARE$LIBRTL
DBG> CALL LIB$INIT_TIMER
value returned is 1
DBG> [enter various debugger commands]
 . . .
DBG> CALL LIB$SHOW_TIMER
 ELAPSED:    0 00:00:21.65  CPU: 0:14:00.21  BUFIO: 16  DIRIO: 0  FAULTS: 3
value returned is 1
```

In the previous example, the run-time library routines are in the shareable image LIBRTL. The SET MODULE command makes the universal symbols (routine names) in the shareable image visible in the main image. See the description of the /SHARE qualifier of the SHOW MODULE command (in the command dictionary) for more information on this subject.

The "value returned" message indicates the value returned in register R0 after the CALL command has been executed. By VMS convention, after a called routine has executed, register R0 contains the function return value (if the routine is a function) or the procedure completion status (if the routine is a procedure that returns a status value). If a called procedure does not return a status value or function value, the value in R0 may be meaningless, and the "value returned" message can be ignored.

The following example shows how to call LIB$SHOW_VM (also in LIBRTL) to display virtual memory statistics. (Again, note that the presence of the debugger affects the counts):

```
DBG> SET MODULE SHARE$LIBRTL
DBG> CALL LIB$SHOW_VM
 1785 calls to LIB$GET_VM, 284 calls to LIB$FREE_VM, 122216 bytes still allocated
value returned is 1
```

You can pass parameters to routines with the CALL command. See the description of the CALL command in the command dictionary for details and examples.

8 Debugging Special Cases

This chapter presents debugging techniques for special cases that are not covered elsewhere in Part I of this manual:

- Optimized code
- Screen-oriented programs
- Multilanguage programs
- Exceptions and condition handlers
- Exit handlers
- AST-driven programs

8.1 Debugging Optimized Code

By default, many compilers optimize the object code they produce so that the program executes faster. The net result is that the code that is executing as you debug may not match the source code as displayed in a screen-mode source display (see Chapter 6) or in a listing file. This section describes some typical situations.

To avoid the problems of debugging optimized code, many compilers allow you to specify the /NOOPTIMIZE (or equivalent) command qualifier at compile time. Specifying this qualifier inhibits most compiler optimization, thereby reducing discrepancies between source code and object code caused by optimization. However, this option is not always available to you, so you should read this section for information on how to debug optimized code.

When debugging optimized code, if you encounter a program segment where source and object code do not seem to match, you can inspect the object code itself by using a screen-mode instruction display (enter the command DISPLAY INST) or by entering EXAMINE/INSTRUCTION or STEP/INSTRUCTION commands. (Note that, in screen mode, pressing keypad key 7 produces the SRC and INST displays, side by side.) 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.

Debugging Special Cases

8.1 Debugging Optimized Code

8.1.1 Eliminated Variables

A compiler may optimize code by eliminating variables, either permanently or temporarily at various points during execution. If you try to examine a variable X that no longer is accessible because of optimization, the debugger may display one of the following messages:

```
%DEBUG-W-UNALLOCATED, entity X was not allocated in memory
   (was optimized away)

%DEBUG-W-NOVALATPC, entity X does not have a value at the current PC
   (was optimized away)
```

The following Pascal example shows how this could happen.

```
PROGRAM DOC(OUTPUT);
   VAR
      X,Y: INTEGER;
   BEGIN
      X := 5;
      Y := 2;
      WRITELN(X*Y);
   END.
```

If you compile this program with the /NOOPTIMIZE (or equivalent) qualifier, you obtain the following (normal) behavior when debugging:

```
$ PASCAL/DEBUG/NOOPTIMIZE DOC
$ LINK/DEBUG DOC
$ RUN DOC
   .
   .
   .
DBG> STEP
stepped to DOC\%LINE 5
     5:         X := 5;
DBG> STEP
stepped to DOC\%LINE 6
     6:         Y := 2;
DBG> STEP
stepped to DOC\%LINE 7
     7:         WRITELN(X*Y);
DBG> EXAMINE X,Y
DOC\X:  5
DOC\Y:  2
DBG>
```

If you compile the program with the /OPTIMIZE (or equivalent) qualifier, because the values of X and Y are not changed after the initial assignment, the compiler calculates X*Y, stores that value (10), and does not allocate storage for X or Y. Therefore, after you invoke the debugger, a STEP command takes you directly to line 7 rather than line 5. Moreover, you cannot examine X or Y:

```
$ PASCAL/DEBUG/OPTIMIZE DOC
$ LINK/DEBUG DOC
$ RUN DOC
   .
   .
   .
DBG> EXAMINE X,Y
%DEBUG-W-NOVALATPC, entity X does not have a value at the current PC
   (was optimized away)
DBG> STEP
stepped to DOC\%LINE 7
     7:         WRITELN(X*Y);
DBG>
```

To see exactly what values are being used in your optimized program, you can use the command EXAMINE/OPERAND .%PC to display the machine code at the current PC value, including the values and symbolization of all of the operands. For example, the following lines show the optimized code when the PC value is at the WRITELN statement:

```
DBG> STEP
stepped to DOC\%LINE 7
     7:         WRITELN(X*Y);
DBG> EXAMINE/OPERAND .%PC
DOC\%LINE 7:    PUSHL   S^#10
```

In contrast, the following lines show the unoptimized code at the WRITELN statement:

```
DBG> STEP
stepped to DOC\%LINE 7
     7:         WRITELN(X*Y);
DBG> EXAMINE/OPERAND .%PC
DOC\%LINE 7:    MOVL    S^#10,B^-4(FP)
     B^-4(FP)   2146279292 contains 62914576
```

8.1.2 Coding Order

Several methods of optimizing consist of performing operations in a sequence different from the sequence specified in the source code. Sometimes code is eliminated altogether.

As a result, the source code displayed by the debugger does not correspond exactly to the actual object code being executed. This is important to keep in mind, especially when using the STEP and EXAMINE/SOURCE commands.

To illustrate, the following example depicts a segment of source code from a FORTRAN program as it might appear on a compiler listing or in a screen-mode source display. 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, because 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            TEMP = 1/X
               DO 100 I=1,10
  6            A(I) = TEMP
  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 execute to line 5 by entering a STEP command, 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 enter 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 previous example, but by a number of other optimization methods as well.

8.1.3 Use of Registers

A compiler may determine that the value of an expression does not change between two given occurrences and may save the value in a register. In such cases, the compiler does not recompute the value for the next occurrence, but assumes the value saved in the register is valid. If, while debugging a program, you use the DEPOSIT command to change the value of the variable in the expression, then the value of that variable is changed, but the corresponding value stored in the register may not be. Thus, when execution continues, the value in the register may be used instead of the changed value in the expression, causing unexpected results.

In addition, when the value of a nonstatic variable (see Chapter 2) is held in a register, its value in memory is generally invalid; therefore, a spurious value may be displayed if you enter the EXAMINE command for a variable under these circumstances.

8.1.4 Use of Condition Codes

One optimization technique takes advantage of the way in which the VAX processor condition codes are set. For example, consider the following Pascal source code:

```
X := X + 2.5;
IF X < 0
THEN
    .
    .
    .
```

Rather than test the new value of X to determine whether to branch, the optimized code bases its decision on the condition code setting after 2.5 is added to X. Thus, if you attempt to set a breakpoint on the IF and deposit a different value into X, you do not achieve the intended result because the condition codes no longer reflect the value of X. In other words, the decision to branch is being made without regard to the deposited value of the variable.

Again, you can use the command EXAMINE/OPERAND .%PC to determine the correct location for depositing so as to achieve the desired effect.

8.2 Debugging Screen-Oriented Programs

The debugger uses portions of the entire terminal screen for input and output (I/O) at various points during a debugging session. If you use a single terminal to debug a screen-oriented program that uses most or all of the screen, debugger I/O may overwrite, or may be overwritten by, program I/O.

Using one terminal for both program I/O and debugger I/O is even more complicated if you are debugging in screen mode and your screen-oriented program calls any VMS screen-management RTL routines (SMG$xxx). This is because the debugger's screen mode also calls SMG routines. In such cases, the debugger and your program share the same SMG pasteboard, causing further interference.

To avoid these problems when debugging a screen-oriented program, use one terminal for program I/O and another terminal for debugger I/O. If debugging at a VAXstation, you can achieve this effect easily by entering the command SET MODE SEPARATE, which creates a separate window for debugger I/O. At the very least, using two terminals (or two VAXstation windows) enables you to see the behavior of your program more clearly.

The rest of this section explains how to allocate a second terminal for debugger I/O, so that your current terminal is devoted exclusively to program I/O during a debugging session.

By default, the debugger input device is SYS$INPUT and the debugger output device is SYS$OUTPUT. The logical names DBG$INPUT and DBG$OUTPUT enable you to specify a new debugger input device and a new debugger output device, respectively.

Assume that TTD1: is your current terminal and you want to have debugger I/O at terminal TTD2:. Assigning both DBG$INPUT and DBG$OUTPUT to TTD2: enables you to enter debugger commands and observe debugger output at TTD2:. Meanwhile, you continue to enter program input and observe program output at TTD1:.

Note that on a properly secured system, terminals are protected so that you can log into but you cannot allocate a terminal. The following steps explain how to allocate TTD2:, assign DBG$INPUT and DBG$OUTPUT to TTD2:, run the program to be debugged, and then deallocate TTD2: after finishing the debugging session:

1 Ask your system manager (or a suitably privileged user) to provide you with read/write access to TTD2:. One way is to use a command line such as the following:

```
$ SET PROTECTION=WORLD:RW/DEVICE TTD2:
```

This command line provides world read/write access and, therefore, allows other users to also allocate and perform I/O to TTD2:.

The following technique is preferred because it uses an access control list (ACL). Assume that your UIC is [PROJ,JONES]. Then the system manager can restrict device access to you alone as follows:

```
$ SET DEVICE/ACL=(IDENT=[PROJ,JONES],ACCESS=(READ,WRITE)) TTD2:
```

Another method is for you to enable SYSPRV privilege in your own process so that you can allocate TTD2: without allowing the world to allocate it. However, note that it is risky to debug an erroneous program with SYSPRV enabled.

2 Once you have read/write access to the terminal, allocate it so that you have exclusive access to it:

```
$ ALLOCATE TTD2:
```

3 Now you can assign DBG$INPUT and DBG$OUTPUT to TTD2 as follows:

```
$ DEFINE DBG$INPUT TTD2:
$ DEFINE DBG$OUTPUT TTD2:
```

4 Make sure that the terminal type is known to the system. Use the following command:

```
$ SHOW DEVICE/FULL TTD2:
```

If the device type is "unknown", make it known to the system as follows (in this example, the terminal is assumed to be a VT100):

```
$ SET TERMINAL/PERMANENT/DEVICE=VT100 TTD2:
```

5 Now you can run your program and observe debugger input and output at TTD2:

```
$ RUN FORMS
```

6 When finished with the debugging session, deallocate TTD2 as follows:

```
$ DEALLOCATE TTD2:
```

8.3 Debugging Multilanguage Programs

The debugger enables you to debug modules whose source code is written in different languages, within the same debugging session. This section highlights some language specific behavior that you should be aware of, to minimize possible confusion.

When debugging in any language, be sure to consult the documentation supplied with that language. The chapter devoted to debugging, in the user's guide, contains all language dependent information for that language. See also Appendix E of this manual, which tabulates the constructs and operators that are supported by the debugger for each language.

8.3.1 Controlling the Current Debugger Language

At debugger startup, the debugger sets the *current language* to that in which the module containing the main program (usually the routine containing the image transfer address) is written. The current language is identified when you invoke the debugger. For example:

```
$ RUN FORMS

                    VAX DEBUG Version 5.0

%DEBUG-I-INITIAL, language is PASCAL, module set to 'FORMS'
DBG>
```

The current language setting determines how the debugger parses and interprets the names, operators, and expressions you specify in debugger commands, including things like the typing of variables, array and record syntax, the default radix for numeric data, case sensitivity, and so on. The language setting also determines how the debugger displays data associated with your program.

Many programs include modules that are written in languages other than that of the main program. To minimize confusion, by default the debugger language remains set to the language of the main program throughout a debugging session, even if execution is suspended within a module written in another language.

To take full advantage of symbolic debugging with such modules, use the SET LANGUAGE command to set the debugging context to that of another language. For example, the following command causes the debugger to interpret any symbols, expressions, and so on according to the rules of the COBOL language:

```
DBG> SET LANGUAGE COBOL
```

The keywords that you can use with the SET LANGUAGE command correspond to all of the VMS supported languages that are also supported by the debugger:

ADA
BASIC
BLISS
C
COBOL
DIBOL
FORTRAN
MACRO

PASCAL
PLI
RPG
SCAN

In addition, when debugging a program that is written in an unsupported language, you can specify the command SET LANGUAGE UNKNOWN. To maximize the usability of the debugger with unsupported languages, the SET LANGUAGE UNKNOWN command causes the debugger to accept a large set of data formats and operators, including some that may be specific to only a few supported languages. The operators and constructs that are recognized when the language is set to UNKNOWN are identified in Appendix E.

8.3.2 Specific Differences Among Languages

This section lists some of the differences you should keep in mind when debugging in various languages. Included are differences that are affected by the SET LANGUAGE command and other differences (for example, language specific initialization code and predefined breakpoints).

This list is not intended to be complete. Consult your language documentation for complete details.

8.3.2.1 Default Radix

The default radix for entering and displaying numeric data is hexadecimal for BLISS and MACRO and decimal for all other languages.

Use the SET RADIX command to establish a new default radix.

8.3.2.2 Evaluating Language Expressions

Several debugger commands and constructs evaluate language expressions:

- The EVALUATE, DEPOSIT, IF, FOR, REPEAT, and WHILE commands.
- WHEN clauses, which are used with the SET BREAK, SET TRACE, and SET WATCH commands.

When processing these commands, the debugger evaluates language expressions in the syntax of the current language and in the current radix as discussed in Section 3.1.5.

Note that operators vary widely among different languages (see Appendix E). For example, the following two commands evaluate equivalent expressions in Pascal and FORTRAN, respectively:

```
DBG> SET WATCH X WHEN (Y < 5)        ! Pascal
DBG> SET WATCH X WHEN (Y .LT. 5)     ! FORTRAN
```

Assume that the language is set to PASCAL and you have entered the first (Pascal) command. You now step into a FORTRAN routine, set the language to FORTRAN, and resume execution. While the language is set to FORTRAN, the debugger is not able to evaluate the expression (Y < 5). As a result, it sets an unconditional watchpoint and, when the watchpoint is triggered, returns a syntax error for the " <" operator.

This type of discrepancy can also occur if you use commands that evaluate language expressions in debugger command procedures and initialization files.

Note also that the debugger processes language expressions that contain variable names (or other address expressions) differently when the language is set to BLISS than when it is set to another language. See Section 3.1.5 for details.

8.3.2.3 Arrays and Records

The syntax for denoting array elements and record components (if applicable) varies among languages.

For example, some languages use brackets, [], and others use parentheses, (), to delimit array elements.

Some languages (like BASIC) have zero-based arrays. Some languages have one-based arrays, as in the following example:

```
DBG> EXAMINE INTEGER_ARRAY
PROG2\INTEGER_ARRAY
   (1,1):       27
   (1,2):       31
   (1,3):       12
   (2,1):       15
   (2,2):       22
   (2,3):       18
DBG>
```

For some languages (like Pascal and Ada) the specific array declaration determines how the array is based.

8.3.2.4 Case Sensitivity

Names and language expressions are case-sensitive in C. You must specify them exactly as they appear in the source code. For example, the following two commands are not equivalent when the language is set to C:

```
DBG> SET BREAK SCREEN_IO\%LINE 10
DBG> SET BREAK screen_io\%LINE 10
```

8.3.2.5 Initialization Code

If you have a multilanguage program that includes an Ada package, or a FORTRAN main program that was compiled with the /CHECK=UNDERFLOW (or /CHECK=ALL) qualifier, a NOTATMAIN message is issued when you invoke the debugger. For example:

```
$ RUN MONITOR

                    VAX DEBUG Version 5.0

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

The NOTATMAIN message indicates that execution is suspended *before* the start of the main program. This enables you to execute and check some initialization code under debugger control.

The initialization code is created by the compiler and is placed in a special PSECT named LIB$INITIALIZE. In the case of an Ada package, the initialization code belongs to the package body (which may contain statements to initialize variables, and so on). In the case of a FORTRAN program, the initialization code declares the handler that is needed if you specify the /CHECK=UNDERFLOW or /CHECK=ALL qualifier.

The NOTATMAIN message indicates that, if you do not want to debug the initialization code, you can execute immediately to the start of the main program by entering a GO command. You are then at the same point as when you invoke the debugger with any other program. Entering the GO command again starts program execution.

8.3.2.6 Ada Predefined Breakpoints

If your program is linked with a module that is written in Ada, two breakpoints that are associated with Ada tasking exception events are automatically established when you invoke the debugger. Note that these breakpoints are not affected by a SET LANGUAGE command. They are established automatically during debugger initialization when the Ada run-time library is present. When you enter a SHOW BREAK command under these conditions, the following breakpoints are displayed:

```
DBG> SHOW BREAK
Breakpoint on ADA event "DEPENDENTS_EXCEPTION" for any value
Breakpoint on ADA event "EXCEPTION_TERMINATED" for any value
```

These breakpoints are equivalent to entering the following commands:

```
DBG> SET BREAK/EVENT=DEPENDENTS_EXCEPTION
DBG> SET BREAK/EVENT=EXCEPTION_TERMINATED
```

8.4 Debugging Exceptions and Condition Handlers

A condition handler is a procedure that the VMS operating system executes when an exception condition occurs.

Exceptions include hardware conditions (such as an arithmetic overflow or a memory access violation) or signaled software exceptions (for example, an exception signaled because a file could not be found).

VMS conventions specify how, and in what order, various condition handlers set up by the operating system, the debugger, or your own program are invoked — for example, the primary handler, call frame (user-declared) handlers, and so on. Section 8.4.3 describes condition handling when you are using the debugger. See the *VMS Run-Time Library Routines Volume* for additional general information on condition handling.

Tools for debugging exceptions and condition handlers include the following:

- The SET BREAK/EXCEPTION and SET TRACE/EXCEPTION commands, which direct the debugger to treat any exception generated by your program as a breakpoint or tracepoint, respectively (see Section 8.4.1 and Section 8.4.2).
- Several built-in symbols (such as %EXC_NAME), which enable you to qualify exception breakpoints and tracepoints (see Section 8.4.4).
- The SET BREAK/EVENT and SET TRACE/EVENT commands, which enable you to break on or trace exception events that are specific to Ada and SCAN programs (see the corresponding documentation for more information).

8.4.1 Setting Breakpoints or Tracepoints on Exceptions

When you enter a SET BREAK/EXCEPTION (or SET TRACE/EXCEPTION) command, you direct the debugger to treat any exception generated by your program as a breakpoint (or tracepoint). As a result of a SET BREAK/EXCEPTION command, if your program generates an exception, the debugger suspends execution, reports the exception condition and the line where execution is suspended, and prompts for commands. The following example illustrates the effect:

```
DBG> SET BREAK/EXCEPTION
DBG> GO
    .
    .
    .
%SYSTEM-F-INTDIV, arithmetic trap, integer divide by zero at PC=0000066C, PSL=03C00022
break on exception preceding TEST\%LINE 13
     6:         X := 3/Y;
DBG>
```

Note that an exception breakpoint (or tracepoint) is triggered even if your program has a condition handler to handle the exception. The SET BREAK/EXCEPTION command causes a breakpoint to occur before any handler can execute (and thereby possibly dismiss the exception). Without the exception breakpoint, the handler would be executed, and the debugger would get control only if no handler dismissed the exception (see Section 8.4.2 and Section 8.4.3).

The following command line is useful for identifying where an exception occurred. It causes the debugger to display automatically the sequence of active calls and the PC value at an exception breakpoint.

```
DBG> SET BREAK/EXCEPTION DO (SET MODULE/CALLS; SHOW CALLS)
```

You can also create a screen-mode DO display that issues a SHOW CALLS command whenever the debugger interrupts execution. For example:

```
DBG> SET DISPLAY CALLS DO (SET MODULE/CALLS; SHOW CALLS)
```

An exception tracepoint (established with the SET TRACE/EXCEPTION command) is like an exception breakpoint followed by a GO command without an address expression specified.

An exception breakpoint cancels an exception tracepoint, and vice versa.

To cancel exception breakpoints or tracepoints, use the CANCEL BREAK /EXCEPTION or CANCEL TRACE/EXCEPTION command, respectively.

8.4.2 Resuming Execution at an Exception Breakpoint

When an exception breakpoint is triggered, execution is suspended *before* any user-declared condition handler is invoked. When you resume execution from the breakpoint with the GO, STEP, or CALL commands, the behavior is as follows:

- Entering a GO command without an address-expression parameter, or entering a STEP command, causes the debugger to resignal the exception. The GO command enables you to observe which user-declared handler, if any, next handles the exception. The STEP command causes you to step into that handler (see the next example).

- Entering a GO command with an address-expression parameter causes execution to resume at the specified location, thus inhibiting the execution of any user-declared handlers.
- A common debugging technique at an exception breakpoint is to call a dump routine with the CALL command (see Chapter 7). When you enter the CALL command at an exception breakpoint, no breakpoints, tracepoints, or watchpoints that were previously set within the called routine are active, so that the debugger does not lose the exception context. After the routine has executed, the debugger prompts for input. Entering a GO or STEP command at this point causes the debugger to resignal the exception, as for the first bulleted item in this list.

The following FORTRAN example shows how to determine the presence of a condition handler at an exception breakpoint and how a STEP command, entered at the breakpoint, enables you to step into the handler.

At the exception breakpoint, the SHOW CALLS command indicates that the exception was generated during a call to routine SYS$QIOW:

```
DBG> SET BREAK/EXCEPTION
DBG> GO
    .
    .
    .
%SYSTEM-F-SSFAIL, system service failure exception, status=0000013C, PC=7FFEDE06, PSL=03C00000
break on exception preceding SYS$QIOW+6
DBG> SHOW CALLS
 module name     routine name                    line       rel PC    abs PC
                 SYS$QIOW                                  00000006  7FFEDE06
*EXC$MAIN        EXC$MAIN                          23      0000003B  0000063B
```

The following SHOW STACK command indicates that no handler is declared in routine SYS$QIOW. However, one level down the call stack, routine EXC$MAIN has declared a handler named SSHAND:

```
DBG> SHOW STACK
stack frame 0 (2146296644)
    condition handler: 0
       SPA:            0
       S:              0
       mask:           ^M<R2,R3,R4,R5,R6,R7,R8,R9,R10,R11>
       PSW:            0020 (hexadecimal)
    saved AP:          2146296780
    saved FP:          2146296704
    saved PC:          EXC$MAIN\%LINE 25
    .
    .
    .
stack frame 1 (2146296704)
    condition handler: SSHAND
       SPA:            0
       S:              0
       mask:           ^M<R11>
       PSW:            0000 (hexadecimal)
    saved AP:          2146296780
    saved FP:          2146296760
    saved PC:          SHARE$DEBUG+2217
    .
    .
    .
```

At this exception breakpoint, entering a STEP command enables you to step directly into condition handler SSHAND:

```
DBG> STEP
stepped to routine SSHAND
     2:       INTEGER*4 FUNCTION SSHAND (SIGARGS, MECHARGS)
DBG> SHOW CALLS
 module name     routine name                    line       rel PC    abs PC
*SSHAND          SSHAND                             2     00000002  00000642
----- above condition handler called with exception 0000045C:
%SYSTEM-F-SSFAIL, system service failure exception, status=0000013C, PC=7FFEDE06, PSL=03C00000
----- end of exception message
                 SYS$QIOW                                   00000006  7FFEDE06
*EXC$MAIN        EXC$MAIN                          23       0000003B  0000063B
```

The debugger symbolizes the addresses of condition handlers into names if that is possible. However, note that with some languages, exception conditions are first handled by an RTL routine, before any user-declared condition handler is invoked. In such cases, the address of the first condition handler may be symbolized to an offset from an RTL shareable image address.

8.4.3 Effect of Debugger on Condition Handling

When you run your program with the debugger, at least one of the following condition handlers is invoked, in the order given, to handle any exceptions caused by the execution of your program:

1. Primary handler
2. Secondary handler
3. Call-frame handlers (user-declared). Also known as stack handlers.
4. Final handler
5. Last-chance handler
6. Catchall handler

A handler can return one of the following three status codes to the VAX Condition Handling Facility:

- SS$_RESIGNAL — The VMS operating system searches for the next handler.
- SS$_CONTINUE — The condition is assumed to be corrected, and execution continues.
- SS$_UNWIND — The call stack is unwound some number of frames, if necessary, and the signal is dismissed.

For more information on condition handling, see the *VMS Run-Time Library Routines Volume.*

8.4.3.1 Primary Handler

When you run your program with the debugger, the primary handler is the debugger. Therefore, the debugger has the first opportunity to handle an exception condition, whether or not the exception is caused by the debugger (Section 2.6 describes how the debugger causes exceptions to occur in your program in order to control and monitor execution).

If you have entered a SET BREAK/EXCEPTION or SET TRACE/EXCEPTION command, the debugger breaks on (or traces) any exceptions caused by your program. The break (or trace) action occurs before any user-declared handler is invoked.

If you have not entered a SET BREAK/EXCEPTION or SET TRACE/EXCEPTION command, the primary handler resignals any exceptions caused by your program.

8.4.3.2 Secondary Handler

The secondary handler is used for special purposes and does not apply to the types of programs covered in this manual.

8.4.3.3 Call-Frame Handlers (User-Declared)

Each routine of your program can establish a condition handler, also known as a call-frame handler. The operating system searches for these handlers starting with the routine that is currently executing. If no handler was established for that routine, the system searches for a handler established by the next routine down the call stack, and so on back to the main program, if necessary.

Once invoked, a handler may perform one of the following actions:

- It handles the exception condition, thus allowing the program to continue execution.
- It resignals the exception. The operating system then searches for another handler down the call stack.
- It encounters a breakpoint or watchpoint, thereby suspending execution at the breakpoint or watchpoint.
- It generates its own exception. In this case, the primary handler is invoked again.
- It exits, thus terminating program execution.

8.4.3.4 Final and Last-Chance Handlers

These handlers are controlled by the debugger. They enable the debugger to ultimately regain control and display the DBG> prompt if no user-declared handler has handled an exception. Otherwise, the debugging session would terminate, and control would pass to the DCL command interpreter.

The final handler is the last frame on the call stack and the first of these two handlers to be invoked. The following example illustrates what happens when an unhandled exception condition is propagated from an exception breakpoint to the final handler:

```
DBG> SET BREAK/EXCEPTION
DBG> GO
    .
    .
    .
%SYSTEM-F-INTDIV, arithmetic trap, integer divide by zero at PC=0000066C, PSL=03C00022
break on exception preceding TEST\%LINE 13
     6:         X := 3/Y;
DBG> GO
%SYSTEM-F-INTDIV, arithmetic trap, integer divide by zero at PC=0000066C, PSL=03C00022
DBG>
```

In this example, the first INTDIV message is issued by the primary handler, and the second is issued by the final handler, which then displays the DBG> prompt.

The last-chance handler is invoked only if the final handler cannot gain control because the stack is corrupted. For example:

```
DBG> DEPOSIT %FP = 10
DBG> GO
    .
    .
    .
%SYSTEM-F-ACCVIO, access violation, reason mask=00, virtual address=0000000A, PC=0000319C, PSL=03C00000
%DEBUG-E-LASTCHANCE, stack exception handlers lost, re-initializing stack
DBG>
```

8.4.3.5 Catchall Handler

The catchall handler, which is part of the VMS operating system, is invoked if the last-chance handler cannot gain control. The catchall handler produces a register dump. This should never occur if the debugger has control of your program. But it may occur if your program encounters an error when running without the debugger.

If, during a debugging session, you observe a register dump and are returned to DCL level, submit an SPR to DIGITAL.

8.4.4 Exception-Related Built-in Symbols

When an exception is signaled, the debugger sets the following exception-related built-in symbols.

Symbol	Description
%EXC_FACILITY	Name of facility that issued the current exception
%EXC_NAME	Name of current exception
%ADAEXC_ NAME	Ada exception name of current exception (for Ada programs only)
%EXC_NUMBER	Number of current exception
%EXC_SEVERITY	Severity code of current exception

You can use these symbols as follows:

- To obtain information about the fields of the VMS condition code of the current exception.

- To qualify exception breakpoints (or tracepoints) so that they trigger only on certain kinds of exceptions.

The following examples illustrate the use of some of these symbols.

```
DBG> EVALUATE %EXC_NAME
'ACCVIO'
DBG> SET TRACE/EXCEPTION WHEN (%EXC_NAME = "ACCVIO")
DBG> EVALUATE %EXC_FACILITY
'SYSTEM'
DBG> EVALUATE %EXC_NUMBER
12
DBG> EVALUATE/CONDITION_VALUE %EXC_NUMBER
%SYSTEM-F-ACCVIO, access violation, reason mask=01, virtual address=FFFFFF30, PC=00007552, PSL=03C00000
DBG> SET BREAK/EXCEPTION WHEN (%EXC_NUMBER = 12)

DBG> SET BREAK/EXCEPTION WHEN (%EXC_SEVERITY .NE. "I" .AND. %EXC_SEVERITY .NE. "S")
```

8.5 Debugging Exit Handlers

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

At program termination, the debugger exit handler executes after all user-declared exit handlers have executed.

To debug a user-declared 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 (usually a call to $EXIT), or by allowing your program to terminate, or by entering the EXIT command (note that the QUIT command does not execute any user declared exit handlers). When the exit handler executes, the breakpoint is activated and control is then returned to the debugger, which prompts for commands.

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 are called. A 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
exit handler at BLIHANDLER\HANDLER1
```

8.6 Debugging AST-Driven Programs

A program may use asynchronous system traps (ASTs) either explicitly, or implicitly by calling VMS system services or RTL routines that call user-defined AST routines. Section 8.6.1 explains how to facilitate debugging by disabling and enabling the delivery of ASTs originating with your program. Section 8.6.2 explains how delivery of an AST affects a SHOW CALLS display.

8.6.1 Disabling and Enabling the Delivery of ASTs

Debugging AST-driven programs may be confusing because interrupts originating from the program being debugged may occur, but will not be processed, while the debugger is running (processing commands, tracing execution, displaying information, and so on).

By default, the delivery of ASTs is enabled while the program is running. The command DISABLE AST disables the delivery of ASTs while the program is running and causes any such potential interrupts to be queued.

The delivery of ASTs is always disabled when the debugger is running.

The command ENABLE AST reenables the delivery of ASTs, including any pending ASTs. The command SHOW AST indicates whether the delivery of ASTs is enabled or disabled.

To control the delivery of ASTs during the execution of a routine called with the CALL command, use the /[NO]AST qualifiers. The command CALL/AST enables the delivery of ASTs in the called routine. The command CALL /NOAST disables the delivery of ASTs in the called routine. If you do not specify /AST or /NOAST with the CALL command, the delivery of ASTs is enabled unless you have previously entered the command DISABLE AST.

8.6.2 Call Frames Associated with ASTs in SHOW CALLS Display

The delivery of an AST creates one or more special call frames that appear in a SHOW CALLS display. These call frames are not symbolized and may make the SHOW CALLS display confusing. The following example illustrates what you might see in a SHOW CALLS display when an AST routine is on the call stack.

Assume that a program calls the system service $SETIMR to set a timer that expires at a specified interval and then execute a user-defined AST routine, TIMER_ROUT, in the program.

The following command lines set a breakpoint on routine TIMER_ROUT, start execution which is then suspended on that routine, and display the sequence of active calls at the breakpoint:

```
DBG> SET BREAK TIMER_ROUT
DBG> GO
break at routine MOD1\TIMER_ROUT
    14:     X = .X + 1;
DBG> SHOW CALLS
module name     routine name            line      rel PC    abs PC
*MOD1           TIMER_ROUT                14    00000002  0000040E
                                                  00000000  80009E5E
```

The bottom line is the call frame associated with the system AST dispatcher. It shows the absolute PC value when the AST was delivered. Because the AST dispatcher is in system space (as indicated by the high absolute address), no symbolic information (module name, routine name, line number) is available. A SHOW CALLS display associated with the delivery of an AST may also show some debugger call frames (module name SHARE$DEBUG) and diagnostic messages related to condition handling by the debugger. You should ignore such messages and call frames.

Part II Debugger Command Dictionary

This part contains detailed reference information on the debugger commands.

This part contains detailed reference information on all debugger commands, organized as follows:

- Section 1 describes the general format for debugger commands.
- Section 2 gives the rules for entering and terminating commands, both interactively at the terminal and within a debugger command procedure.
- Section 3 lists commands that apply only when you are running the debugger on a VAXstation.
- Section 4 lists commands and qualifiers that are obsolete starting with VMS Version 5.0.
- Section 5, which is most of Part II, is the debugger command dictionary.

1 General 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, OUT2, and PROG4.COM are parameters (@ is the "execute procedure" command); /SCROLL and /OUTPUT are qualifiers.

```
DBG> SET WATCH COUNT
DBG> EXAMINE I,J,K
DBG> SELECT/SCROLL/OUTPUT OUT2
DBG> @PROG4.COM
```

Some commands accept optional WHEN or DO clauses. DO clauses are also used in some screen display definitions.

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 in the order that they are listed. You must separate multiple command strings with semicolons (;). These points are illustrated in the next example.

The following command string sets a breakpoint on routine SWAP that is 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)
```

The debugger checks the syntax of the commands in a DO clause when it executes the DO clause. You can nest commands within DO clauses.

2 Rules for 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 @ (execute procedure) command. The conventions are described for each mode of operation.

When you use any debugger command, if the debugger issues a diagnostic message with a severity level of I (informational), the command is still executed. The debugger aborts an illegal command line only when the severity level of the message is W (warning) or greater.

2.1 Interactively at the Terminal

When entering a debugger 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 debugger 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. As a result, the debugger prompt is 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 command strings with a semicolon (;).

To enter a comment (explanatory text that is recorded in a debugger log file but is otherwise ignored by the debugger), precede the comment text with an exclamation point (!). If the comment wraps to another line, start that line with an exclamation point.

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 the 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 puts 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.)

2.2 Within a Debugger 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.

Start a debugger command line at the left margin (in contrast, each command line of a DCL command procedure starts with a dollar sign ($)).

The start of a new line terminates the previous command line (end of file also terminates the previous command 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 command strings with a semicolon (;).

To enter a comment (explanatory text that does not affect the execution of the command procedure), precede the comment text with an exclamation point (!). If the comment wraps to another line, start that line with an exclamation point.

3 Commands Recognized Only on VAXstations

The following commands are recognized only when you are running the debugger on a VAXstation:

- SET MODE [NO]SEPARATE
- SET PROMPT/[NO]POP

See the descriptions of these commands in the command dictionary in Section 5. All of the other debugger commands apply to VAXstations as well as terminals.

4 Obsolete Commands

The following debugger commands and command qualifiers are obsolete starting with VMS Version 5.0 and are no longer documented. For compatibility with previous VMS versions, these commands and qualifiers will be supported indefinitely, however, except as indicated.

Obsolete Command or Qualifier	Reason
ALLOCATE	The debugger now allocates and deallocates memory automatically. This command now has no effect.
CANCEL EXCEPTION BREAK	This command duplicates the effect of the newer command CANCEL BREAK/EXCEPTION, which better conforms to the general command format for canceling breakpoints.
SET EXCEPTION BREAK	This command duplicates the effect of the newer command SET BREAK/EXCEPTION, which better conforms to the general command format for setting breakpoints.
SET MODULE/ALLOCATE	The debugger now allocates and deallocates memory automatically. This qualifier now has no effect.
UNDEFINE	This command duplicates the effect of the newer command DELETE, which conforms to the analogous DCL command DELETE.
UNDEFINE/KEY	This command duplicates the effect of the newer command DELETE /KEY, which conforms to the analogous DCL command DELETE/KEY.

5 Debugger Command Dictionary

The debugger command dictionary, which starts on the next page, describes each of the debugger commands in detail. Commands are listed alphabetically. The following information is provided for each command: command description, format, parameters, qualifiers, and one or more examples. See the preface of this manual for documentation conventions.

@ (Execute Procedure)

Executes a debugger command procedure.

FORMAT

@ file-spec [parameter[, . . .]]

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. You can specify a logical name.

parameter

Specifies a parameter that is passed to the command procedure. The parameter may be an address expression, a value expression in the current language, or a debugger command (the command must be enclosed within quotation marks (")). Note that, unlike with DCL, you must separate parameters by commas. Also, you can pass as many parameters as there are formal parameter declarations within the command procedure. For more information on passing parameters to command procedures, see the DECLARE command description.

QUALIFIERS

None.

DESCRIPTION

A debugger command procedure can contain any debugger commands, including another @ 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 command such as SET BREAK, or a DO clause in a screen display definition.

By default, commands read from a command procedure are not echoed. If you enter the command 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).

For information on passing parameters to command procedures, see the DECLARE command description.

Related commands: (SET, SHOW) ATSIGN, SET OUTPUT [NO]VERIFY, SHOW OUTPUT, DECLARE.

EXAMPLE

```
DBG> SET ATSIGN USER:[JONES.DEBUG].DBG
DBG> SET OUTPUT VERIFY
DBG> @CHECKOUT
%DEBUG-I-VERIFYICF, entering command procedure CHECKOUT
 SET MODULE/ALL
 SET BREAK SUB1
 GO
break at routine PROG5\SUB2
 EXAMINE X
PROG5\SUB2\X:  376
    .
    .
    .
%DEBUG-I-VERIFYICF, exiting command procedure MAIN
DBG>
```

In this example, the command SET ATSIGN establishes that debugger command procedures are, by default, in USER:[JONES.DEBUG] and have a file type of DBG. The command @CHECKOUT executes the command procedure USER:[JONES.DEBUG]CHECKOUT.DBG. Commands contained within the command procedure are echoed because the command SET OUTPUT VERIFY was entered.

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 use the SPAWN command to create 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 command: SPAWN.

EXAMPLES

1

```
DBG> SPAWN
$ ATTACH JONES
%DEBUG-I-RETURNED, control returned to process JONES
DBG> ATTACH JONES_1
$
```

In this example, the series of commands creates 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 follows:

%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. For complex or structured variables (such as arrays, records, and access types), the address is passed when you specify %ADDR, but the called routine may not handle the passed data properly. Do not specify a literal value (a number or an expression composed of numbers) when using %ADDR.

%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 the delivery of aynchronous system traps (ASTs) is enabled or disabled during the execution of the called routine. /AST specifies that ASTs can be delivered. /NOAST specifies that ASTs cannot be delivered. By default, if you do not specify /AST or /NOAST, delivery of ASTs is enabled in the called routine if it was enabled before the CALL command was issued.

DESCRIPTION

The CALL command is one of the four debugger commands that can cause your program to execute (the others are GO, STEP, and EXIT). The command enables you to execute a routine independently of the normal execution of your program.

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

When you enter the CALL command at an exception breakpoint, any breakpoints, tracepoints, or watchpoints that were previously set within the called routine are disabled temporarily so that the debugger does not lose the exception context. However, such eventpoints are active if you enter the CALL command at a location other than an exception breakpoint.

When you enter 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. By VMS convention, after a called routine has executed, register R0 contains the function return value (if the routine is a function) or the procedure completion status (if the routine is a procedure that returns a status value). If a called procedure does not return a status value or function value, the value in R0 may be meaningless, and the "value returned" message can be ignored.
6. Restores the values of the general registers to the values they had just before the CALL command was executed.
7. Issues the prompt.

The debugger assumes that the called routine conforms to the 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. Therefore, you may need to specify how to pass parameters—for example, use CALL SUB1(%VAL X) rather than CALL SUB1(X). 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 the address of "X" as the required parameter (by default, the address of the argument specified is passed). The routine is a function whose returned value is 19.

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.

3

```
DBG> SET MODULE SHARE$LIBRTL
DBG> CALL LIB$SHOW_VM
 1785 calls to LIB$GET_VM, 284 calls to LIB$FREE_VM, 122216 bytes still allocated
value returned is 00000001
```

This example shows how you could call the run-time library routine LIB$SHOW_VM (in the shareable image LIBRTL) to display virtual memory statistics. The SET MODULE command makes the universal symbols (routine names) in LIBRTL visible in the main image. See the description of the /SHARE qualifier of the SHOW MODULE command for more information on this subject.

4

```
      SUBROUTINE CHECK_TEMPERATURE (TEMPERATURE,ERROR_MESSAGE)
         REAL TOLERANCE /4.7/
         REAL TARGET_TEMPERATURE /92.0/
         CHARACTER*(*) ERROR_MESSAGE

         IF (TEMPERATURE .GT. (TARGET_TEMPERATURE + TOLERANCE)) THEN
            TYPE *,'Input temperature out of range:',TEMPERATURE
            TYPE *,ERROR_MESSAGE
         ELSE
            TYPE *,'Input temperature in range:',TEMPERATURE
         END IF
      RETURN
      END

DBG> CALL CHECK_TEMPERATURE(%REF 100.0, %DESCR 'TOLERANCE-CHECK 1 FAILED')
Input temperature out of range:   100.0000
TOLERANCE-CHECK 1 FAILED
value returned is 0

DBG> CALL CHECK_TEMPERATURE(%REF 95.2, %DESCR 'TOLERANCE-CHECK 2 FAILED')
Input temperature in range:   95.2000
value returned is 0
```

In this example, the source code is that of a FORTRAN routine (CHECK_TEMPERATURE) that accepts two parameters, TEMPERATURE (a real number) and ERROR_MESSAGE (a string). Depending on the value of the temperature, the routine prints different output. Each of the two CALL commands passes a temperature value (by reference) and an error message (by descriptor). Because this routine does not have a formal return value, the value returned is undefined, in this case, 0.

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 entering 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 entering 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 entering 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 entering 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 asterisk wildcard character (*). Do not specify an address expression when using any of the qualifiers except for /EVENT.

QUALIFIERS

/ALL

Cancels all breakpoints. Do not specify an address expression with /ALL.

/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: This qualifier 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 entering 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.

/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 permits you also to specify an address expression for certain event names.

Note that the command CANCEL ALL also cancels all breakpoints.

Related commands: (SET, SHOW) BREAK, (SET, SHOW, CANCEL) TRACE, CANCEL ALL, (SET, SHOW) EVENT_FACILITY.

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 asterisk wildcard character (*). Do not specify a display name with /ALL.

QUALIFIERS

/ALL

Cancels all displays, except for the PROMPT display. Do not specify a display name with /ALL.

DESCRIPTION

When a display is canceled, its contents are permanently lost, it is deleted 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.

EXAMPLES

1 `DBG> CANCEL DISPLAY SRC2`

This command permanently deletes display SRC2.

2 `DBG> CANCEL DISPLAY/ALL`

This command permanently deletes all displays, except for the PROMPT display.

CANCEL IMAGE

Deletes 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 asterisk wildcard character (*). Do not specify an image name with /ALL.

QUALIFIERS

/ALL

Specifies that all shareable images except the main image are to be canceled. Do not specify an image name with /ALL.

DESCRIPTION

The CANCEL IMAGE command deallocates the data structures previously built to debug a shareable image by a SET IMAGE command. Use the CANCEL IMAGE command if the debugger performance has slowed down because of many images and modules being set. You can also use the CANCEL MODULE command to delete only certain modules from an image's run-time symbol table (RST) without canceling the entire image. Also, if dynamic mode is enabled (this is the default), you can disable it with the command SET MODE NODYNAMIC. As a result, the debugger does not set images or modules automatically.

If the current image (the image last set with the SET IMAGE command) is canceled, the main image (the image containing the image transfer address) becomes the current image.

Related commands: (SET, SHOW) IMAGE, (SET, SHOW, CANCEL) MODULE, SET MODE [NO]DYNAMIC.

EXAMPLE

```
DBG> CANCEL IMAGE SHARE2,SHARE3
```

This 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 entering the following commands:

```
DBG> SET MODE SYMBOLIC,LINE,NOG_FLOAT,NOSCREEN,SCROLL,KEYPAD,-
     DYNAMIC,NOSEPARATE
DBG> CANCEL RADIX
```

Note that, although the same default modes apply to all languages, the default radix for both data entry and display is decimal for all languages except BLISS and MACRO. It is hexadecimal for BLISS and MACRO.

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 in the current image from the run-time symbol table (RST) for that image.

FORMAT

CANCEL MODULE *[module-name[, . . .]]*

PARAMETERS

module-name

Specifies the name of a module whose symbol records are to be deleted from the RST. Do not use the asterisk wildcard character (*). Do not specify a module name with /ALL.

QUALIFIERS

/ALL

Deletes the symbol records of all modules from the RST. Do not specify a module name with /ALL.

/[NO]RELATED

Note: This qualifier 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 current image is either the main image (by default) or the image established as the current image by a previous SET IMAGE command.

Use the CANCEL MODULE command if the debugger performance has slowed down because of many modules being set. You can also use the CANCEL IMAGE command to delete the symbols of an entire image (this automatically cancels all of the modules in that image). Also, if dynamic mode is enabled (this is the default), you can disable it with the command SET MODE NODYNAMIC. As a result, the debugger does not set modules or images automatically.

The CANCEL MODULE command does not cancel any breakpoints, tracepoints, or watchpoints that are set currently. It deletes the symbolization of any breakpoints, tracepoints, or watchpoints associated with the canceled modules.

Related commands: (SET, SHOW) MODULE, SET MODE [NO]DYNAMIC, (SET, SHOW, CANCEL) IMAGE.

CANCEL MODULE

EXAMPLES

1
```
DBG> CANCEL MODULE SUB1
```

This command deletes the symbols of module SUB1 from the RST.

2
```
DBG> CANCEL MODULE/ALL
```

This command deletes the symbols of all modules from the RST.

CANCEL RADIX

Restores the default radix for the entry and display of integer data.

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 (decimal for all languages except BLISS and MACRO, hexadecimal for BLISS and MACRO).

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 (decimal for all languages except BLISS and MACRO, hexadecimal for BLISS and MACRO).

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, EVALUATE.

EXAMPLES

1
```
DBG> CANCEL RADIX
```

This command restores the default input and output radix.

2
```
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: This qualifier 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]
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]
DBG> CANCEL SOURCE/MODULE=COBOLTEST
DBG> SHOW SOURCE
no directory search list in effect
```

In this example, 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

Cancels tracepoints.

FORMAT

CANCEL TRACE *[address-expression[, . . .]]*

PARAMETERS

address-expression

Specifies a tracepoint to be canceled. Do not use the asterisk wildcard character (*). Do not specify an address expression when using any of the qualifiers except for /EVENT.

QUALIFIERS

/ALL

Cancels all tracepoints. Do not specify an address expression with /ALL.

/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: This qualifier 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 entering 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.

/LINE

Cancels the effect of a previous SET TRACE/LINE command. Do not specify an address expression with /LINE.

CANCEL TRACE

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 permits you also to 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, (SET, SHOW) EVENT_FACILITY.

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 is interpreted according to that type.

Related commands: (SET, SHOW) TYPE/OVERRIDE, EXAMINE, DEPOSIT.

EXAMPLE

```
DBG> CANCEL TYPE/OVERRIDE
```

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 asterisk wildcard character (*). Do not specify an address expression with /ALL.

QUALIFIERS

/ALL

Cancels all watchpoints. Do not specify an address expression with /ALL.

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.

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 asterisk wildcard character (*). Do not specify a window definition name with /ALL.

QUALIFIERS

/ALL

Cancels all predefined and user-defined window definitions. Do not specify a window definition name with /ALL.

DESCRIPTION

When a window definition is canceled, you can 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 deletes the screen window definition MIDDLE.

CTRL/C, CTRL/W, CTRL/Y, CTRL/Z

CTRL/Y interrupts a debugging session, or interrupts a program that is running without debugger control (enabling you to then invoke the debugger with the DCL DEBUG command). CTRL/C is like CTRL/Y unless the program has a CTRL/C AST service routine enabled. 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 a CTRL/C AST service routine enabled, 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.

When you enter a CTRL/Y—DEBUG sequence, the DCL command interpreter causes an SS$_DEBUG exception to be delivered. Note that you cannot invoke the debugger with a CTRL/Y—DEBUG sequence if your program has a handler that will prevent the SS$_DEBUG exception from reaching the traceback handler.

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. You may then want to enter any of the following DCL commands:

- 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 enter debugger commands.
- CONTINUE — You return to the debugging session at the same point where you interrupted it. Control passes either to the debugger or to your program, whichever had control when you pressed CTRL/Y.

- EXIT — Causes normal termination of the debugging session. The debugger exit handler is executed.
- STOP — Causes abnormal termination of the debugging session. The debugger exit handler is not executed.
- Any other DCL command — Causes orderly termination of the debugging session. The debugger exit handler is executed.

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 /TRACEBACK qualifier. To reference all of 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>
```

In this example, 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

Declares a formal parameter within a command procedure. This enables you to pass an actual parameter to the procedure when entering an @ (Execute Procedure) command.

FORMAT

DECLARE *p-name:p-kind [,p-name:p-kind, . . .]*

PARAMETERS

p-name

Specifies a formal parameter (a symbol) that is declared within the command procedure.

Do not specify a null parameter (represented either by two consecutive commas or by a comma at the end of the command).

p-kind

Specifies the parameter kind of a formal parameter. Valid keywords are the following:

ADDRESS	Specifies that the actual parameter is to be interpreted as an address expression. Has the same effect as the command DEFINE/ADDRESS p-name = actual-parameter.
COMMAND	Specifies that the actual parameter is to be interpreted as a command. Has the same effect as the command DEFINE/COMMAND p-name = actual-parameter.
VALUE	Specifies that the actual parameter is to be interpreted as a value expression in the current language. Has the same effect as the command DEFINE/VALUE p-name = actual-parameter.

QUALIFIERS

None.

DESCRIPTION

The DECLARE command is valid only within a command procedure.

The DECLARE command binds one or more actual parameters (specified on the command line following the "@ file-spec") to formal parameters (symbols) declared within a command procedure.

Each p-name:p-kind pair specified by a DECLARE command binds one formal parameter to one actual parameter. Formal parameters are bound to actual parameters in the order in which the debugger processes the parameter declarations. If you specify several formal parameters on a single DECLARE command, the leftmost formal parameter is bound to the first actual parameter, the next formal parameter is bound to the second, and so on. If you use a DECLARE command in a loop, the formal parameter is bound to the first actual parameter on the first iteration of the loop; the same formal parameter is bound to the second actual parameter on the next iteration, and so on.

Each parameter declaration acts like a DEFINE command: it associates a formal parameter with an address expression, a command, or a value expression in the current language, according to the parameter kind specified. The formal parameters themselves are consistent with those accepted by the DEFINE command and may in fact be deleted from the symbol table with the DELETE command. For more information, see the descriptions of the DEFINE and DELETE commands.

The %PARCNT built-in symbol, which can be used only within a command procedure, enables you to pass a variable number of parameters to a command procedure. The value of %PARCNT is the number of actual parameters passed to the command procedure.

Related commands: @ (Execute Procedure), DEFINE, DELETE.

EXAMPLES

1

```
! ***** Command Procedure EXAM.COM *****
SET OUTPUT VERIFY
DECLARE K:ADDRESS
EXAMINE K

DBG> @EXAM ARR4
%DEBUG-I-VERIFYIC, entering command procedure EXAM
 DECLARE K:ADDRESS
 EXAMINE K
PROG_8\ARR4
    (1):        18
    (2):        1
    (3):        0
    (4):        1
%DEBUG-I-VERIFYIC, exiting command procedure EXAM
```

In this example, the command DECLARE K:ADDRESS declares the formal parameter K within command procedure EXAM.COM. When EXAM.COM is executed, the actual parameter passed to EXAM.COM is interpreted as an address expression, and the command EXAMINE K displays the value of that address expression. The command SET OUTPUT VERIFY causes the commands to echo when they are read by the debugger.

At the debugger prompt, the command @EXAM ARR4 executes EXAM.COM, passing the actual parameter ARR4. Within EXAM.COM, ARR4 is interpreted as an address expression (an array variable, in this case).

2

```
! ***** Debugger Command Procedure EXAM_GO.COM *****
DECLARE L:ADDRESS, M:COMMAND
EXAMINE L; M

DBG> @EXAM_GO X "@DUMP"
```

In this example, the command procedure EXAM_GO.COM accepts two parameters, an address expression (L) and a command string (M). The address expression is then examined and the command is executed.

At the debugger prompt, the command @EXAM_GO X "@DUMP" executes EXAM_GO.COM, passing the address expression X and the command string @DUMP.

```
3  ! ***** Debugger Command Procedure VAR.DBG *****
   SET OUTPUT VERIFY
   FOR I = 1 TO %PARCNT DO (DECLARE X:VALUE; EVALUATE X)

   DBG> @VAR.DBG 12,37,45
   %DEBUG-I-VERIFYIC, entering command procedure VAR.DBG
    FOR I = 1 TO %PARCNT DO (DECLARE X:VALUE; EVALUATE X)
   12
   37
   45
   %DEBUG-I-VERIFYIC, exiting command procedure VAR.DBG
```

In this example, the command procedure VAR.DBG accepts a variable number of parameters. That number is stored in the built-in symbol %PARCNT.

At the debugger prompt, the command @VAR.DBG executes VAR.DBG, passing the actual parameters 12, 37, and 45. Therefore, %PARCNT has the value 3, and the FOR loop is repeated 3 times. The FOR loop causes the DECLARE command to bind each of the three actual parameters (starting with 12) to a new declaration of X. Each actual parameter is interpreted as a value expression in the current language, and the command EVALUATE X displays that value.

DEFINE

Assigns a symbolic name to an address expression, command, or value.

FORMAT

DEFINE *symbol-name=parameter*
[,symbol-name=parameter,...]

PARAMETERS

symbol-name

Specifies a symbolic name to be assigned to an address, command, or value. The symbolic name 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 symbolic name 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 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 enables you to assign a symbolic name to an address expression 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.

DEFINE

The DEFINE/COMMAND command enables you to define abbreviations for debugger commands or even define new commands, either from the debugger command level or from command procedures.

The DEFINE/VALUE command enables you to 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 enter 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.

If a symbol is redefined, the previous definition is canceled, even if different qualifiers were used with the DEFINE command.

Definitions created with the DEFINE/ADDRESS and DEFINE/VALUE commands are available only when the image in whose context they were created is the current image. If you use the SET IMAGE command to establish a new current image, these definitions are temporarily unavailable. Definitions created with the DEFINE/COMMAND and DEFINE/KEY commands are always available for all images, 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, DECLARE.

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)
```

In this example, the first command assigns a value of 0 to the symbol COUNTER. The second command causes the debugger to increment the value of the symbol COUNTER by 1 whenever address R is encountered. In other words, this example counts 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 Keyboard	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 enables you to 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. Newer terminals and workstations have the LK201 keyboard. On LK201 keyboards, the function keys you can use include all of the numeric keypad keys, the nonarrow 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, enter 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 enables you to 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
```

In this example, 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 the 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 the keypad key sequence BLUE—KP9, you can enter a line number and then press the RETURN key to terminate and 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
```

In this example, 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 enter "RED" commands by pressing keys that have definitions associated with the RED state.

DELETE

Deletes a symbol definition that was established with the DEFINE command.

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 asterisk wildcard character (*). Do not specify a symbol name with /ALL. 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). Do not specify a symbol name with /ALL.

/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.

Related command: DEFINE, SHOW SYMBOL/DEFINED, SHOW DEFINE, DECLARE.

EXAMPLES

1
```
DBG> DEFINE X = INARR, Y = OUTARR
DBG> DELETE X,Y
```

In this example, 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.

```
2  DBG> DELETE/ALL/LOCAL
```

In this example, the DELETE/ALL/LOCAL commmand deletes all local symbol definitions from the current command procedure.

DELETE/KEY

Deletes a key definition that was established with the DEFINE/KEY command or, by default, by the debugger.

FORMAT

DELETE/KEY *[key-name]*

PARAMETERS

key-name

Specifies a key whose definition is to be deleted. Do not use the asterisk wildcard character (*). Do not specify a key name with /ALL. Valid key names are as follows:

Key-name	LK201 Keyboard	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. Do not specify a key name with /ALL. 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

The DELETE/KEY command is like the DCL DELETE/KEY command.

Keypad mode must be enabled (SET MODE KEYPAD) before you can use this command. Keypad mode is enabled by default.

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 program variable. More generally, deposits a new value at the location denoted by an address expression.

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, this is typically the name of a variable and may include a path name to specify the variable uniquely. More generally, an address expression may also be a virtual memory address or a register and may be composed of numbers (offsets) and symbols, as well as one or more operators, operands, or delimiters. Appendix D identifies the operators that may be used in address expressions.

You cannot specify an entire aggregate variable (a composite data structure such as an array or a record). To specify an individual array element or a record component, use the syntax of the current language.

language-expression

Specifies the value to be deposited. You can specify any language expression that is valid in the current language. For most languages, the expression can include the names of simple (non-structured, single-valued) variables but not the names of aggregate variables (such as arrays or records). If the expression contains symbols with different compiler generated types, the debugger uses the rules of the current language to evaluate the expression.

If the expression is an ASCII string or a VAX assembly-language instruction, you must enclose it in quotation marks (") or apostrophes ('). If the string contains quotation marks or apostrophes, use the other delimiter to enclose the string.

If the string has more characters (1-byte ASCII) than can fit into the program location denoted by the address expression, the debugger truncates the extra characters from the right. If the string has fewer characters, the debugger pads the remaining characters to the right of the string by inserting ASCII space characters.

QUALIFIERS

/ASCIC

Deposits a counted ASCII string into the specified location. You must specify a string on the right-hand side of the equal sign. The deposited string is 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. You must specify a string on the right-hand side of the equal sign. 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 an ASCII string into the specified location. You must specify a string on the right-hand side of the equal sign. 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 into the specified location. You must specify a string on the right-hand side of the equal sign. The deposited string is 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. You must specify a string on the right-hand side of the equal sign. The deposited string is terminated by a zero byte that indicates 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, 21-DEC-1988 21:08:47.15) to the VMS internal format for date and time and deposits that value (length 8 bytes) into the specified location. Specify an absolute date and time in the following format: *[dd-mmm-yyyy[:]] [hh:mm:ss.cc]*.

/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 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 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 and deposits the resulting value into the specified location. The value of *n* is the number of decimal digits. Each digit occupies one nibble (4 bits).

/QUADWORD
Deposits a quadword integer (length 8 bytes) into the specified location.

/TASK
Note: This qualifier 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 declared in the program), then deposits the resulting value into the specified location. This enables you to specify a user-declared type.

/WORD
Deposits a word integer (length 2 bytes) into the specified location.

DESCRIPTION

The DEPOSIT command may be used to change the contents of any memory location or register that is accessible in your program. For high-level languages the command is used mostly to change the value of a variable (an integer, real, string, array, record, and so on).

The DEPOSIT command is like an assignment statement in most programming languages. The value of the expression specified to the right of the equal sign is assigned to the variable or other location specified to the left of the equal sign. Note that for Ada and Pascal, you can use ":=" instead of "=" in the command syntax.

The debugger recognizes the compiler generated types associated with symbolic address expressions (symbolic names declared in your program). Symbolic address expressions include the following:

- Variable names. When specifying a variable with the DEPOSIT command, use the same syntax that is used in the source code.
- Routine names, labels, and line numbers. These are associated with VAX instructions. You can deposit instructions using basically the same techniques as when depositing into string variables. However, you must also use the /INSTRUCTION qualifier or first enter a SET TYPE INSTRUCTION or SET TYPE/OVERRIDE INSTRUCTION command.

In general, when you enter a DEPOSIT command, the debugger does the following:

- It evaluates the address expression specified to the left of the equal sign, to yield a program location.
- If the program location has a symbolic name, the debugger associates the location with the symbol's compiler generated type. If the location does not have a symbolic name (and, therefore, no associated compiler generated type) the debugger associates the location with the type longword integer, by default. This means that, by default, you can deposit integer values that do not exceed 4 bytes into these locations.
- It evaluates the language expression specified to the right of the equal sign, in the syntax of the current language and in the current radix, to yield a value. The current language is the language last established with the SET LANGUAGE command. If no SET LANGUAGE command was entered, the current language is, by default, the language of the module containing the main program.
- It checks that the value and type of the language expression is consistent with the type of the address expression. If you try to deposit a value that is incompatible with the type of the address expression, the debugger issues a diagnostic message. If the value is compatible, the debugger deposits the value into the location denoted by the address expression.

The debugger may do type conversion during a deposit operation if the language rules allow it. For example a real value that is specified to the right of the equal sign may be converted to an integer value if it is being deposited into a location with an integer type. In general, the debugger tries to follow the assignment rules for the current language.

There are several ways of changing the type associated with a program location so that you can deposit data of a different type into that location:

- To change the default type for all locations that do *not* have a symbolic name, you can specify a new type with the SET TYPE command.
- To change the default type for *all* locations (both those that do and do not have a symbolic name), you can specify a new type with the SET TYPE/OVERRIDE command.
- To override the type currently associated with a particular location for the duration of a single DEPOSIT command, you can specify a new type by means of a qualifier (/ASCII:*n*, /BYTE, TYPE=(*type-expression*), and so on).

The debugger can interpret and display integer data in any one of four radixes: binary, decimal, hexadecimal, and octal. The default radix for both data entry and display is decimal for all languages except BLISS and MACRO. It is hexadecimal for BLISS and MACRO. You can use the SET RADIX and SET RADIX/OVERRIDE commands to change the default radix.

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.

Related commands: EXAMINE, EVALUATE, (SET, SHOW, CANCEL) RADIX, (SET, SHOW) TYPE, CANCEL TYPE/OVERRIDE.

DEPOSIT

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 into a variable (in this case, if you try to deposit an integer value into an enumerated type 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 21
```

This command deposits the expression %HEX 21 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 the location that is 20 bytes beyond that denoted by the symbol WORK.

14 `DBG> DEPOSIT/INSTR  SUB2+2 = 'MOVL   #20A,R0'`

This command deposits the instruction MOVL #20A,R0' into the location SUB2 + 2 bytes.

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 asynchronous system traps (ASTs) in your program.

FORMAT

DISABLE AST

PARAMETERS

None.

QUALIFIERS

None.

DESCRIPTION

The DISABLE AST command disables the delivery of ASTs in your program and thereby prevents interrupts from occurring while the program is running. If ASTs are delivered while the debugger is running (processing commands, and so on), they are queued and are delivered when control is returned to the program.

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 in your program, as confirmed with the SHOW AST command.

DISPLAY

Modifies an existing screen display.

FORMAT

DISPLAY *[disp-name [AT w-spec] [disp-kind]] [, . . .]*

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 (w-spec) and display kind (disp-kind).

w-spec

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 *w-spec* parameter, the screen position of the display is not changed.

disp-kind

Specifies the new display kind if you want to change the kind of display. Valid keywords are the following:

DO (command[; . . .])	Specifies an automatically updated output display. The commands are executed in the order listed each time the debugger gains control. Their output forms the contents of the display. If you specify more than one command, they must be separated by semicolons.
INSTRUCTION	Specifies an instruction display. If selected as the current instruction display with the SELECT/INSTRUCTION command, it displays 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 displays any debugger output that is not directed to another display. If selected as the current input display with the SELECT /INPUT command, it echoes debugger input. If selected as the current error display with the SELECT/ERROR command, it displays 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 displays 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. Do not use /GENERATE with /CLEAR.

/[NO]DYNAMIC

Controls whether a display automatically adjusts its window dimensions proportionally when the screen height or width is changed by a SET TERMINAL command. By default (/DYNAMIC), all user-defined and predefined displays adjust their dimensions automatically.

/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. Do not use /CLEAR with /GENERATE.

/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 entered /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. Do not specify any command parameters with /REFRESH. You can also use CTRL/W to refresh the screen.

/REMOVE

Marks the display as being removed from the display pasteboard, so it is not 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
```

In this example, 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 now echoes debugger input.

EDIT

Invokes the editor established with the SET EDITOR command. If no SET EDITOR command was entered, 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 editor. If you specify /EXIT, the debugging session is terminated and the editor is then invoked. If you specify /NOEXIT (default), the editing session is started and you return to your debugging session after terminating the editing session.

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
```

In this example, 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 is positioned at the start of the line that was centered in the source display.

2
```
DBG> EDIT SWAP\12
```

In this example, the EDIT command spawns the VAX Language-Sensitive Editor in a subprocess to edit the source file containing the module SWAP. The editing cursor is positioned at the start of source line 12.

3
```
DBG> SET EDITOR/CALLABLE_EDT
DBG> EDIT
```

In this example, 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 is 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 ASTs while your program is running, including any pending ASTs (ASTs waiting to be delivered). If ASTs are delivered while the debugger is running (processing commands, and so on), they are queued and are delivered when control is returned to the program. Delivery of ASTs in your program 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 in your program, as confirmed with the SHOW AST command.

EVALUATE

Evaluates a language expression in the current language (by default, the language of the module containing the main program).

FORMAT

EVALUATE *language-expression[, . . .]*

PARAMETERS

language-expression

Specifies any valid expression in the current language.

QUALIFIERS

/BINARY

Specifies that the result be displayed in binary radix.

/CONDITION_VALUE

Specifies that the expression be interpreted as a 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.

/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 expression specified in an EVALUATE command as a language expression, evaluates it in the syntax of the current language and in the current radix, and displays its value as a literal (for example, an integer value) in the current language.

The current language is the language last established with the SET LANGUAGE command. If no SET LANGUAGE command was entered, the current language is, by default, the language of the module containing the main program.

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.

The debugger can interpret and display integer data in any one of four radixes: binary, decimal, hexadecimal, and octal. The current radix is the radix last established with the SET RADIX command. If no SET RADIX command was entered, the current radix for both data entry and display is, by default, decimal for all languages except BLISS and MACRO. It is hexadecimal for BLISS and MACRO. You can use a radix qualifier with the EVALUATE command (/BINARY, /OCTAL, and so on) to display integer data in some other radix. These qualifiers do not affect how the debugger

interprets the data you specify — that is, they override the current output radix, but not the input radix.

Debugger support for language-specific operators and constructs is described in Appendix E.

Related commands: EVALUATE/ADDRESS, (SET, SHOW) LANGUAGE, (SET, SHOW, CANCEL) RADIX, (SET, SHOW) TYPE.

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
```

In this example, the EVALUATE command evaluates the logical AND of the current values of two boolean variables, WILLING and ABLE.

5
```
DBG> EVALUATE COLOR'FIRST
RED
```

In this 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 or a register name.

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 memory address is to be displayed in binary radix.

/DECIMAL

Specifies that the memory address is to be displayed in decimal radix.

/HEXADECIMAL

Specifies that the memory address is to be displayed in hexadecimal radix.

/OCTAL

Specifies that the memory address is to be displayed in octal radix.

DESCRIPTION

The EVALUATE/ADDRESS command enables you to determine the virtual address or register associated with an address expression.

The debugger can interpret and display integer data in any one of four radixes: binary, decimal, hexadecimal, and octal. The default radix for both data entry and display is decimal for all languages except BLISS and MACRO. It is hexadecimal for BLISS and MACRO. You can use a radix qualifier with the EVALUATE command (/BINARY, /OCTAL, and so on) to display address values in some other radix. Note that these qualifiers do not affect how the debugger interprets the data you specify — that is, they override the current output radix, but not the input radix.

If the value of a variable is currently stored in a register instead of virtual memory, the EVALUATE/ADDRESS command identifies the register. The radix qualifiers have no effect in that case.

Related commands: EVALUATE, (SET, SHOW, CANCEL) RADIX, SYMBOLIZE, SHOW SYMBOL/ADDRESS.

EXAMPLES

1
```
DBG> EVALUATE/ADDRESS MODNAME\%LINE 110
3942
```

This command displays the virtual memory address denoted by the address expression MODNAME\%LINE 110.

2
```
DBG> EVALUATE/ADDRESS/HEX A,B,C
000004A4
000004AC
000004A0
```

This command displays the virtual memory addresses denoted by the address expressions A, B, and C in hexadecimal radix.

3
```
DBG> EVALUATE/ADDRESS X
MOD3\%R1
```

This command indicates that variable X is associated with register R1. X is a nonstatic (register) variable.

EXAMINE

Displays the current value of a program variable. More generally, displays the value of the entity denoted by an address expression.

FORMAT

EXAMINE *[address-expression[:address-expression]] [, . . .]*

PARAMETERS

address-expression

Specifies an entity to be examined. With high-level languages, this is typically the name of a variable and may include a path name to specify the variable uniquely. More generally, an address expression may also be a virtual memory address or a register and may be composed of numbers (offsets) and symbols, as well as one or more operators, operands, or delimiters. Appendix D identifies the operators that may be used in address expressions.

If you specify the name of an *aggregate* variable (a composite data structure such as an array or a record structure), the debugger displays the values of all elements. For an array, the display shows the subscript (index) and value of each array element. For a record, the display shows the name and value of each record component.

To specify an individual array element, array slice, or record component, use the syntax of the current language.

If you specify a range of entities, 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. The debugger displays the entity specified by the first address expression, the logical successor of that address expression, the next logical successor, and so on, until it displays the entity specified by the last address expression. You can specify a list of ranges by separating ranges with a comma.

QUALIFIERS

/ASCIC

Interprets each examined entity as a counted ASCII string preceded by a 1-byte count field that gives the length of the string. The string is then 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 preceded by a 2-byte count field that gives the length of the string. The string is then displayed. /AW is also accepted.

/ASCIZ
Interprets each examined entity as a zero-terminated ASCII string. The trailing zero byte indicates the end of the string. The string is then displayed. /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 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 (length 8 bytes) containing the internal 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, depending on the number of instruction operands and the kind of addressing modes used). See also /OPERANDS.

In screen mode, the output of an EXAMINE/INSTRUCTION command is directed at the current instruction display, not at an output or DO display. The arrow in the instruction display points to the examined instruction.

/[NO]LINE

Controls whether code locations are displayed in terms of line numbers (*%LINE x*) or as *routine-name + 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). This is the default type for program locations that do not have a compiler generated type.

/OCTAL

Displays each examined entity as an octal integer.

/OCTAWORD

Displays each examined entity in the octaword integer type (length 16 bytes).

/OPERANDS[=keyword]

Displays operand information associated with an examined instruction (displays each operand's address and its contents). The level of information displayed depends on whether you use the keyword BRIEF or FULL. The default is /OPERANDS=BRIEF.

Use /OPERANDS only when examining the instruction at the current PC value (for example, EXAMINE/OPERANDS .0\%PC). Examining the operands of an instruction that is not at the current PC value may give erroneous results, because the state of the machine (such as the contents of the registers) is not set up for that instruction.

See also the SET MODE [NO]OPERANDS=keyword command. It enables you to set a default level for the amount of operand information displayed when examining instructions.

/PACKED:n

Interprets each examined entity as a packed decimal number. The value of *n* is the number of decimal digits. Each digit occupies one nibble (4 bits).

/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 (2 bytes) 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. The examined entity must be associated with a machine code instruction and, therefore, must be a line number, a label, a routine name, or the virtual address of an instruction. The examined entity cannot be a variable name or any other address expression that is associated with data.

In screen mode, the output of an EXAMINE/SOURCE command is directed at the current source display, not at an output or DO display. The arrow in the source display points to the source line associated with the last entity specified (or the last one specified in a list of entities).

/[NO]SYMBOLIC

Controls whether symbolization occurs. By default (/SYMBOLIC), the debugger symbolizes all addresses, if possible; that is, it converts numeric addresses into their symbolic representation. If you specify /NOSYMBOLIC, the debugger suppresses symbolization of entities you specify as absolute addresses. If you specify entities as variable names, symbolization still occurs. /NOSYMBOLIC is useful if you are interested in identifying numeric addresses rather than their symbolic names (if symbolic names exist for those addresses). If you specify /NOSYMBOLIC, command processing may speed up somewhat, because the debugger does not need to convert numbers to names.

/TASK

Note: This qualifier 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 and displays each examined entity according to the type specified by *type-expression* (the name of a variable or data type declared in the program). This enables you to specify a user-declared type.

/WORD

Displays each examined entity in the word integer type (length 2 bytes).

DESCRIPTION

The EXAMINE command displays the entity at the location denoted by an address expression. The command may be used to display the contents of any virtual memory location or register that is accessible in your program. For high-level languages the command is used mostly to obtain the current value of a variable (an integer, real, string, array, record, and so on).

The debugger recognizes the compiler generated types associated with symbolic address expressions (symbolic names declared in your program). Symbolic address expressions include the following:

- Variable names. When specifying a variable with the EXAMINE command, use the same syntax that is used in the source code.
- Routine names, labels, and line numbers. These are associated with VAX instructions. You can examine instructions using the same techniques as when examining variables.

EXAMINE

In general, when you enter an EXAMINE command, the debugger evaluates the address expression specified to yield a program location. The debugger then displays the value stored at that location as follows:

- If the location has a symbolic name, the debugger formats the value according to the compiler generated type associated with that symbol — that is, as a variable of a particular type or as a VAX instruction.
- If the location does not have a symbolic name (and, therefore, no associated compiler generated type) the debugger formats the value in the type *longword integer*, by default. This means that, by default, the EXAMINE command displays the contents of these locations as longword (4-byte) integer values.

There are several ways of changing the type associated with a program location so that you can display the data at that location in another data format:

- To change the default type for all locations that do *not* have a symbolic name, you can specify a new type with the SET TYPE command.
- To change the default type for *all* locations (both those that do and do not have a symbolic name), you can specify a new type with the SET TYPE/OVERRIDE command.
- To override the type currently associated with a particular location for the duration of a single EXAMINE command, you can specify a new type by means of a type qualifier (/ASCII:*n*, /BYTE, /TYPE=(*type-expression*), and so on). Most of the EXAMINE command qualifiers are type qualifiers.

The debugger can interpret and display integer data in any one of four radixes: binary, decimal, hexadecimal, and octal. The default radix for both data entry and display is decimal for all languages except BLISS and MACRO. It is hexadecimal for BLISS and MACRO. The EXAMINE command has four radix qualifiers (/BINARY, /DECIMAL, /HEXADECIMAL, /OCTAL) that enable you to display integer data in another radix. You can also use the SET RADIX and SET RADIX/OVERRIDE commands to change the default radix.

In addition to the type and radix qualifiers, the EXAMINE command has qualifiers for other purposes:

- The /SOURCE qualifier enables you to identify the line of source code corresponding to a line number, routine name, label, or any other address expression that is associated with an instruction rather than data.
- The /[NO]LINE and /[NO]SYMBOL qualifiers enable you to control the symbolization of address expressions.

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.

Related commands: DEPOSIT, EVALUATE, (SET, SHOW, CANCEL) RADIX, (SET, SHOW) TYPE, CANCEL TYPE/OVERRIDE, SET MODE [NO]OPERANDS, SET MODE [NO]SYMBOLIC.

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
module MAIN
    47:  procedure SWAP(X,Y: in out INTEGER) is
```

This command displays the source line in which routine SWAP is declared (the location of routine SWAP).

7
```
DBG> DEPOSIT/ASCII:7 WORK+20 = 'abcdefg'
DBG> EXAMINE/ASCII:7 WORK+20
DETAT\WORK+20:  "abcdefg"
DBG> EXAMINE/ASCII:5 WORK+20
DETAT\WORK+20:  "abcde"
```

In this example, the DEPOSIT command deposits the entity 'abcdefg' as an ASCII string of length 7 bytes into the location that is 20 bytes beyond the location denoted by the symbol WORK. The first EXAMINE command displays the value of the entity at that location as an ASCII string of length 7 bytes (abcdefg). The second EXAMINE command displays the value of the entity at that location as an ASCII string of length 5 bytes (abcde).

8
```
DBG> EXAMINE/INST MAIN+2
MAIN\MAIN+02:  MOVAL   L^MAINA,R11
```

This command displays the contents of the location that is 2 bytes beyond the location denoted by the symbol MAIN as an instruction (MOVAL).

9
```
DBG> EXAMINE/OPERANDS=FULL .0\%PC
X\X$START+0C:   MOVL    B^04(R4),R7
     B^04(R4)   R4 contains X\X$START\M (address 00001054),
                B^04(00001054) evaluates to X\X$START\K
                (address 00001058), which contains 00000016
     R7         R7 contains 00000000
```

This command displays the instruction (MOVL) at the current PC value. The /OPERANDS qualifier with the keyword FULL displays the maximum level of operand information.

10
```
DBG> SET RADIX HEXADECIMAL
DBG> EVALUATE/ADDRESS WORKDATA
0000086F
DBG> EXAMINE/SYMBOLIC 0000086F
MOD3\WORKDATA:  03020100
DBG> EXAMINE/NOSYMBOLIC 0000086F
0000086F:  03020100
```

In this example, the EVALUATE/ADDRESS command indicates that the virtual address of variable WORKDATA is 0000086F, hexadecimal. The two EXAMINE commands display the value contained at that address using the /[NO]SYMBOL qualifier to control whether the address is symbolized to WORKDATA.

11
```
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.

12
```
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.

13
```
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 enter 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), and control is returned to the command interpreter. You cannot then continue to debug your program by entering 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 are triggered 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 one or more enclosing FOR, REPEAT, or WHILE loops.

FORMAT

EXITLOOP *[n]*

PARAMETERS

n
A decimal integer that specifies the number of nested loops to exit from. The default is 1.

QUALIFIERS

None.

DESCRIPTION

Use the EXITLOOP command to exit one or more enclosing FOR, REPEAT, or WHILE loops.

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 (FORTRAN example).

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.

You must specify at least one qualifier.

QUALIFIERS

/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.

EXPAND

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 causes 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`

This command moves the right border of the current scrolling display to the right by 6 columns.

2 `DBG> EXPAND/UP/RIGHT:-12 OUT2`

This 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`

This 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 recreate the current screen state at a later time.

FORMAT

EXTRACT *[disp-name[, . . .]] [file-spec]*

PARAMETERS

disp-name

Specifies a display to be extracted. You may specify any of the following:

- A predefined display: SRC, OUT, PROMPT, INST, REG
- A display previously created with the SET DISPLAY command

You can use the asterisk wildcard character (*) in a display name. Do not specify a display name with /ALL.

file-spec

Specifies the file to which the information is 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. Do not specify a display name with /ALL. 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.

EXTRACT

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.

FOR

EXAMPLES

1
```
DBG> FOR I = 10 TO 1 BY -1 DO (EXAMINE A(I))
```

This command examines an array backwards.

2
```
DBG> FOR I = 1 TO 10 DO (DEPOSIT A(I) = 0)
```

This command 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 startup, at the image 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, during a debugging session you can restart execution at the beginning of the program by entering the command GO %LINE 1. However, because the program has executed, the contents of some variables may now be initialized differently from when you first invoked the debugger.

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.

GO

```
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 asterisk 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 predefined keypad-key functions, 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 causes 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.

You must specify at least one qualifier.

QUALIFIERS

/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`

This command moves the current scrolling display to the left by 1 column.

2 `DBG> MOVE/UP:3/RIGHT:5 NEW_OUT`

This 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 enter 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), and control is returned to the command interpreter. You cannot then continue to debug your program by entering 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 entered from the 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 (EXAMINE Y; STEP)
```

This command line sets up a loop that issues a sequence of two commands (EXAMINE Y then 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 are to be saved. 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

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 enables you to save a "snapshot" copy of an existing display 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, you can also 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	Uses the current scope to find a module and searches that module from the specified line number to the end of the module. The current scope is that established by a previous SET SCOPE command, or the PC scope if no SET SCOPE command was entered. If you specify a scope search list with the SET SCOPE command, the debugger searches only the module associated with the first named scope.
line-num:line-num	Uses the current scope to find a module and searches that module from the line number specified on the left of the colon to the line number specified on the right. The current scope is that established by a previous SET SCOPE command, or the PC scope if no SET SCOPE command was entered. If you specify a scope search list with the SET SCOPE command, the debugger searches only the module associated with the first named scope.
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 (') under the following conditions:

- 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 two consecutive quotation marks ("") to indicate an enclosed quotation mark. If the string is enclosed in apostrophes, use two consecutive apostrophes (') to indicate an enclosed apostrophe.

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 next 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

The SEARCH command displays the lines of source code that contain an occurrence of a specified string.

When specifying a module name with the SEARCH command, note that the module must be set. Use the SHOW MODULE command to determine whether a particular module is set. Then use the SET MODULE command, if necessary.

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).

If you plan to enter 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, the module that is in the current scope.

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.

5
```
DBG> SEARCH 43 D
module COBOLTEST
    47: 02      DR0     COMP-2 VALUE  0.1.
```

This command searches for the next occurrence (by default) of the letter D, starting with line 43.

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 VAXstations, 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`

This command selects display SRC2 as the current source and scrolling display.

2 `DBG> SELECT/INPUT/ERROR OUT`

This 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`

This command unselects (deletes the source attribute from) the currently selected source display. The output of a TYPE or EXAMINE/SOURCE command then goes 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 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 enables you to override this default.

Related commands: @file-spec, SHOW ATSIGN.

EXAMPLES

1

```
DBG> SET ATSIGN USER:[JONES.DEBUG].DBG
DBG> @TEST
```

In this example, when the user invokes @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, at instructions of a particular class, or at the occurrence of events.

FORMAT

SET BREAK *[address-expression[, . . .]]*
*[***WHEN***(conditional-expression)]*
*[***DO***(command[; . . .])]*

PARAMETERS

address-expression

Specifies an address expression (a program location) at which a breakpoint is to be set. With high-level languages, this is typically a line number, a routine name, or a label, and may include a path name to specify the entity uniquely. More generally, an address expression may also be a virtual memory address or a register and may be composed of numbers (offsets) and symbols, as well as one or more operators, operands, or delimiters. Appendix D identifies the operators that may be used in address expressions.

Do not specify the asterisk wildcard character (*). Do not specify an address expression with /BRANCH, /CALL, /EXCEPTION, /INSTRUCTION[=(opcode-list)], /INTO, /[NO]JSB, /LINE, /OVER, /[NO]SHARE, or /[NO]SYSTEM. The /MODIFY and /RETURN qualifiers are used with specific kinds of address expressions.

If you specify a virtual memory address or an address expression whose value is not a symbolic location, check (with the EXAMINE command) 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. When you set a breakpoint by specifying an address expression whose value is not a symbolic location, the debugger does not verify that the location specified marks the beginning of an instruction. CALLS and CALLG routines start with an entry mask.

command

Specifies a debugger command that is to be executed as part of the DO clause when break action is taken.

conditional-expression

Specifies a conditional expression in the currently set language that is to be evaluated when execution reaches the breakpoint. 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 (if specified) 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 during execution (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). Do not specify an address expression with /BRANCH. 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 /CALL. See also /INTO, /OVER.

/EVENT=event-name

Note: This qualifier applies only to Ada and SCAN. See the VAX Ada and VAX SCAN documentation for complete information.

Causes the debugger to break on the specified event (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 whenever 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. You can display the event names associated with the current run-time facility by entering the SHOW EVENT_FACILITY command. Note that you cannot specify an address expression with certain event names.

Do not specify /EVENT with /BRANCH, /CALL, /EXCEPTION, /INSTRUCTION[=(opcode-list)], /INTO, /[NO]JSB, /LINE, /MODIFY, /OVER, /RETURN, /[NO]SHARE, or /[NO]SYSTEM.

/EXCEPTION

Causes the debugger to break whenever an exception is signaled. The break action occurs before any user-written exception handlers are invoked. Do not specify an address expression with /EXCEPTION.

As a result of a SET BREAK/EXCEPTION command, whenever your program generates an exception condition, the debugger suspends program execution, reports the exception condition, and displays its prompt. When you resume execution from an exception breakpoint, the behavior is as follows:

- If you enter a GO command without an address-expression parameter, the exception is resignalled, thus allowing any user-declared exception handler to execute.
- If you enter a GO command with an address-expression parameter, program execution continues at the specified location, thus inhibiting the execution of any user-declared exception handler.
- If you enter a STEP command, the debugger steps into any user-declared exception handler. If there is no user-declared handler for that exception, the debugger resignals the exception.

- If you enter a CALL command, the routine specified is executed. If a routine is called with the CALL command directly after an exception breakpoint has been triggered, no breakpoints, tracepoints, or watchpoints set within that routine are triggered. However, they are triggered if the CALL command is given at another time.

/INSTRUCTION

Causes the debugger to break on every instruction executed. Do not specify an address expression with /INSTRUCTION. 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 /INSTRUCTION. See also /INTO, /OVER.

/INTO

Applies only to breakpoints set with /BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE; that is, when an address expression is not explicitly specified. When used with those qualifiers, causes the debugger to break at the specified points within called routines (as well as within the routine where execution is currently suspended). /INTO is the default behavior and is the opposite of /OVER.

When using /INTO, you can further qualify the break action with the /[NO]JSB, /[NO]SHARE, and /[NO]SYSTEM qualifiers.

/[NO]JSB

Qualifies /INTO. Use /[NO]JSB only with /INTO and one of the following qualifiers: /BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE. /JSB is the default for all languages except DIBOL. /JSB permits the debugger to 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 /[NO]JSB.

/LINE

Causes the debugger to break at the start of each new line. Do not specify an address expression with /LINE. 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 (SET TYPE WORD, which changes the default length to 2 bytes) or BYTE (SET TYPE BYTE, which changes the default length to 1 byte). SET TYPE LONGWORD restores the default length of 4 bytes.

/OVER

Applies only to breakpoints set with /BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE; that is, when an address expression is not explicitly specified. When used with those qualifiers, causes the debugger to break at the specified points only within the routine where execution is currently suspended (not within called routines). /OVER is the opposite of /INTO (the default behavior).

/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 deletes 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.

A SET BREAK/RETURN command cancels a previous SET BREAK command if the same address expresion is specified.

/[NO]SHARE

Qualifies /INTO. Use /[NO]SHARE only with /INTO and one of the following qualifiers: BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE. /SHARE (default) permits the debugger to 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 /[NO]SHARE.

/[NO]SILENT

Controls whether or not the "break . . . " message and source code are 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. See also SET STEP [NO]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 only with /INTO and one of the following qualifiers: /BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE. /SYSTEM (default) permits the debugger to 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 /[NO]SYSTEM.

/TEMPORARY

Causes the breakpoint to disappear after it is triggered (the breakpoint does not remain permanently set).

SET BREAK

DESCRIPTION

When a breakpoint is triggered, the debugger does the following:

1. Suspends program execution at the breakpoint location.
2. If /AFTER was specified when the breakpoint was set, checks the AFTER count. If the specified number of counts has not been reached, execution is resumed and the debugger does not perform the remaining steps.
3. 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 is resumed and the debugger does not perform the remaining steps.
4. Reports that execution has reached the breakpoint location, unless /SILENT was specified.
5. Displays the line of source code where execution is suspended, unless /NOSOURCE or /SILENT was specified when the breakpoint was set, or SET STEP NOSOURCE was entered previously.
6. 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.
7. Issues the 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

The /LINE qualifier sets a breakpoint on each line of source code.

The following qualifiers set breakpoints on classes of instructions:

/BRANCH
/CALL
/INSTRUCTION
/INSTRUCTION=(opcode-list)
/RETURN

The following qualifiers set breakpoints on classes of events:

/EVENT=event-name
/EXCEPTION

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).

If you set a breakpoint at a location currently used as a tracepoint, the tracepoint is canceled in favor of the breakpoint, and vice versa.

Related commands: (SHOW, CANCEL) BREAK, CANCEL ALL, SET TRACE, SET WATCH, GO, STEP, (SET, SHOW) EVENT_FACILITY, SET STEP [NO]SOURCE.

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 triggers 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 triggered, 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 are treated as DEFINE/ADDRESS. This is the default.
COMMAND	Subsequent DEFINE commands are treated as DEFINE/COMMAND.
VALUE	Subsequent DEFINE commands are 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
```

This 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** *w-spec*] [*disp-kind*] [*, . . .]*

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 (w-spec) and display kind (disp-kind).

w-spec

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 *w-spec* parameter, the display is positioned at window H1 or H2 by default, alternating between H1 and H2 with each SET DISPLAY command.

disp-kind

Specifies the display kind. Valid keywords are the following:

DO (command[; . . .])	Specifies an automatically updated output display. The commands are executed in the order listed each time the debugger gains control. Their output forms the contents of the display. If you specify more than one command, they must be separated by semicolons.
INSTRUCTION	Specifies an instruction display. If selected as the current instruction display with the SELECT/INSTRUCTION command, it displays 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 displays any debugger output that is not directed to another display. If selected as the current input display with the SELECT/INPUT command, it echoes debugger input. If selected as the current error display with the SELECT/ERROR command, it displays 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 displays 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 *disp-kind* parameter, an OUTPUT display is created.

QUALIFIERS

/[NO]DYNAMIC

Controls whether a display automatically adjusts its window dimensions in proportion when the screen height or width is changed by a SET TERMINAL command. By default (/DYNAMIC) all newly created displays adjust their window dimensions automatically.

/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. You can specify the display's 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 does 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
```

In this example, 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
```

In this example, 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.

3
```
DBG> SET DISPLAY CALLS AT Q3 DO (SHOW CALLS)
```

This command creates a DO display named 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 display CALLS, replacing any previous contents.

SET EDITOR

Establishes the editor that is 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 enter 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 "TPU".

/[NO]START_POSITION

Note: Currently, only VAXTPU and the VAX Language-Sensitive Editor (specified either as TPU or /CALLABLE_TPU, and LSEDIT or /CALLABLE_LSEDIT, respectively) supports /START_POSITION.

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 the 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.

SET EDITOR

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 is executed by the callable editor.

Related commands: SHOW EDITOR, EDIT, (SET, SHOW, CANCEL) SOURCE.

EXAMPLES

1

```
DBG> SET EDITOR '@MAIL$EDIT ""'
```

This 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
```

This command causes the EDIT command to invoke callable VAXTPU with the default command line of TPU.

3

```
DBG> SET EDITOR/CALLABLE_TPU TPU/SECTION=MYSECINI.TPU$SECTION
```

This command causes the EDIT command to invoke callable VAXTPU with the command line TPU/SECTION=MYSECINI.TPU$SECTION.

4

```
DBG> SET EDITOR/CALLABLE_LSEDIT/START_POSITION
```

This command causes the EDIT command to invoke callable LSEDIT with the default command line of LSEDIT. Also the /START_POSITION qualifier is appended to the command line, so that the editing session starts 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: This 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 enables you to set breakpoints and tracepoints on tasking events and exception events. The SCAN event facility enables you to 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.

EXAMPLE

```
DBG> SET EVENT_FACILITY ADA
```

This command establishes Ada as the current run-time library facility.

SET IMAGE

Loads symbol information 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". Do not use the asterisk wildcard character (*). Do not specify an image name with /ALL.

QUALIFIERS

/ALL

Specifies that all shareable images are to be set. Do not specify an image with /ALL.

DESCRIPTION

The SET IMAGE command builds data structures for one or more specified images but does not set any modules within the images specified.

The "current" image is the current debugging context: you have access to symbols in 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 /TRACEBACK qualifier on the LINK command. If an image was linked /NOTRACEBACK, no symbol information is available for that image and you cannot specify it with the SET IMAGE command.

Definitions created with the DEFINE/ADDRESS and DEFINE/VALUE commands are available only when the image in whose context they were created is the current image. When you use the SET IMAGE command to establish a new current image, these definitions are temporarily unavailable. Definitions created with the DEFINE/COMMAND and DEFINE/KEY commands are always available for all images, however.

Related commands: (SHOW, CANCEL) IMAGE, (SET, SHOW, CANCEL) MODULE, SET MODE [NO]DYNAMIC.

EXAMPLE

```
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 is not processed. The SET KEY/STATE command enables you to 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
```

This command changes the key state to the PROG3 state. The user 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 ADA, BASIC, BLISS, C, COBOL, DIBOL, FORTRAN, MACRO, PASCAL, PLI, RPG, SCAN, and UNKNOWN.

QUALIFIERS

None.

DESCRIPTION

When you invoke the debugger, the debugger sets the current language to that in which the module containing the main program is written. This is usually the module containing the image transfer address. 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 setting determines how the debugger parses and interprets the names, operators, and expressions you specify in debugger commands, including things like the typing of variables, array and record syntax, the default radix for the entry and display of integer data, case sensitivity, and so on. The language setting also determines how the debugger formats and displays data associated with your program.

The default radix for both data entry and display is decimal for all languages except BLISS and MACRO. It is hexadecimal for BLISS and MACRO. The default type for program locations that do not have a compiler generated type is longword integer.

The SET LANGUAGE UNKNOWN command may be used when debugging a program that is written in an unsupported language. To maximize the usability of the debugger with unsupported languages, the SET LANGUAGE UNKNOWN command causes the debugger to accept a large set of data formats and operators, including some that may be specific to only a few supported languages.

The operators and constructs that are recognized for each value of the SET LANGUAGE command are identified in Appendix E.

Related commands: SHOW LANGUAGE, SET TYPE, SET RADIX, SET MODE, DEPOSIT, EXAMINE, EVALUATE.

SET LANGUAGE

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 entered.

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 entered 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: SHOW LOG, SET OUTPUT LOG, SET OUTPUT SCREEN_LOG.

EXAMPLES

1

```
DBG> SET LOG CALC
DBG> SET OUTPUT LOG
```

In this example, the SET LOG command specifies the debugger log file to be SYS$DISK:[ ]CALC.LOG. The SET OUTPUT command causes user input and debugger output to be logged to that file.

2

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

In this example, the SET LOG command specifies the debugger log file to be [CODEPROJ]FEB29.TMP. The SET OUTPUT command causes user input and debugger output to be logged to that file.

SET MARGINS

Specifies the leftmost and rightmost source-line character position at which to begin and 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

The SET MARGINS command affects only the display of source lines. It does not affect the display of other debugger output, as from an EXAMINE command.

The SET MARGINS command is useful for controlling the display of source code when, for example, the code is deeply indented or long lines wrap at the right margin. In such cases, you can set the left margin to eliminate indented space in the source display, and you can decrease the right margin setting (from its default value of 255) to truncate lines and prevent them from wrapping.

The SET MARGINS command is useful mostly in line (noscreen) mode. In line mode, the SET MARGINS command affects the display of source lines resulting from a TYPE, EXAMINE/SOURCE, SEARCH, or STEP command, or when a breakpoint, tracepoint, or watchpoint is triggered.

In screen mode, the SET MARGINS command has no effect on the display of source lines in a source display, such as the predefined display SRC. Therefore it does not affect the output of a TYPE or EXAMINE/SOURCE command, since that output is directed at a source display. The SET MARGINS command affects only the display of any source code that might appear in an output or DO display (for example after a STEP command has been executed). However, note that such display is normally suppressed if you invoke screen mode with the keypad key sequence PF1-PF3, because that sequence issues the command SET STEP NOSOURCE in addition to SET MODE SCREEN, to eliminate redundant source display.

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.

Related commands: SHOW MARGINS, SET STEP [NO]SOURCE.

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.

SET MARGINS

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 (*n* is a decimal integer). 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 enter 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 also 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
```

In this example, 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 mode. When dynamic mode is enabled, the debugger sets modules and images automatically during program execution so that you typically do not have to enter the SET MODULE or SET IMAGE command. Specifically, whenever the debugger interrupts execution (whenever the debugger prompt is displayed), the debugger automatically sets the module and image that contain the routine where execution is currently suspended. If the module or image is already set, dynamic mode has no effect on that module or image. The debugger issues an informational message when it sets a module or image automatically. SET MODE DYNAMIC is the default.
NODYNAMIC	Disables dynamic mode. Because additional memory is allocated when a module or image is set, you may want to disable dynamic mode if performance becomes a problem (you can also free up memory by canceling modules and images with the CANCEL MODULE and CANCEL IMAGE commands). When dynamic mode is disabled, you must set modules and images explicitly with the SET MODULE and SET IMAGE commands.
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.
NOLINE	Specifies that the debugger display code locations in terms of *routine-name + byte-offset* rather than in terms of line numbers.
OPERANDS[=keyword]	Specifies that the EXAMINE command, when used to examine an instruction, display the address and contents of the instruction's operands in addition to the instruction and its operands. The level of information displayed depends on whether you use the keyword BRIEF or FULL. The default is OPERANDS=BRIEF.
NOOPERANDS	Specifies that the EXAMINE command, when used to examine an instruction, display only the instruction and its operands. SET MODE NOOPERANDS is the default.
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 enables you to 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.

SET MODE

SEPARATE	(Applies only to VAXstations.) Specifies that a separate window be created for debugger input and output. This is useful when debugging screen-oriented applications, since it moves all debugger displays out of the original window (where the program is running). The separate window is created with a height of 24 lines and a width of 80 columns wide, emulating a VT-series terminal screen.
NOSEPARATE	(Applies only to VAXstations.) Specifies that no separate window be created for debugger input and output. SET MODE NOSEPARATE is the default.
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 numbers to names). This is useful if you are interested in identifying numeric addresses rather than their symbolic names (if symbolic names exist for those addresses). When symbolic mode is disabled, command processing may speed up somewhat, because the debugger does not need to convert numbers to names. EXAMINE/SYMBOLIC may be used to override SET MODE NOSYMBOLIC for the duration of an EXAMINE command.

QUALIFIERS

None.

DESCRIPTION

See the parameter descriptions for details about the SET MODE command. The default values of these modes are the same for all languages.

Related commands: (SHOW, CANCEL) MODE, (SET, SHOW, CANCEL) MODULE, (SET, SHOW, CANCEL) IMAGE, (SET, SHOW) TYPE, EXAMINE, DEPOSIT, EVALUATE, DEFINE/KEY, SYMBOLIZE, DISPLAY, SET PROMPT, (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 in the current image into the run-time symbol table (RST) of that image.

FORMAT

SET MODULE *[module-name[, . . .]]*

PARAMETERS

module-name

Specifies a module of the current image whose symbol records are to be loaded into the RST. Do not use the asterisk wildcard character (*). Do not specify a module name with /ALL.

QUALIFIERS

/ALL

Specifies that the symbol records of all modules in the current image be loaded into the RST. Do not specify a module name with /ALL.

/CALLS

Sets all the modules that currently have routines on the call stack. If a module is already set, /CALLS has no effect on that module.

/[NO]RELATED

Note: This qualifier 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 that you can reference them in debugger commands exactly as they can be referenced 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).

DESCRIPTION

Note: The current image 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 run-time symbol table (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 startup, the debugger sets the module containing the transfer address (the main program). By default, dynamic mode is enabled (SET MODE DYNAMIC). Therefore, the debugger sets modules (and images) 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 and image containing the routine where execution is suspended. 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 mode makes accessible most of the symbols you might need to reference. If you need to reference a symbol in a module that is not already set, proceed as follows:

- If the module is in the current image, use the SET MODULE command to set the module where the symbol is defined.
- If the module is in another image, use the SET IMAGE command to make that image the current image, then use the SET MODULE command to set the module where the symbol is defined.

If dynamic mode is disabled (SET MODE NODYNAMIC), only the module containing the transfer address is set automatically. You must set any other modules explicitly.

If you use the SET IMAGE command to establish a new current image, all modules previously set remain set. However, only the symbols in the set modules of the current image are accessible. Symbols in the set modules of other images are temporarily unaccessible.

When dynamic mode is enabled, memory is allocated automatically to accommodate the increasing size of the RST. If dynamic mode is disabled, the debugger automatically allocates more memory as needed when you set a module or an image. Whether dynamic mode 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 a 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.

EXAMPLES

1
```
DBG> SET MODULE SUB1
```

This command sets module SUB1 (loads the symbol records of module SUB1 into the RST).

2
```
DBG> SET IMAGE SHARE3
DBG> SET MODULE MATH
DBG> SET BREAK %LINE 31
```

In this example, 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.

```
3  DBG> SHOW MODULE/SHARE
   module name                    symbols   language   size

   FOO                            yes       MACRO       432
   MAIN                           no        FORTRAN     280
       .
       .
       .
   SHARE$DEBUG                    no        Image         0
   SHARE$LIBRTL                   no        Image         0
   SHARE$MTHRTL                   no        Image         0
   SHARE$SHARE1                   no        Image         0
   SHARE$SHARE2                   no        Image         0

   total modules: 17.             bytes allocated: 162280.
   DBG> SET MODULE SHARE$SHARE2
   DBG> SHOW SYMBOL * IN SHARE$SHARE2
```

In this example, the SHOW MODULE/SHARE command identifies all of the modules in the current image and all of the shareable images (the names of the shareable images are prefixed with "SHARE$"). The command SET MODULE SHARE$SHARE2 sets the shareable image module SHARE$SHARE2. The SHOW SYMBOL command identifies any universal symbols defined in the shareable image SHARE2. See the description of the /SHARE qualifier of the SHOW MODULE command for more information.

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 (VERIFY).
- Record debugger output and user input in a log file (LOG).
- Not display output at the terminal, except for diagnostic messages (NOTERMINAL).

SET PROMPT

Changes the debugger prompt string from DBG> to a string of your choice.

FORMAT

SET PROMPT *[prompt-string]*

PARAMETERS

prompt-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 quotation marks (") or apostrophes ('). By default, the prompt string is DBG> . If you do not specify a string, the current prompt string remains unchanged.

QUALIFIERS

/[NO]POP

Note: This qualifier applies only to VAXstations.

/POP causes the debugger window to pop over other windows and become attached to the keyboard when the debugger prompts for input. /NOPOP disables this behavior and is the default (the debugger window is not popped over other windows and is not attached to the keyboard automatically when the debugger prompts for input).

If you do not specify /POP or /NOPOP, the prompt behavior is set to /NOPOP

DESCRIPTION

The SET PROMPT command enables you to tailor the debugger prompt to your individual preference.

EXAMPLE

```
DBG> SET PROMPT "$ "
$ SET PROMPT "d b g : "
d b g : SET PROMPT "DBG> "
DBG>
```

In this example, the successive SET PROMPT commands change the debugger prompt from "DBG> " to "$", to "d b g :", then back to "DBG> ".

SET RADIX

Establishes the radix for the entry and 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 radix to be established. Valid keywords are the following:

BINARY	Sets the radix to binary.
DECIMAL	Sets the radix to decimal. This is the default for all languages except BLISS and MACRO.
DEFAULT	Sets the radix to the language default.
OCTAL	Sets the radix to octal.
HEXADECIMAL	Sets the default radix to hexadecimal. This is the default for BLISS and MACRO.

QUALIFIERS

/INPUT

Sets only the input radix (the radix for entering integer data) to the specified radix.

/OUTPUT

Sets only the output radix (the radix for displaying integer data) to the specified radix.

/OVERRIDE

Causes all data to be displayed as integer data of the specified radix.

DESCRIPTION

The current radix setting influences how the debugger interprets and displays integer data in the following contexts:

- Integer data that you specify in address expressions or language expressions.
- Integer data that is displayed by the commands EXAMINE and EVALUATE.

The default radix for both data entry and display is decimal for all languages except BLISS and MACRO. It is hexadecimal for BLISS and MACRO.

The SET RADIX command enables you to specify a new radix for data entry or display (the input radix and output radix, respectively).

If you do not specify a qualifier, the SET RADIX command changes both the input and output radix. If you specify the /INPUT or /OUTPUT qualifier, the command changes the input or output radix, respectively.

SET RADIX

If you specify the /OVERRIDE qualifier, the SET RADIX command changes only the output radix but causes *all* data (not just data that has an integer type) to be displayed as integer data of the specified radix.

Note that, except when used with the /OVERRIDE qualifier, the SET RADIX command does not affect the interpretation or display of non-integer values (such as real or enumeration type values).

The EVALUATE, EXAMINE, and DEPOSIT commands have radix qualifiers (/BINARY, /HEXADECIMAL, and so on) that enable you to override, for the duration of that command, any radix previously established with the SET RADIX or SET RADIX/OVERRIDE command.

You can also use the built-in symbols %BIN, %DEC, %HEX, and %OCT in address expressions and language expressions to specify that an integer literal that follows should be interpreted in binary, decimal, hexadecimal, or octal radix, respectively (see Appendix D).

Related commands: (SHOW, CANCEL) RADIX, (SET, SHOW, CANCEL) MODE, EVALUATE, EXAMINE, DEPOSIT.

EXAMPLES

1 `DBG> SET RADIX HEX`

This command sets the radix to hexadecimal. This means that, by default, integer data is interpreted and displayed in hexadecimal radix.

2 `DBG> SET RADIX/INPUT OCT`

This command sets the radix for input to octal. This means that, by default, integer data that is entered is interpreted in octal radix.

3 `DBG> SET RADIX/OUTPUT BIN`

This command sets the radix for output to binary. This means that, by default, integer data is 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 (not just data that has an integer type) is 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 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 denoted by the routine which is *n* levels down the call stack (*n* is a decimal integer). A scope 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 — 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, 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 the current PC value and changes dynamically as the 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) — that is, all set modules and the global symbol table (GST), if necessary.

The SET SCOPE command enables you to 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, SHOW SYMBOL, SYMBOLIZE.

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
```

In this example, 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 0, STACKS\R2, SCREEN_IO
```

This command tells the debugger to look for symbols without path name prefixes according to the following scope search list. First the debugger looks in the PC scope (denoted by "0"). If the debugger cannot find a specified symbol in the PC scope, it then looks in routine R2 of module STACKS; if necessary, it then looks in module SCREEN_IO. If the debugger still cannot find a specified symbol, 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
```

In this example, 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 are treated as SEARCH/ALL, rather than SEARCH/NEXT.
IDENTIFIER	Subsequent SEARCH commands are treated as SEARCH /IDENTIFIER, rather than SEARCH/STRING.
NEXT	Subsequent SEARCH commands are treated as SEARCH/NEXT, rather than SEARCH/ALL. This is the default.
STRING	Subsequent SEARCH commands are 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
```

In this example, 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 that have been moved to another directory after being compiled.

FORMAT

SET SOURCE *directory-spec[, . . .]*

PARAMETERS

directory-spec

Specifies any part of a 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: This qualifier applies mainly to Ada programs.

Specifies that the directory search list is used to locate source files for editing when you use the EDIT command.

/MODULE=module-name

Specifies that the directory search list is 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, use the SET SOURCE command to specify a source directory search list.

When a source file is moved to another directory, the version number of the source file may change. To locate the correct version of the source file in the event that a version number was not specified in the directory-spec parameter, the debugger inserts the match-all asterisk wildcard character (*) in the version number field of the new file specification. Therefore, 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. If the debugger does not find the correct version, it uses the file that has the closest revision

date and time (if such a file exists in that directory) and issues a message such as the following when first displaying source code:

```
%DEBUG-I-NOTORIGSRC, original version of source file not found
        file used is WORK:[JONES.PROG3]PRG.FOR;14
```

If you enter 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.

A full VMS file specification consists of the following fields:

node::device:[directory]file-name.file-type;version-number

If the full file specification of a source file exceeds 231 characters, the debugger cannot locate the file. You can work around this problem by first defining a logical name "X" (at DCL level) to expand to your long file specification, and then using the command "SET SOURCE X".

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]
```

In this example, the SET SOURCE command specifies that the debugger should search directories [PROJA], [PROJB], and USER$:[PETER.PROJC], in that order, for source files.

2

```
DBG> SET SOURCE/MODULE=COBOLTEST [], DISK$2:[PROJD]
DBG> SHOW SOURCE
source directory search list for COBOLTEST:
        []
        DISK$2:[PROJD]
source directory search list for all other modules:
        [PROJA]
        [PROJB]
        USER$:[PETER.PROJC]
```

In this example, the SET SOURCE command specifies that the debugger should search the current default directory ([]) and DISK$2:[PROJD], in that order, 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 are treated as STEP/BRANCH (step to the next branch instruction).
CALL	Subsequent STEP commands are treated as STEP/CALL (step to the next call instruction).
EXCEPTION	Subsequent STEP commands are treated as STEP/EXCEPTION (step to the next exception condition).
INSTRUCTION	Subsequent STEP commands are treated as STEP/INSTRUCTION (step to the next instruction). You can also specify one or more instructions (INSTRUCTION=(opcode-list)). The debugger then steps to the next instruction that is in the specified list.
INTO	Subsequent STEP commands are 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 are 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 are 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 are treated as STEP/LINE (step to the next line). This is the default for all languages.
OVER	Subsequent STEP commands are 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 are treated as STEP/RETURN (step to the RETURN instruction of the current routine). Thus, STEP/RETURN *n* takes you up *n* levels of the call stack.

SHARE	If INTO is in effect, subsequent STEP commands are 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 are treated as STEP/INTO/NOSHARE (step over called routines in shareable images, but step into other routines).
SILENT	Subsequent STEP commands are treated as STEP/SILENT (suppress the "stepped to . . . " message as well as other debugger output).
NOSILENT	Subsequent STEP commands are treated as STEP/NOSILENT (display the "stepped to . . . " message as well as other output). This is the default.
SOURCE	Subsequent STEP commands are treated as STEP/SOURCE (display source code after a step). Also, subsequent SET BREAK, SET TRACE, and SET WATCH commands are treated as SET BREAK/SOURCE, SET TRACE/SOURCE, and SET WATCH/SOURCE, respectively (display source code when a breakpoint, tracepoint, or watchpoint is triggered). This is the default.
NOSOURCE	Subsequent STEP commands are treated as STEP/NOSOURCE (do not display source code after a step). Also, subsequent SET BREAK, SET TRACE, and SET WATCH commands are treated as SET BREAK/NOSOURCE, SET TRACE/NOSOURCE, and SET WATCH/NOSOURCE, respectively (do not display source code when a breakpoint, tracepoint, or watchpoint is triggered).
SYSTEM	If INTO is in effect, subsequent STEP commands are 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 are 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 where the STEP command suspends execution after a 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 [NO]SYSTEM

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.

If you invoke screen mode with the keypad-key sequence PF1-PF3, the command SET STEP NOSOURCE is entered in addition to the command SET MODE SCREEN. Therefore, any display of source code in output and DO displays that would result from a STEP command or from an eventpoint being triggered is suppressed, to eliminate redundancy with the source display.

Related commands: STEP, SHOW STEP.

EXAMPLES

1 `DBG> SET STEP INSTRUCTION,NOSOURCE`

This command causes the debugger to execute the program to the next instruction when a STEP command is entered, and to not display lines of source code with each STEP command.

2 `DBG> SET STEP LINE,INTO,NOSYSTEM,NOSHARE`

This command causes the debugger to execute the program to the next line when a STEP command is entered, and to step into called routines in user space only. The debugger steps 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: This 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 one of the following:

- 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 asterisk 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. 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 runs 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. Do not specify a task nor the /ACTIVE, /VISIBLE, or /TIME_SLICE qualifiers with /ALL.

/[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 or height, or both, that the debugger uses when it formats screen and other output.

FORMAT

SET TERMINAL

PARAMETERS

None.

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.

QUALIFIERS

/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, VT200, or VT300 series terminal, *n* is typically either 80 or 132.

DESCRIPTION

The SET TERMINAL command enables you to define the portion of the screen that the debugger has available for formatting screen output. This command is useful with VT100, VT200, or VT300 series terminals, where you can set the screen width to typically 80 or 132 columns. It is also useful with VAXstations, where you can modify the size of the window that the debugger uses.

When you enter 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 changes 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. If you have specified /NODYNAMIC in a SET DISPLAY or DISPLAY command, the display is no longer dynamic. In that case, the display does 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.

SET TERMINAL

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, at instructions of a particular class, or at the occurrence of specified events.

FORMAT

SET TRACE *[address-expression[, . . .]]*
*[***WHEN***(conditional-expression)]*
*[***DO***(command[; . . .])]*

PARAMETERS

address-expression

Specifies an address expression (a program location) at which a tracepoint is to be set. With high-level languages, this is typically a line number, a routine name, or a label, and may include a path name to specify the entity uniquely. More generally, an address expression may also be a virtual memory address or a register and may be composed of numbers (offsets) and symbols, as well as one or more operators, operands, or delimiters. Appendix D identifies the operators that may be used in address expressions. Do not specify the asterisk wildcard character (*). Do not specify an address expression with /BRANCH, /CALL, /EXCEPTION, /INSTRUCTION[=(opcode-list)], /INTO, /[NO]JSB, /LINE, /OVER, /[NO]SHARE, or /[NO]SYSTEM. The /MODIFY and /RETURN qualifiers are used with specific kinds of address expressions.

If you specify a virtual memory address or an address expression whose value is not a symbolic location, check (with the EXAMINE command) 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. When you set a tracepoint by specifying an address expression whose value is not a symbolic location, the debugger does not verify that the location specified marks the beginning of an instruction. CALLS and CALLG routines start with an entry mask.

command

Specifies a debugger command that is to be executed as part of the DO clause when trace action is taken.

conditional-expression

Specifies a conditional expression in the currently set language that is to be evaluated whenever execution reaches the tracepoint. 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 (if specified) 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 during execution (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). Do not specify an address expression with /BRANCH. 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 /CALL. See also /INTO, /OVER.

/EVENT=event-name

Note: This qualifier applies only to Ada and SCAN. See the VAX Ada and VAX SCAN documentation for complete information.

Causes the debugger to trace the specified event (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 whenever 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. You can display the event names associated with the current run-time facility by entering the SHOW EVENT_FACILITY command. Note that you cannot specify an address expression with certain event names.

Do not specify /EVENT with /BRANCH, /CALL, /EXCEPTION, /INSTRUCTION[=(opcode-list)], /INTO, /[NO]JSB, /LINE, /MODIFY, /OVER, /RETURN, /[NO]SHARE, or /[NO]SYSTEM.

/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 /EXCEPTION.

As a result of a SET TRACE/EXCEPTION command, whenever your program generates an exception condition, the debugger reports the exception condition and resignals the exception, thus allowing any user-declared exception handler to execute.

/INSTRUCTION

Causes the debugger to trace every instruction executed. Do not specify an address expression with /INSTRUCTION. 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 /INSTRUCTION. See also /INTO, /OVER.

/INTO

Applies only to tracepoints set with /BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE; that is, when an address expression is not explicitly specified. When used with those qualifiers, causes the debugger to trace the specified points within called routines (as well as within the routine where execution is currently suspended). /INTO is the default behavior and is the opposite of /OVER.

When using /INTO, you can further qualify the trace action with the /[NO]JSB, /[NO]SHARE, and /[NO]SYSTEM qualifiers.

/[NO]JSB

Qualifies /INTO. Use /[NO]JSB only with /INTO and one of the following qualifiers: /BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE. /JSB is the default for all languages except DIBOL. /JSB permits the debugger to 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 /[NO]JSB.

/LINE

Causes the debugger to trace the start of each new line. Do not specify an address expression with /LINE. 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 (SET TYPE WORD, which changes the default length to 2 bytes) or BYTE (SET TYPE BYTE, which changes the default length to 1 byte). SET TYPE LONGWORD restores the default length of 4 bytes.

/OVER

Applies only to tracepoints set with /BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE; that is, when an address expression is not explicitly specified. When used with those qualifiers, causes the debugger to trace the specified points only within the routine where execution is currently suspended (not within called routines). /OVER is the opposite of /INTO (the default behavior). Sets tracepoints only within the routine where execution is currently suspended (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.

A SET TRACE/RETURN command cancels a previous SET TRACE command if the same address expresion is specified.

/[NO]SHARE

Qualifies /INTO. Use /[NO]SHARE only with /INTO and one of the following qualifiers: BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE. /SHARE (default) permits the debugger to set tracepoints 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 /[NO]SHARE.

/[NO]SILENT

Controls whether or not the "trace . . . " message and source code are 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. See also SET STEP [NO]SOURCE.

/[NO]SYSTEM

Qualifies /INTO. Use /[NO]SYSTEM only with /INTO and one of the following qualifiers: /BRANCH, /CALL, /INSTRUCTION[=(opcode-list)], or /LINE. /SYSTEM (default) permits the debugger to set tracepoints 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 /[NO]SYSTEM.

/TEMPORARY

Causes the tracepoint to disappear after it is triggered (the tracepoint does not remain permanently set).

DESCRIPTION

When a tracepoint is triggered, the debugger takes the following action:

1. Suspends program execution at the tracepoint location.
2. If /AFTER was specified when the tracepoint was set, checks the AFTER count. If the specified number of counts has not been reached, execution is resumed and the debugger does not perform the remaining steps.
3. 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 is resumed and the debugger does not perform the remaining steps.
4. Reports that execution has reached the tracepoint location, unless /SILENT was specified.
5. Displays the line of source code corresponding to the tracepoint, unless /NOSOURCE or /SILENT was specified when the breakpoint was set, or SET STEP NOSOURCE was entered previously.
6. Executes the commands in a DO clause, if one was specified when the tracepoint was set.

7 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

The /LINE qualifier sets a tracepoint on each line of source code.

The following qualifiers set tracepoints on classes of instructions:

/BRANCH
/CALL
/INSTRUCTION
/INSTRUCTION=(opcode-list)
/RETURN

The following qualifiers set tracepoints on classes of events:

/EVENT=event-name
/EXCEPTION

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).

If you set a tracepoint at a location currently used as a breakpoint, the breakpoint is canceled in favor of the tracepoint, and vice versa.

Related commands: (SHOW, CANCEL) TRACE, CANCEL ALL, SET BREAK, SET WATCH, GO, (SET, SHOW) EVENT_FACILITY, SET STEP [NO]SOURCE.

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 triggers only when A equals B. When the tracepoint is triggered, variable Y is examined. The /SILENT qualifier suppresses the "trace . . . " message.

3 DBG> SET TRACE/BRANCH/CALL

This command causes the debugger to trace every BRANCH instruction and every CALL 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 symbolic name (and, therefore, do not have an associated 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 with a 1-byte count field that precedes the string and gives its length. 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 counted ASCII string with a 2-byte count field that precedes the string and gives its 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. The trailing zero byte indicates the end of the string. AZ is also accepted as a keyword.
BYTE	Sets the default type to byte integer (length 1 byte).
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.
DATE_TIME	Sets the default type to date-time. This is a quadword integer (length 8 bytes) containing the internal VMS representation of date-time. Values are displayed in the format *dd-mmm-yyyy hh:mm:ss.xx*. Specify an absolute date and time as follows: `[dd-mmm-yyyy[:]] [hh:mm:ss.cc]`

SET TYPE

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.
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 is the default type for program locations that do not have a symbolic name (do not have a compiler generated type).
OCTAWORD	Sets the default type to octaword integer (length 16 bytes).
PACKED:n	Sets the default type to packed decimal. The value of *n* is the number of decimal digits. Each digit occupies one nibble (4 bits).
QUADWORD	Sets the default type to quadword integer (length 8 bytes).
TYPE=(expression)	Sets the default type to the type denoted by *expression* (the name of a variable or data type declared in the program). This enables you to specify a user-declared type.
WORD	Sets the default type to word integer (length 2 bytes).

QUALIFIERS

/OVERRIDE

Associates the type specified with *all* program locations, whether or not they have a symbolic name (whether or not they have an associated compiler generated type).

DESCRIPTION

When you use the EXAMINE, DEPOSIT, or EVALUATE commands, the default types associated with address expressions influence how the debugger interprets and displays program entities.

The debugger recognizes the compiler generated types associated with symbolic address expressions (symbolic names declared in your program), and it interprets and displays the contents of these locations accordingly. For program locations that do not have a symbolic name and, therefore, no associated compiler generated type, the default type in all languages is longword integer.

The SET TYPE command enables you to change the default type associated with locations that do not have a symbolic name. The SET TYPE/OVERRIDE command enables you to set a default type for *all* program locations, both those that do and do not have a symbolic name.

The EXAMINE and DEPOSIT commands have type qualifiers (/ASCII, /BYTE, /G_FLOAT, and so on) that enable you to override, for the duration of a single command, the type previously associated with *any* program location.

Related commands: SHOW TYPE, CANCEL TYPE/OVERRIDE, (SET, SHOW, CANCEL) RADIX, (SET, SHOW, CANCEL) MODE, EXAMINE, DEPOSIT.

EXAMPLES

1 `DBG> SET TYPE ASCII: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 *address-expression[, . . .]*
*[***WHEN***(conditional-expression)]*
*[***DO***(command[; . . .])]*

PARAMETERS

address-expression

Specifies an address expression (a program location) at which a watchpoint is to be set. With high-level languages, this is typically the name of a program variable and may include a path name to specify the variable uniquely. More generally, an address expression may also be a virtual memory address or a register and may be composed of numbers (offsets) and symbols, as well as one or more operators, operands, or delimiters. Appendix D identifies the operators that may be used in address expressions.

Do not use the asterisk wildcard character (*).

command

Specifies a debugger command that is to be executed as part of the DO clause when watch action is taken.

conditional-expression

Specifies a conditional expression in the currently set language that is to be evaluated whenever execution reaches the watchpoint. 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.

/INTO

Specifies that the debugger is to monitor a nonstatic variable by tracing instructions not only within the defining routine, but also within a routine that is called from the defining routine (and any other such nested calls). SET WATCH/INTO enables you to monitor nonstatic variables within called routines more precisely than SET WATCH/OVER; but the speed of execution within called routines is faster with SET WATCH/OVER.

/OVER

Specifies that the debugger is to monitor a nonstatic variable by tracing instructions only within the defining routine, not within a routine that is called by the defining routine. As a result, the debugger executes a called routine at normal speed and resumes tracing instructions only when execution returns to the defining routine. SET WATCH/OVER provides faster execution than SET WATCH/INTO; but if a called routine modifies the watched variable, execution is interrupted only upon returning to the defining routine. SET WATCH/OVER is the default behavior when you set watchpoints on nonstatic variables.

/[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. See also SET STEP [NO]SOURCE.

/[NO]STATIC

Enables you to override the debugger's default determination of whether a specified variable is static or nonstatic. SET WATCH/STATIC tells the debugger to treat the variable as a static variable. SET WATCH/NOSTATIC tells the debugger to treat the variable as a nonstatic variable. Exercise caution when using this qualifier.

/TEMPORARY

Causes the watchpoint to disappear after it is triggered (the watchpoint does not remain permanently set).

DESCRIPTION

Whenever an instruction causes the modification of a watched location, the debugger does the following:

1. Suspends program execution after that instruction has completed execution.
2. If /AFTER was specified when the watchpoint was set, checks the AFTER count. If the specified number of counts has not been reached, execution continues and the debugger does not perform the remaining steps.
3. 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 remaining steps.
4. Reports that execution has reached the watchpoint location, unless /SILENT was specified.
5. Reports the old (unmodified) value at the watched location.
6. Reports the new (modified) value at the watched location.
7. Displays the line of source code where execution is suspended, unless /NOSOURCE or /SILENT was specified when the watchpoint was set, or SET STEP NOSOURCE was entered previously.

8 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.

9 Issues the prompt.

For high-level language programs, the address expressions you specify with the SET WATCH command are typically variable names. If you specify an absolute memory address that is associated with a a compiler-generated type, the debugger symbolizes the address and uses the length in bytes associated with that type to determine the length in bytes of the watched location. If you specify an absolute memory address that the debugger cannot associate with a compiler-generated type, the debugger watches 4 bytes of virtual memory, by default, beginning at the byte identified by the address expression. You can change this length, however, by setting the type to either WORD (SET TYPE WORD, which changes the default length to 2 bytes) or BYTE (SET TYPE BYTE, which changes the default length to 1 byte). SET TYPE LONGWORD restores the default length of 4 bytes.

You can set watchpoints on aggregates (that is, entire arrays or records). A watchpoint set on an array or record triggers 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.

The following qualifiers affect what output is seen when a watchpoint is reached:

/[NO]SILENT
/[NO]SOURCE

The following qualifiers affect the timing and duration of watchpoints:

/AFTER:n
/TEMPORARY

The following qualifiers apply only to nonstatic variables:

/INTO
/OVER

The following qualifiers are used to override the debugger's determination of whether a variable is static or nonstatic:

/[NO]STATIC

The technique for setting a watchpoint depends on whether the variable is static or nonstatic. A static variable is associated with the same virtual memory address throughout execution of the program. You can always set a watchpoint on a static variable throughout execution.

A nonstatic variable is allocated on the stack or in a register and has a value only when its defining routine is active (on the call stack). Therefore, you can set a watchpoint on a nonstatic variable only when the PC value is within the scope of the defining routine (including any routine called by the definining routine). The watchpoint is cancelled when execution returns from the defining routine.

The debugger determines whether a variable is static or nonstatic by checking how it is allocated. Typically, a static variable is in P0 space (0 through 3FFFFFFF, hexadecimal); a nonstatic variable is in P1 space (40000000 through 7FFFFFFF) or in a register. The debugger issues a warning if you try to set a watchpoint on a variable that is allocated in P1 space or in a register when the PC value is not within the scope of the defining routine. The /[NO]STATIC qualifier enables you to override the default behavior. For example, if you have allocated nonstack storage in P1 space, use the /STATIC qualifier when setting a watchpoint on a variable that is allocated in that storage area.

Another distinction between static and nonstatic watchpoints is speed of execution. To watch a static variable, the debugger write-protects the page containing the variable. If your program attempts to write to that page, an access violation occurs and the debugger handles the exception, determining whether the watched variable was modified. Except when writing to that page, the program executes at normal speed.

To watch a nonstatic variable, the debugger traces every instruction in the variable's defining routine and checks the value of the variable after each instruction has been executed. Since this significantly slows down execution, the debugger issues a message when you set a nonstatic watchpoint. The /INTO and /OVER qualifiers enable you to choose whether to also trace instructions within any routine that is called by the defining routine or to execute the called routine at normal speed.

Related commands: (SHOW, CANCEL) WATCH, SET BREAK, SET TRACE, SET STEP [NO]SOURCE.

EXAMPLES

1

```
DBG> SET WATCH MAXCOUNT
```

This command establishes a watchpoint on the variable MAXCOUNT.

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.

```
3  DBG> SET TRACE SUB2 DO (SET WATCH K)
```

In this example variable K is a nonstatic variable and is defined only when its defining routine, SUB2, is active (on the call stack). The SET TRACE command sets a tracepoint on SUB2. When the tracepoint is triggered during execution, the DO clause sets a watchpoint on K. The watchpoint is then canceled when execution returns from routine SUB2.

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 is 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.

Window definitions are dynamic—that is, window dimensions expand and contract proportionally when a SET TERMINAL command changes the screen width or height.

SET WINDOW

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
```

This example shows that, if the SET ATSIGN command was not used, command procedures are assumed to 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
```

In this example, 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.)

See Section 8.3.2 for information on predefined Ada event breakpoints.

Related commands: (SET, CANCEL) BREAK.

EXAMPLE

```
DBG> SHOW BREAK
breakpoint at SUB1\LOOP
breakpoint at MAIN\MAIN+1F
   do (EX SUB1\D ; EX/SYMBOLIC 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
A decimal integer that specifies the number of call frames to be identified. By default, all currently active call frames are identified.

QUALIFIERS

None.

DESCRIPTION

Whenever a call is made to a routine as your program executes, the VMS operating system creates a separate call frame on the stack. Each call frame stores information about the calling routine. The call frame for the most recently called routine is on the top of the stack.

When a routine returns execution to its caller, the call frame for that routine is removed from the stack.

The SHOW CALLS command shows a traceback that lists the sequence of active routine calls that lead to the routine where execution is currently suspended. Any recursive routine calls are shown in the display, so you can use the SHOW CALLS command to examine the chain of recursion.

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. An asterisk (*) to the left of a module name indicates that the module is set.
- 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.

Note that, even if your program contains no routine calls, the SHOW CALLS command displays an active call. The reason for this is that your program has a stack frame built for it when it is first activated. Thus, if the SHOW CALLS display shows no active calls, either your program has terminated or the stack has been corrupted.

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.

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 entered, 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>
```

In this example, 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 asterisk wildcard character (*) by itself, all display definitions are listed. You can use * within a display name. Do not specify a display name with /ALL.

QUALIFIERS

/ALL

Lists all display definitions. Do not specify a display name with /ALL.

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. In this example, they 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)"
```

This command indicates that, when you enter the EDIT command, you 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 is 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"
```

In this example, the SHOW EDITOR command indicates that, when you enter the EDIT command, you invoke the callable version of the VAX Text Processing Utility (VAXTPU). The editing cursor is 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: This 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.

EXAMPLE

```
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 are 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
```

This 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 asterisk 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:

- Name of the shareable image
- Start and end addresses of the image
- Whether the image has been set with the SET IMAGE command (loaded into the RST)
- Current image that is your debugging context (marked with an asterisk)
- Total number of images selected in the display
- Number of bytes allocated for the RST and other internal structures

Related commands: (SET, CANCEL) IMAGE, (SET, SHOW, CANCEL) MODULE.

EXAMPLE

```
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            bytes allocated: 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 debugger predefined key definitions and those 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 asterisk wildcard character (*). Do not specify a key name with /ALL. Valid key names are the following:

Key-name	LK201 Keyboard	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. Do not specify a key name with /ALL.

/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 command displays all the key definitions for the current state.

2
```
DBG> SHOW KEY/STATE=BLUE KP8
GOLD keypad definitions:
  KP8 = "Scroll/Top" (noecho,terminate,nolock)
```

This command displays the definition for keypad key 8 in the BLUE state.

3
```
DBG> SHOW KEY/BRIEF KP8
DEFAULT keypad definitions:
  KP8 = "Scroll/Up"
```

This command displays the definition for keypad key 8 in the current key state.

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 entered, the current language is, by default, the language of the module containing the main program.

Related commands: SET LANGUAGE.

EXAMPLE

```
DBG> SHOW LANGUAGE
language: BASIC
```

This command displays the name of the current language as BASIC.

SHOW LOG

Indicates whether the debugger is writing to a log file and identifies the current log 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 entered, 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
```

In this example, 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 entered, 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 entered, 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 entered, the current modes are, by default: DYNAMIC, NOG_FLOAT (d_float), KEYPAD, LINE, NOSCREEN, SCROLL, NOSEPARATE, SYMBOLIC.

Related commands: (SET, CANCEL) MODE, (SET, SHOW, CANCEL) RADIX.

EXAMPLE

```
DBG> SHOW MODE
modes: symbolic, line, d_float, screen, scroll, keypad, dynamic, no separate window
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 the modules in the current image.

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 asterisk 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: This qualifier 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 images that have been linked with your program. By default (/NOSHARE) no shareable image modules are selected for display.

The debugger creates dummy modules for each shareable image in your program. The names of these shareable "image modules" have the prefix "SHARE$". SHOW MODULE/SHARE identifies these shareable image modules, as well as the modules in the current image.

Setting a shareable image module loads the universal symbols for that image into the run-time symbol table so that you can reference these symbols from the current image. However, you cannot reference other (local or global) symbols in that image from the current image. Note that this feature overlaps the effect of the newer SET IMAGE and SHOW IMAGE commands.

DESCRIPTION

Note: The current image 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:

- Name of the module.
- Programming language in which the module is coded, unless all modules are coded in the same language.

- Whether or not the module has been set with the SET MODULE command. That is, whether or not the symbol records of the module have been loaded into the debugger's run-time symbol table (RST).
- Space (in bytes) required in the RST for symbol records in that module.
- Total number of modules selected in the display.
- Number of bytes allocated for the RST and other internal structures.

Related commands: (SET, CANCEL) MODULE, (SET, SHOW, CANCEL) IMAGE, SET MODE [NO]DYNAMIC, SHOW SYMBOL, (SET, SHOW, CANCEL) SCOPE.

EXAMPLES

1

```
DBG> SHOW MODULE
module name                     symbols   size

TEST                            yes       432
SCREEN_IO                       no        280

total PASCAL modules: 2.            bytes allocated: 2740.
```

In this example, the SHOW MODULE command, without a parameter specified, displays information about all of the modules in the current image, which is the main image by default. This example shows the display format when all modules have the same source language. The "symbols" column shows that module TEST has been set, but module SCREEN_IO has not.

2

```
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.             bytes allocated: 60720.
```

In this example, the SHOW MODULE command displays information about the modules FOO and MAIN, and all modules having the prefix SUB. This example shows the display format when the modules do not have the same source language.

3

```
DBG> SHOW MODULE/SHARE
module name                     symbols   language   size

FOO                             yes       MACRO       432
MAIN                            no        FORTRAN     280
    .
    .
    .
SHARE$DEBUG                     no        Image         0
SHARE$LIBRTL                    no        Image         0
SHARE$MTHRTL                    no        Image         0
SHARE$SHARE1                    no        Image         0
SHARE$SHARE2                    no        Image         0
```

```
total modules: 17.              bytes allocated: 162280.
DBG> SET MODULE SHARE$SHARE2
DBG> SHOW SYMBOL * IN SHARE$SHARE2
```

In this example, the command SHOW MODULE/SHARE identifies all of the modules in the current image and all of the shareable images (the names of the shareable images are prefixed with "SHARE$"). The command SET MODULE SHARE$SHARE2 sets the shareable image module SHARE$SHARE2. The SHOW SYMBOL command identifies any universal symbols defined in the shareable image SHARE2.

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 entered, 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 the following current output options:

- Debugger commands read from debugger command procedures are not echoed on the terminal.
- Debugger output is being displayed on the terminal.
- The debugging session is being logged to the log file USER$:[JONES.WORK]DEBUG.LOG;9.
- The screen contents are logged 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 debugger can interpret and display integer data in any one of four radixes: binary, decimal, hexadecimal, and octal. 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 entered, the radix for both entry and display (input radix and output radix, respectively) is decimal for all languages except BLISS and MACRO. It is hexadecimal for BLISS and MACRO.

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 entered, the override radix is "none".

Related commands: (SET, CANCEL) RADIX, EXAMINE, DEPOSIT, EVALUATE.

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
```

In this example, 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 is 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 entered, 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) — that is, all set modules and 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> SET SCOPE 0,STACKS\R2,SCREEN_IO,\
DBG> SHOW SCOPE
scope:
    0, [= TEST ],
    STACKS\R2,
    SCREEN_IO,
    \
```

In this example, the SET SCOPE command tells the debugger to look for symbols without path name prefixes according to the following scope search list. First the debugger looks in the PC scope (denoted by "0", which is in module TEST). If the debugger cannot find a specified symbol in the PC scope, it then looks in routine R2 of module STACKS; if necessary, it then looks in module SCREEN_IO, and then finally in the global symbol table

(denoted by the global scope, \). The SHOW SCOPE command identifies the current scope search list for symbol lookup.

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 entered, 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
```

In this example, 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 searches for the next occurrence of the string, but displays 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 third SHOW SEARCH command indicates that the debugger searches for all occurrences of the string, but displays the strings only 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.

SHOW SELECT

EXAMPLE

```
DBG> SHOW SELECT
display selections:
     scroll = SRC
     input  = none
     output = OUT
     error  = PROMPT
     source = SRC
     instruction = none
     program = PROMPT
     prompt = PROMPT
```

In this example, 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.

SHOW SOURCE

Displays the source directory search lists currently in effect.

FORMAT

SHOW SOURCE

PARAMETERS

None.

QUALIFIERS

/EDIT

Note: This qualifier 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 version number and 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.

SHOW SOURCE

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]
```

In this example, the SET SOURCE command directs the debugger to search the directories [PROJA],[PROJB], and DISK:[PETER.PROJC].

2

```
DBG> SET SOURCE/MODULE=COBOLTEST [], DISK$2:[PROJD]
DBG> SHOW SOURCE
source directory search list for COBOLTEST:
        []
        DISK$2:[PROJD]
source directory search list for all other modules:
        [PROJA]
        [PROJB]
        DISK:[PETER.PROJC]
```

In this example, the SET SOURCE command directs the debugger to search the current default directory ([]) and directory DISK$2:[PROJD] 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, use the command EXAMINE *address* to display the values of these arguments.

Related commands: SHOW CALLS.

EXAMPLE

```
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\%LINE 69
    saved R2:          0
    argument list:(1)  EIGHTQUEENS\%LINE 68+2

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
```

In this example, 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 entered, the default qualifiers are /LINE, /OVER, /NOSILENT, and /SOURCE.

If you invoke screen mode with the keypad-key sequence PF1-PF3, the command SET STEP NOSOURCE is issued in addition to the command SET MODE SCREEN (to eliminate redundant source display in output and DO displays). In that case, the default qualifiers for the STEP command are /LINE, /OVER, /NOSILENT, and /NOSOURCE.

Related commands: SET STEP, STEP.

EXAMPLE

```
DBG> SET STEP INTO,NOSYSTEM,NOSHARE,INSTRUCTION,NOSOURCE
DBG> SHOW STEP
step type: nosystem, noshare, nosource, nosilent, into routine calls, by instruction
```

In this example, this SHOW STEP command indicates that the debugger does the following:

- Steps into called routines, but not those in system space or in shareable images
- Steps by instruction
- Does not display lines of source code while stepping

SHOW SYMBOL

Displays information about the symbols in the debugger's run-time symbol table (RST) for the current image.

FORMAT

SHOW SYMBOL *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 asterisk wildcard character (*) by itself, all symbols are listed. You can use * within a symbol 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 in the current image.

The SHOW SYMBOL command displays only those symbols in the RST for the current image 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 global symbol table (GST) for the current image are searched for symbols that match the name specified by the symbol-name 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 (symbol definitions that are in the DEFINE symbol table).

/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 are not shown.

/LOCAL

Displays symbols that are defined with the DEFINE/LOCAL command (symbol definitions that are in the DEFINE symbol table).

SHOW SYMBOL

/TYPE
Displays data type information for each selected symbol.

/USE_CLAUSE
Note: This qualifier 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

Note: The current image is either the main image (by default) or the image established as the current image by a previous SET IMAGE command.

The SHOW SYMBOL command displays information that the debugger has about a given symbol in the current image. 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.

If you do not specify a qualifier, the SHOW SYMBOL command lists all of the possible declarations or definitions of a specified symbol that exist in the RST for the current image — that is, in all set modules and in the GST for that image. Symbols are displayed with their path names. A path name identifies the search scope (module, nested routines, blocks, and so on) that the debugger must follow to reach a particular declaration of a symbol. When specifying symbolic address expressions in debugger commands, you need to use path names only if a symbol is multiply defined and the debugger cannot resolve the ambiguity.

The /DEFINED and /LOCAL qualifiers display information about symbols defined with the DEFINE command (not the symbols that are derived from your program). The other qualifiers display information about symbols defined within your program.

Related commands: DEFINE, SHOW DEFINE, DELETE, SYMBOLIZE, SET MODE [NO]LINE, SET MODE [NO]SYMBOLIC.

EXAMPLES

1
```
DBG> SHOW SYMBOL I
data FORARRAY\I
```

This command shows that symbol I is defined in module FORARRAY and is a variable (data) rather than a routine.

2
```
DBG> SHOW SYMBOL/ADDRESS INTARRAY1
data FORARRAY\INTARRAY1
     descriptor address: 0009DE8B
```

This command shows that symbol INTARRAY1 is defined in module FORARRAY and has a virtual address of 0009DE8B.

3
```
DBG> SHOW SYMBOL *PL*
```

This command lists all the symbols whose names contain the string "PL".

4
```
DBG> SHOW SYMBOL/TYPE/ADDRESS *
```

This command displays all information about all symbols.

5
```
DBG> SHOW SYMBOL * IN MOD3\COUNTER
    routine MOD3\COUNTER
    data MOD3\COUNTER\X
    data MOD3\COUNTER\Y
```

This command lists all the symbols that are defined in the scope denoted by the path name MOD3\COUNTER.

6
```
DBG> DEFINE/COMMAND SB=SET BREAK
DBG> SHOW SYMBOL/DEFINED SB
defined SB
    bound to: SET BREAK
    was defined /command
```

In this example, 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: This 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 one of the following:

- 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 asterisk 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 value 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
```

In this example, 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 entered, the current height and width are, by default, the height and width known to the 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 routine CALC\MULT
tracepoint on calls:
        RET     RSB     BSBB    JSB     BSBW    CALLG   CALLS
```

The SHOW TRACE command identifies the tracepoints that are currently set. This example indicates that a tracepoint is triggered whenever execution reaches routine MULT in module CALC or one of the instructions RET, RSB, BSBB, JSB, BSBW, CALLG, or CALLS.

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 entered, 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 entered, the override type is "none".

Related commands: SET TYPE, CANCEL TYPE/OVERRIDE, (SET, SHOW, CANCEL) RADIX, (SET, SHOW, CANCEL) MODE, EXAMINE, DEPOSIT.

EXAMPLES

1

```
DBG> SET TYPE QUADWORD
DBG> SHOW TYPE
type: quadword integer
```

This 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 interprets and displays 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\X
watchpoint of SUB2\TABLE+20
```

This command displays two watchpoints, one at the variable X (defined in module MAIN), and the other at the location SUB2\TABLE+20 (20 bytes beyond the address denoted by the address expression TABLE).

SHOW WINDOW

Displays the name and screen position of predefined and user-defined screen-mode windows.

FORMAT

SHOW WINDOW *[wname[, . . .]]*

PARAMETERS

wname

Specifies the name of a screen window definition. If you do not specify a name, or if you specify the asterisk wildcard character (*) by itself, all window definitions are listed. You can use * within a window name. Do not specify a window definition name with /ALL.

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

Creates a subprocess, enabling you to execute DCL commands without terminating a debugging session or losing your 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 command) enables 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

/INPUT=file-spec

Specifies an input DCL command file containing one or more DCL commands to be executed by the spawned subprocess. The default file type is .COM. If you specify a DCL command string with the SPAWN command and an input file with the /INPUT qualifier, the command string is processed before the input file. Once processing of the input file is complete, the subprocess is terminated. Do not use the asterisk wildcard character (*) in the file specification.

/OUTPUT=file-spec

Writes the output from the SPAWN operation to the specified file. The default file type is .LOG. Do not use the asterisk wildcard character (*) in the file specification.

/[NO]WAIT

Controls whether the debugging session (the parent process) is suspended while the subprocess is running. /WAIT (default) suspends the debugging session until the subprocess is terminated. You cannot enter debugger commands until control returns to the parent process.

/NOWAIT executes the subprocess in parallel with the debugging session. You can enter 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. A message indicates when the spawned subprocess completes.

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 processes the second SPAWN command, including any qualifier specified with that command.

Related commands: ATTACH.

EXAMPLES

1
```
DBG> SPAWN
$
```

This command shows that the SPAWN command, with no parameter specified, creates a subprocess at DCL level. You can now enter DCL commands. Log out to return to the debugger prompt.

2
```
DBG> SPAWN/NOWAIT/INPUT=READ_NOTES/OUTPUT=0428NOTES
DBG>
```

This command creates a subprocess that is executed in parallel with the debugging session. This subprocess executes the DCL command procedure READ_NOTES.COM. The output from the spawned operation is written to the file 0428NOTES.LOG.

3
```
DBG> SPAWN/NOWAIT SPAWN/OUT=MYCOM.LOG @MYCOM
DBG>
```

This command creates a subprocess that is executed in parallel with the debugging session. This subprocess creates another subprocess to execute the DCL 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 (*n* is a decimal integer). 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.

/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 causes the debugger to 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 causes the debugger to 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.

/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* takes 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 causes the debugger to 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 causes the debugger to step into that routine. This is the default. /NOSYSTEM causes the debugger to step over that system routine (see description of /OVER qualifier).

STEP

DESCRIPTION

STEP command qualifiers determine the exact stepping behavior. In general, when you enter a STEP command, the debugger does the following:

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 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

If you plan to enter 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 BREAK/EXCEPTION.

EXAMPLES

1

```
DBG> STEP
stepped to FORSQUARE$MAIN\%LINE 4
     4:         OPEN(UNIT=8, FILE='DATAFILE.DAT', STATUS='OLD')
```

This command causes 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 causes 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 causes the debugger to step into the routine that is being called at the current PC value. The PC is then positioned at routine SUB1.

SYMBOLIZE

Converts a virtual address to a symbolic representation, if possible.

FORMAT

SYMBOLIZE *address-expression[, . . .]*

PARAMETERS

address-expression
Specifies an address expression to be symbolized. Do not use the asterisk 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 and a line number can be found that covers the address, the line number is included in the symbolization.

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, SHOW SYMBOL, (SET, SHOW, CANCEL) MODULE, EVALUATE/ADDRESS.

EXAMPLES

1
```
DBG> SYMBOLIZE %R5
address PROG\%R5:
    PROG\X
```

This example shows that the local variable X in routine PROG is located in register R5.

2

```
DBG> SYMBOLIZE %HEX 27C9E3
address 0027C9E3:
    MOD5\X
```

This command directs the debugger to treat the integer literal 27C9E3 as a hexadecimal value and convert that address to a symbolic representation, if possible. The address converts to the symbol X in module MOD5.

TYPE

Displays lines of source code.

FORMAT

TYPE *[[mod-name\]line-num[:line-num]*
[,[mod-name\]line-num[:line-num][, . . .]]

PARAMETERS

mod-name

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 (as established by a previous SET SCOPE command, or the PC scope if no SET SCOPE command was entered) to find source lines for display. If you specify a scope search list with the SET SCOPE command, the debugger searches for source lines only in the module associated with the first named scope.

line-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 display all the source lines of a module 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 module.

After displaying a single line of source code, you can display the next line of that module by entering a TYPE command without a line number—that is, by entering 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.

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. They appear in a compiler-generated listing and in a screen-mode source display.

When specifying a module name with the TYPE command, note that the module must be set. Use the SHOW MODULE command to determine whether a particular module is set. Then use the SET MODULE command, if necessary.

In screen mode, the output of a TYPE command is directed at the current source display, not at an output or DO display. The source display shows the lines specified and any surrounding lines that fit in the display window.

Related commands: SET MODE [NO]SCREEN, EXAMINE/SOURCE, SET STEP [NO]SOURCE, STEP/[NO]SOURCE, SET (BREAK, TRACE, WATCH) /[NO]SOURCE, (SET, SHOW, CANCEL) SCOPE.

EXAMPLES

1

```
DBG> TYPE 160
module COBOLTEST
   160: START-IT-PARA.
DBG> TYPE
module COBOLTEST
   161:          MOVE SC1 TO ES0.
```

In this example, the first TYPE command displays line 160, using the current scope to locate the module containing that line number. The second TYPE command, entered without specifying a line number, displays the next line in that module.

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, using the current scope to locate the module.

3

```
DBG> TYPE SCREEN_IO\7,22:24
```

This command displays line 7 and lines 22 through 24 in module SCREEN_IO.

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

Part III contains debugger reference information.

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. To change the default, use the 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 are interpreted according to the currently set language. Address expressions that are associated with compiler generated types are treated according to that type. Other address expressions are treated as having the type longword integer.
DISPLAY	DISPLAY/DYNAMIC/NOMARK_CHANGE/POP. The current display kind, window, and size remain unchanged.
EDIT	EDIT/NOEXIT. The default is to invoke the VAX Language Sensitive Editor in a spawned subprocess. 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	The contents of program locations that are associated with a compiler generated type are interpreted and displayed according to that type. The contents of other locations are interpreted and displayed as longword integers.
EXPAND	EXPAND/DOWN, /UP: 1 line. EXPAND/LEFT, RIGHT: 1 column.

Command Defaults

Command	Default
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. If no module name is specified, the debugger uses the current scope to find a module and searches that module for an occurrence of the string. The current scope is that established by a previous SET SCOPE command, or the PC scope if no SET SCOPE command was entered. Also, if no string is specified, the string specified in the last SEARCH command, if any, is used.
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, NOOPERANDS, NOSCREEN, NOSEPARATE, SCROLL, SYMBOLIC
SET OUTPUT	SET OUTPUT NOLOG, NOSCREEN_LOG, TERMINAL, NOVERIFY
SET PROMPT	SET PROMPT/NOPOP 'DBG> '
SET RADIX	For all languages except BLISS and MACRO: SET RADIX DECIMAL. For BLISS and MACRO: SET RADIX HEXADECIMAL.

Command Defaults

Command	Default
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	When searching for a source file, the debugger uses the full file specification that is stored in the run-time symbol table (RST).
SET STEP	SET STEP SOURCE, NOSILENT, OVER, LINE
SET TERMINAL	The values of /PAGE and /WIDTH default to those set at DCL level (see the *VMS DCL Dictionary* or enter the DCL command HELP SET TERMINAL).
SET TRACE	SET TRACE/INTO/JSB/SHARE/SYSTEM /NOSILENT/SOURCE
SET TYPE	The default type for program locations that are associated with a compiler generated type is that type. The default type for other locations is longword integer.
SET WATCH	For static variables: SET WATCH/NOSILENT /SOURCE. For nonstatic variables: SET WATCH /NOSILENT/OVER/SOURCE.
SPAWN	SPAWN/WAIT
STEP	STEP/OVER/LINE
TYPE	If no module name is specified, the debugger uses the current scope to find a module and searches that module for source lines for display. The current scope is that established by a previous SET SCOPE command, or the PC scope if no SET SCOPE command was entered. Also, if no line is specified after a single source line has been displayed with the TYPE command, the next line in that module 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 enter certain commands with fewer keystrokes than if you were to type them at the keyboard. For example, pressing the COMMA (,) keypad key is equivalent to typing 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.

To use function keys, keypad mode must 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 by pressing keypad key PF3 (SET MODE SCREEN).

If you want to use the keypad keys to enter numbers rather than debugger commands, enter the command SET MODE NOKEYPAD.

B.1 DEFAULT, GOLD, and BLUE Functions

A given key typically has three predefined functions:

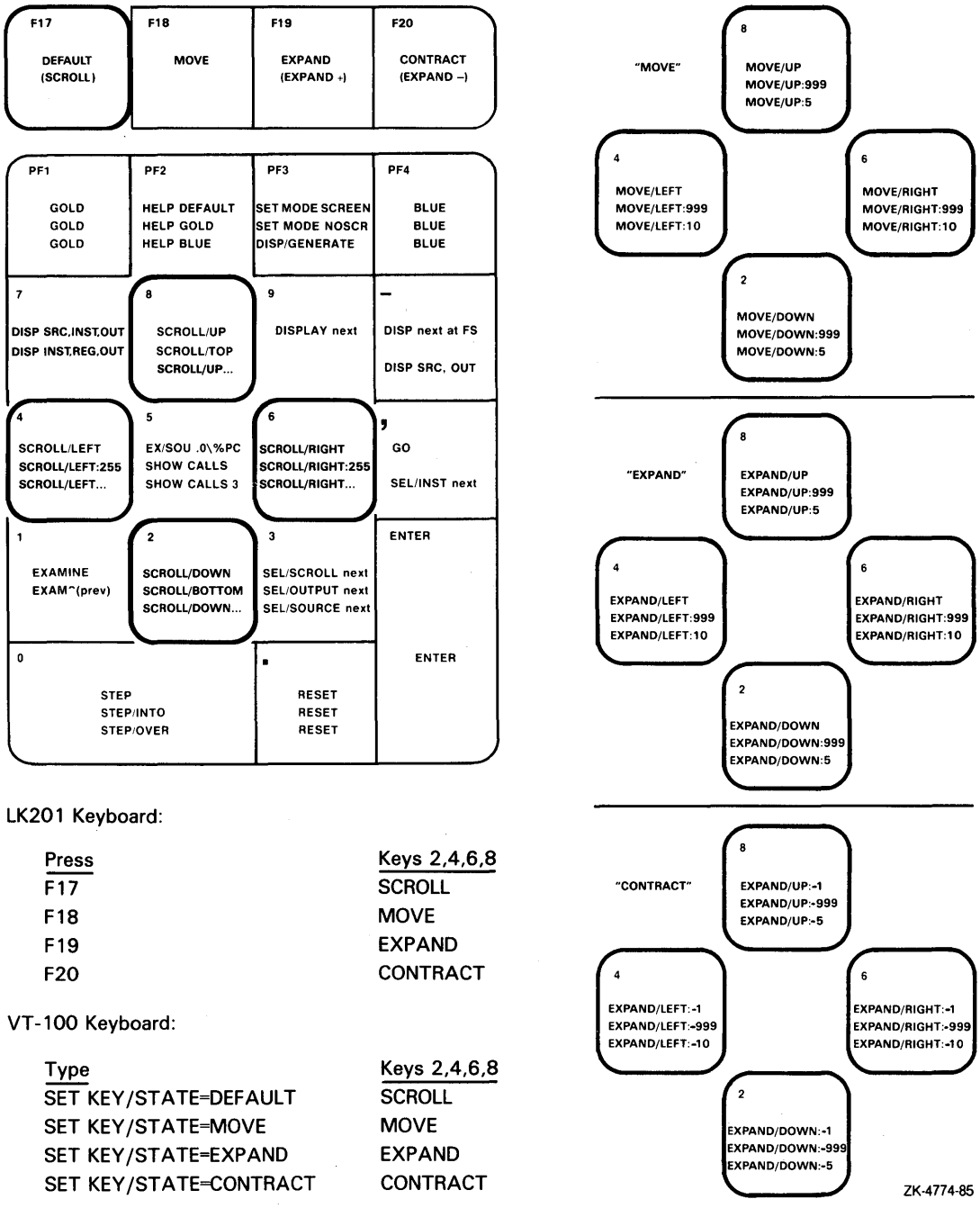
- One function is entered by pressing the given key by itself. This is the *DEFAULT* function.
- A second function is entered by pressing the PF1 key and then the given key. This is the *GOLD* function, because PF1 is also called the GOLD key.
- A third function is entered by pressing the PF4 key and 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).

Predefined Key Functions

B.1 DEFAULT, GOLD, and BLUE Functions

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

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, do the following:

1 Press key PF4
2 Press keypad key 2
3 Type :12 at the keyboard
4 Press the RETURN key

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.
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 puts the keypad in the DEFAULT (scroll) state; pressing BLUE-KP9 cycles 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
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	Command 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).

Table B–4 (Cont.) Debugger Key Definitions

Key	State	Command Invoked or Function
	GOLD	EXAMINE ^. Enables you to 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
	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

Table B–4 (Cont.) Debugger Key Definitions

Key	State	Command Invoked or Function
5	DEFAULT	EXAM/SOURCE .%SOURCE_SCOPE\%PC; EXAM/INST .0\%PC. In line (noscreen) mode, enables you to 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
	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

Table B–4 (Cont.) Debugger Key Definitions

Key	State	Command Invoked or Function
	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).
	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		Enables you to enter (terminate) a command. Same effect as RETURN.

Table B–4 (Cont.) Debugger Key Definitions

Key	State	Command Invoked or Function
PERIOD	Default	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	Default	SCROLL/DOWN
Prev Screen	Default	SCROLL/UP
Remove	Default	DISPLAY/REMOVE %CURSCROLL. Removes the current scrolling display from the display list.
Select	Default	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.
F20		See Table B–2.
CTRL/W		DISPLAY/REFRESH
CTRL/Z		EXIT

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 (command[; . . .])	Specifies an automatically updated output display. The commands are executed in the order listed each time the debugger gains control. Their output forms the contents of the display. If you specify more than one command, they must be separated by semicolons.
INSTRUCTION	Specifies an instruction display. If selected as the current instruction display with the SELECT/INSTRUCTION command, it displays 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 displays any debugger output that is not directed to another display. If selected as the current input display with the SELECT/INPUT command, it echoes debugger input. If selected as the current error display with the SELECT/ERROR command, it displays debugger diagnostic messages.
REGISTER	Specifies an automatically updated register display. The display is updated each time the debugger gains control.

Screen Mode Reference Information

C.1 Display Kinds

SOURCE	Specifies a source display. If selected as the current source display with the SELECT/SOURCE command, it displays 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	EFFECT
/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.

SELECT	EFFECT
/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.

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	no display selected
Output	OUT
Program	PROMPT
Prompt	PROMPT
Scroll	SRC
Source	SRC

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)

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 value (where execution is suspended).

The default characteristics of the SRC display are the following:

Display kind	source (examine/source .%source_scope\%pc)
Attributes	scroll, source
Position	H1
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 module where execution is suspended, the SRC display is centered on the line corresponding to the current PC value. 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. When displaying source lines that are not associated with the module where execution is suspended, the debugger issues the following message:

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

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 the display in which 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 the current PC value.

The default characteristics of the INST display are the following:

Display kind	instruction (examine/instruction .0\%pc)
Attributes	none
Position	H1, removed
Size	64 lines
Dynamic	yes

C.3.5 REG (Register Display)

REG automatically shows the current values (in hexadecimal format) of all VAX machine registers (R0 through R11), 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	yes

If the register window is resized, the debugger automatically reformats the displayed information to adapt to the new window size. The debugger always displays the contents of registers R0 through R11, AP, FP, SP, PC, and PSL. If the resized window is too small to display all the register information, you can scroll vertically or horizontally to view any information that may be hidden. If the resized window is larger than necessary to display register information, the debugger fills the remaining space with information (in hexadecimal format) contained in the user stack.

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 C.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 Screen 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 entering 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, enable you to 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 enter 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 that 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 workstation, the screen is larger in both height and width. The debugger can accommodate screen sizes up to 100 lines by 255 columns.

The debugger has many predefined windows that you can use to position displays on the screen. The SHOW WINDOW command identifies all predefined and user defined windows. The predefined windows are expressed in terms of fractions of the screen dimensions (for example, quarters, halves, and so on). Therefore, the positions and dimensions of the predefined windows that are indicated by the SHOW WINDOW command are adjusted for the screen dimensions.

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, sixths, and eighths, 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, E: eighth). The trailing numbers denote specific fractions of the screen height, starting from the top. For example:

- Windows T1, T2, and T3 occupy the top, middle and bottom thirds of the screen, respectively.
- Window RH2 occupies the right bottom half of the screen.
- Window LQ23 occupies the left middle two quarters of the screen.
- Window S45 occupies the fourth and fifth sixths of the screen.

The horizontal boundaries (start-column, column-count) of the predefined windows for the default terminal screen width of 80 columns are as follows:

- Left hand windows: (1,40)
- Right hand windows: (42,39)

Screen Mode Reference Information

C.5 Screen Dimensions and Predefined Windows

The vertical boundaries (start-line, line-count) of the predefined windows for the default terminal screen height of 24 lines are as follows:

Window Name	Start-line,Line-count	Window Location
FS	(1,23)	Full screen
H1	(1,11)	Top half
H2	(13,11)	Bottom half
T1	(1,7)	Top third
T2	(9,7)	Middle third
T3	(17,7)	Bottom third
Q1	(1,5)	Top quarter
Q2	(7,5)	Second quarter
Q3	(13,5)	Third quarter
Q4	(19,5)	Bottom quarter
S1	(1,3)	Top sixth
S2	(5,3)	Second sixth
S3	(9,3)	Third sixth
S4	(13,3)	Fourth sixth
S5	(17,3)	Fifth sixth
S6	(21,3)	Bottom sixth
E1	(1,2)	Top eighth
E2	(4,2)	Second eighth
E3	(7,2)	Third eighth
E4	(10,2)	Fourth eighth
E5	(13,2)	Fifth eighth
E6	(16,2)	Sixth eighth
E7	(19,2)	Seventh eighth
E8	(22,2)	Bottom eighth

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. Signaling 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 want 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 enter 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
DBG$INIT	Points to your debugger initialization file. Default: no debugger initialization file. DBG$INIT accepts a full or partial VMS file specification as well as a search list.
DBG$INPUT	Points to the debugger input device. Default: SYS$INPUT.
DBG$OUTPUT	Points to the debugger output device. Default: SYS$OUTPUT.

You can use the DCL commands 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
```

See Section 7.2 for information on debugger initialization files. See Section 8.2 for information on using DBG$INPUT and DBG$OUTPUT to debug screen-oriented programs at two terminals.

D.3 Built-in Symbols

The debugger's built-in symbols provide options for specifying entities in your program and enable you to 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—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>), %LABEL, %LINE, backslash (\) —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
%AP (R12)	Argument pointer
%FP (R13)	Frame pointer
%SP (R14)	Stack pointer
%PC (R15)	Program counter
%PSL	Processor status longword

For example, the following EXAMINE command obtains the contents of the PC (the address contained in the PC):

```
DBG> EXAMINE %PC
MOD\%PC: 1553
```

You can abbreviate registers by leaving out the percent character (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 built-in symbols. The debugger interprets these symbols as debugger built-in symbols only if your program does not contain variables of the same names.

D.3.2 Constructing Identifiers

The %NAME built-in symbol enables you to 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 may be used in address expressions and language expressions to specify that an integer literal that follows (or all integer literals in a parenthesized expression that follows) should be interpreted in binary, decimal, hexadecimal, or octal radix, respectively. Use these radix built-in symbols only with integer literals.

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

DBG> SET RADIX DECIMAL
DBG> EVALUATE %HEX 20 + 33 ! Treat 20 as hexadecimal, 33 as decimal)
65                         ! Resulting value is decimal
DBG> EVALUATE %HEX (20+33) ! Treat both 20 and 33 as hexadecimal
83
DBG> EVALUATE %HEX (20+ %OCT 10 +33) ! Treat 20 and 33 as
91                                   ! hexadecimal and 10 as octal
DBG> SYMBOLIZE %HEX 27C9E3  ! Symbolize a hexadecimal address
DBG>DEP/INST %HEX 5432 = 'MOVL ^O%DEC 222, R1'
   ! Treat address 5432 as hexadecimal, and operand 222 as decimal
```

D.3.5 Specifying Program Locations and the Current Value of an Entity

The following built-in symbols enable you to 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.

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.

D.3.6 Using Symbols and Operators in Address Expressions

The symbols and 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
%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.
\ (backslash)	When used within a path name, delimits each element of the path name. In this context, the backslash cannot be the leftmost element of the complete path name. When used as the *prefix* to a symbol, specifies that the symbol is to be interpreted as a global symbol. In this context, the backslash must be the leftmost element of the symbol's complete path name.

Symbol	Description
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.
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.

The following examples illustrate the use of built-in symbols and operators in address expressions.

%LINE and %LABEL Operators

The following command sets a tracepoint at line 26 of the module where execution is currently suspended:

```
DBG> SET TRACE %LINE 26
DBG>
```

The next command displays the source line associated with line 47:

```
DBG> EXAMINE/SOURCE %LINE 47
module MAIN
    47:  procedure SWAP(X,Y: in out INTEGER) is
DBG>
```

The next command sets a breakpoint at label 10 of module MOD4:

```
DBG> SET BREAK MOD4\%LABEL 10
DBG>
```

Path Name Operators

The following command displays the value of the variable COUNT that is declared in routine ROUT2 of module MOD4. The backslash (\) path name delimiter separates the path name elements:

```
DBG> EXAMINE MOD4\ROUT2\COUNT
MOD4\ROUT2\COUNT: 12
DBG>
```

The following command sets a breakpoint on line 26 of the module QUEUE_MANAGER:

```
DBG> SET BREAK QUEUE_MANAGER\%LINE 26
DBG>
```

The following command displays the value of the global symbol X:

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

Arithmetic Operators

The order in which the debugger evaluates the elements of an address expression is similar to that used by most programming languages. The order 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 debugger operators are listed in decreasing order of precedence as follows:

1. Unary operators ((.), (@), (+), (-))
2. Multiplication and division operators ((*), (/))
3. Addition and subtraction operators ((+), (-))

For example, when evaluating the following 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
```

The following command displays the value contained in the virtual memory location X + 4 bytes:

```
DBG> EXAMINE X + 4
```

Contents-of Operator

The following examples illustrate use of the contents-of operator. In the next example, the instruction at the current PC value 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
```

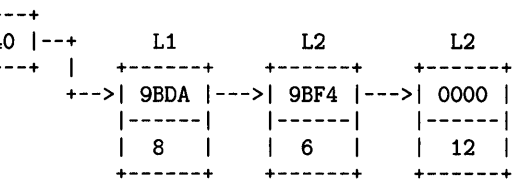
In the next example, the source line at the PC value one level down the call stack is obtained (at the call to routine SWAP):

```
DBG> EXAMINE/SOURCE .1\%PC
module MAIN
MAIN\%LINE 134:      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 the 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).
```

Bit-Field Operator

The following example shows how to use the bit-field operator. 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 enter the following command:

```
DBG> EXAMINE X_NAME <3,4,0>
```

D.3.7 Obtaining Information About Exceptions

The following built-in symbols enable you to obtain information about the current exception and use that information to qualify breakpoints or tracepoints.

Symbol	Description
%EXC_FACILITY	Name of facility that issued the current exception
%EXC_NAME	Name of current exception
%ADAEXC_ NAME	Ada exception name of current exception (for Ada programs only)
%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 executes when a GO or STEP command is entered.
%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_TASK NEQ %ACTIVE DO (SET TASK %NEXT_TASK; SHOW CALLS)
```

E Summary of Debugger Support for Languages

The debugger supports most of the 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 and address 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 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)

Summary of Debugger Support for Languages

E.1 Debugger Support for Language ADA

Kind	Symbol	Function
Infix	> =	Greater than or equal (only scalar and string types)
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)

Note: 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**
- User-defined operators

E.1.2 Constructs in Language and Address Expressions

Supported constructs in language and address expressions for ADA follow:

Symbol	Construct
( )	Subscripting
.	Record component selection
.ALL	Pointer dereferencing

E.1.3 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 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.

Summary of Debugger Support for Languages

E.1 Debugger Support for Language ADA

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 Tasking States

Support for ADA tasking states is as follows:

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.1.6 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.

Summary of Debugger Support for Languages

E.1 Debugger Support for Language ADA

Exception-Related Events

Event Keyword	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 Keyword	Description
RENDEZVOUS_EXCEPTION	Triggers when an exception begins to propagate out of a rendezvous.
DEPENDENTS_EXCEPTION	Triggers when an unhandled exception causes a task to wait for dependent tasks in some scope (includes unhandled exceptions, such as task rundown signals, that are internal to the VAX Ada run-time library). Often immediately precedes a deadlock.

Task Termination Events

Event Keyword	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 unhandled exception.
ABORT_TERMINATED	Triggers when a task is terminating due to an abort.

Low-Level Task Scheduling Events

Event Keyword	Description
RUN	Triggers when a task is about to run.
PREEMPTED	Triggers when a task is being preempted from the RUN state, and its state changes to READY.
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 Language BASIC

This section includes information about debugger support for BASIC.

E.2.1 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 Constructs in Language and Address Expressions

Supported constructs in language and address expressions for BASIC follow:

Symbol	Construct
( )	Subscripting
::	Record component selection

E.2.3 Data Types

Supported BASIC data types follow:

E.2 Debugger Support for Language BASIC

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)

Note:

1 **Expressions that overflow in the BASIC language do not necessarily overflow when evaluated by the debugger. The debugger tries 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 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

Kind	Symbol	Function
Infix		Left shift
Infix	EQL	Equal to
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 Constructs in Language and Address Expressions

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 Data Types

Supported BLISS data types follow:

E.3 Debugger Support for BLISS

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 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

Kind	Symbol	Function
Infix	<	Less than
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 Constructs in Language and Address Expressions

Supported constructs in language and address expressions for C follow:

Symbol	Construct
[ ]	Subscripting
.	Structure component selection
->	Pointer dereferencing

E.4.3 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)

Note

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 nonblank 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. 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 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 Constructs in Language and Address Expressions

Supported constructs in language and address expressions for COBOL follow:

Symbol	Construct
( )	Subscripting
OF	Record component selection
IN	Record component selection

E.5.3 COBOL Data Types

Supported COBOL data types follow:

COBOL Type	VAX Type Name
COMP	Longword Integer (L,LU)
COMP	Word Integer (W,WU)
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	Numeric string, unsigned (NU)
Leading Separate Sign	Numeric string, left separate sign (NL)
Leading Overpunched Sign	Numeric string, left overpunched sign (NLO)
Trailing Separate Sign	Numeric string, right separate sign (NR)
Trailing Overpunched Sign	Numeric string, right overpunched sign (NRO)

Note

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 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 Constructs in Language and Address Expressions

Supported constructs in language and address expressions for DIBOL follow:

Symbol	Construct
( )	Substring
[ ]	Subscripting
.	Record component selection

E.6.3 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 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

Kind	Symbol	Function
Infix	.XOR.	Exclusive OR
Infix	.EQV.	Equivalence
Infix	.NEQV.	Exclusive OR

E.7.2 Constructs in Language and Address Expressions

Supported constructs in language and address expressions for FORTRAN follow:

Symbol	Construct
( )	Subscripting
.	Record component selection

E.7.3 Predefined Symbols

Supported FORTRAN predefined symbols follow:

Symbol	Description
.TRUE.	Logical True
.FALSE.	Logical False

E.7.4 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)

FORTRAN Type	VAX Type Name
CHARACTER	ASCII Text (T)
Arrays	(None)
Records	(None)

Note

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 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

Kind	Symbol	Function
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
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 Constructs in Language and Address Expressions

Supported constructs in language and address expressions for MACRO follow:

Symbol	Construct
[ ]	Subscripting
<p,s,e>	Bitfield selection as in BLISS

Note

The DST information generated by the MACRO assembler treats a label that is followed by an assembler directive for storage allocation as an array variable whose name is the label. This enables you to use the array syntax of a high-level language when examining or manipulating such data.

In the following example of MACRO source code, the label LAB4 designates hexadecimal data stored in four words:

```
LAB4:    .WORD    ^X3F,5[2],^X3C
```

The debugger treats LAB4 as an array variable. For example, the next command displays the value stored in each element (word):

```
DBG> EXAMINE LAB4
.MAIN.\MAIN\LAB4
    [0]:        003F
    [1]:        0005
    [2]:        0005
    [3]:        003C
DBG>
```

The next command displays the value stored in the fourth word (the first word is indexed as element "0"):

```
DBG> EXAMINE LAB4[3]
.MAIN.\MAIN\LAB4[3]:    03C
DBG>
```

E.8.3 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)
(Not applicable)	F_Floating (F)
(Not applicable)	D_Floating (D)
(Not applicable)	G_Floating (G)
(Not applicable)	H_Floating (H)
(Not applicable)	Packed decimal (P)

E.9 Debugger Support for Language PASCAL

This section includes information about debugger support for PASCAL.

E.9.1 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

Kind	Symbol	Function
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 Constructs in Language and Address Expressions

Supported constructs in language and address expressions for PASCAL follow:

Symbol	Construct
[ ]	Subscripting
.	Record component selection
^	Pointer dereferencing

E.9.3 Predefined Symbols

Supported PASCAL predefined symbols follow:

Symbol	Meaning
TRUE	Boolean True
FALSE	Boolean False
NIL	Nil pointer

E.9.4 Built-In Functions

Supported PASCAL built-in functions follow:

Symbol	Meaning
SUCC	Logical successor
PRED	Logical predecessor

E.9.5 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 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 Constructs in Language and Address Expressions

Supported constructs in language and address expressions for PL/I follow:

Symbol	Construct
( )	Subscripting
.	Structure component selection
->	Pointer dereferencing

E.10.3 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.

Summary of Debugger Support for Languages

E.11 Debugger Support for Language RPG

E.11.1 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
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 Constructs in Language and Address Expressions

Supported constructs in language and address expressions for RPG follow:

Symbol	Construct
( )	Subscripting

E.11.3 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 Language SCAN

This section includes information about debugger support for SCAN.

E.12.1 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 Constructs in Language and Address Expressions

Supported constructs in language and address expressions for SCAN follow:

Symbol	Construct
( )	Subscripting
.	Record component selection
->	Pointer dereferencing

E.12.3 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
VARYING STRING (n)	TEXT with CLASS=VS
DYNAMIC STRING	TEXT with CLASS=D
TREE	(None)
TREEPTR	(None)
RECORD	(None)
OVERLAY	(None)

Note

1 There is no specific support for the following datatypes: FILE, TOKEN, GROUP, SET. Examining a FILL variable displays 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 produces 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 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	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 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

Kind	Symbol	Function
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 Constructs in Language and Address Expressions

Supported constructs in language and address expressions for UNKNOWN follow:

Symbol	Construct
[ ]	Subscripting
( )	Subscripting
.	Record component selection
^	Pointer dereferencing

E.13.3 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.

Note

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

/ABORT qualifier • CD–142
/AC
  See /ASCIC qualifier
/ACTIVE qualifier • CD–142
%ACTIVE_TASK • D–9
/AD
  See /ASCID qualifier
%ADAEXC_NAME • 8–15, D–8
Address expression
  compared to language expression • 3–7
  current entity • 3–8, D–4
  DEPOSIT command • 3–3, CD–44
  EVALUATE/ADDRESS command • 2–13, 3–12, CD–60
  EXAMINE command • 3–2, CD–62
  EXAMINE/SOURCE command • 5–4
  logical predecessor • 3–8, D–4
  logical successor • 3–8, D–4
  SET BREAK command • 2–10, CD–96
  SET TRACE command • 2–10, CD–147
  SET WATCH command • 2–17, CD–156
  symbolic • 3–4
  SYMBOLIZE command • 3–13, CD–212
  type of • 3–4
/ADDRESS qualifier • 7–6, CD–35, CD–60, CD–195
/AFTER qualifier • CD–97, CD–147, CD–156
Aggregate
  DEPOSIT command • 3–16, 3–18, CD–44
  EXAMINE command • 3–16, 3–18, CD–62
  SET WATCH command • 2–18
/ALL qualifier
  CANCEL BREAK command • CD–14
  CANCEL DISPLAY command • CD–16
  CANCEL IMAGE command • CD–17
  CANCEL MODULE command • CD–19
  CANCEL TRACE command • CD–25
  CANCEL WATCH command • CD–28
  CANCEL WINDOW command • CD–29
  DELETE command • CD–40
  DELETE/KEY command • CD–42
  EXTRACT command • CD–73
  SEARCH command • CD–90
  SET IMAGE command • CD–110
/ALL qualifier (cont'd.)
  SET MODULE command • CD–123
  SET TASK command • CD–142
  SHOW DISPLAY command • CD–169
  SHOW KEY command • CD–174
  SHOW TASK command • CD–198
  SHOW WINDOW command • CD–205
%AP • 3–22, D–2
Apostrophe (')
  ASCII string delimiter • 3–16
  instruction delimiter • 3–21
/APPEND qualifier • CD–73
Array type • 3–16
/ASCIC qualifier • CD–44, CD–62
/ASCID qualifier • CD–44, CD–62
/ASCII qualifier • CD–45, CD–62
ASCII string type • 3–16, 3–26, CD–44, CD–62, CD–153
/ASCIW qualifier • CD–45, CD–63
/ASCIZ qualifier • CD–45, CD–63
AST (asynchronous system trap) • 8–16
  CALL command • 8–17, CD–10
  disabling • CD–50
  displaying AST handling conditions • CD–163
  enabling • CD–57
  SHOW CALLS command • 8–17
AST-driven program
  debugging • 8–16
Asterisk (*)
  HELP command • CD–79
  multiplication operator • D–6
/AST qualifier • 8–17, CD–11
At sign (@)
  contents-of operator • D–6
  execute-procedure command • 7–1, CD–7
  SET ATSIGN command • CD–95
  SHOW ATSIGN command • CD–164
ATTACH command • 2–6, CD–9
Attribute
  display • 6–3, 6–16, CD–92, CD–189
/AW
  See /ASCIW qualifier
/AZ
  See /ASCIZ qualifier

B

Backslash (\)
  current value • 3–5
  global-symbol specifier • 4–9, CD–131, D–5
  path name delimiter • 4–8, 5–4, D–5
%BIN • 3–12, D–4
/BINARY qualifier • 3–11, CD–58, CD–60, CD–63
Bit field operator (<p,s,e>) • D–6
/BOTTOM qualifier • CD–87
/BRANCH qualifier • CD–14, CD–25, CD–97, CD–148, CD–208
Breakpoint
  canceling • 2–17, CD–14
  defined • 2–10
  delayed triggering of • 2–15, CD–97
  displaying • CD–165
  DO clause • 2–15
  exception • 8–10, CD–96
  predefined • 8–10
  setting • 2–10, CD–96
  source display at • 5–7
  WHEN clause • 2–15
/BRIEF qualifier • CD–174
Built-in symbol • C–6, D–2
/BYTE qualifier • CD–45, CD–63

C

/CALLABLE_EDT qualifier • CD–107
/CALLABLE_LSEDIT qualifier • CD–107
/CALLABLE_TPU qualifier • CD–107
CALL command • 7–11, CD–10
  and ASTs • 8–17, CD–10
%CALLER_TASK • D–9
/CALL qualifier • CD–14, CD–25, CD–97, CD–148, CD–208
/CALLS qualifier • CD–123, CD–198
Call stack
  displaying • 1–13, 8–12, CD–166, CD–193
CANCEL ALL command • CD–13
CANCEL BREAK command • 2–17, CD–14
CANCEL DISPLAY command • 6–9, CD–16
CANCEL IMAGE command • 4–13, CD–17
CANCEL MODE command • CD–18
CANCEL MODULE command • 4–6, CD–19
CANCEL RADIX command • 3–11, CD–21
CANCEL SCOPE command • 4–10, CD–22
CANCEL SOURCE command • 5–3, CD–23
CANCEL TRACE command • 2–17, CD–25
CANCEL TYPE/OVERRIDE command • 3–25, CD–27
CANCEL WATCH command • 2–17, CD–28
CANCEL WINDOW command • 6–12, CD–29
Case sensitivity • 8–9
Catchall handler • 8–13
Circumflex (^) • 3–8, D–4
/CLEAR qualifier • CD–52
Colon (:)
  range delimiter • 3–17, CD–62
Command format
  debugger • CD–3
Command procedure
  debugger • 7–1
  default directory • CD–95, CD–164
  displaying commands in • CD–126
  exiting • CD–7, CD–69, CD–84
  invoking • CD–7
  log file as • 7–5
  passing parameters to • 7–2, CD–32
  recreating displays • 6–19, CD–73
/COMMAND qualifier • 7–6, CD–35
Comment
  format • CD–4
Compiler
  compiler generated type • 3–4
  /DEBUG qualifier • 4–2, 5–1
  /LIST qualifier • 5–1
  /NOOPTIMIZE qualifier • 4–2, 8–1
Condition handler
  debugging • 8–10
/CONDITION_VALUE qualifier • CD–58, CD–63
Contents-of operator • 3–6, 3–19, D–6
CONTINUE command • 2–5
CTRL/C • 2–5, CD–30
CTRL/W • CD–30, CD–53
CTRL/Y • 2–3, 2–4, 2–5, CD–30
CTRL/Z • 2–4, CD–30
%CURDISP • C–6
%CURLOC • 3–8, D–4
Current
  display • 6–3, 6–16, CD–92, CD–189
  entity • 3–8, 3–19, D–4
  image • 4–13, CD–110, CD–173
  language • 3–10, CD–113, CD–176
  radix • 3–10, CD–129, CD–185
  scope • 4–10, CD–131, CD–186
  type • 3–24, CD–153, CD–203
  value • 3–5, D–4

%CURSCROLL • C–6
%CURVAL • 3–5, D–4

D

/D_FLOAT qualifier • CD–45, CD–63
Data type
see Type
/DATE_TIME qualifier • CD–45, CD–63
DBG$INIT • 7–4, D–1
DBG$INPUT • 8–5, D–1
DBG$OUTPUT • 8–5, D–1
DEBUG command • 2–3, 2–5
Debugger command
dictionary • CD–3
format • CD–3
repeating • CD–75, CD–85, CD–216
summary • 1–25
/DEBUG qualifier • 2–1, 4–2, 4–4, 5–1
shareable image • 4–11
Debug symbol table
see DST
%DEC • 3–12, D–4
/DECIMAL qualifier • 3–11, CD–58, CD–60, CD–63
DECLARE command • 7–2, CD–32
/DEFAULT qualifier • CD–63
DEFINE command • 7–6, CD–35
displaying default qualifiers for • CD–168
setting default qualifiers for • CD–102
/DEFINED qualifier • CD–195
DEFINE/KEY command • 7–8, CD–37
DELETE command • 7–6, CD–40
DELETE/KEY command • 7–8, CD–42
DEPOSIT command • 3–3, CD–44
/DIRECTORY qualifier • CD–175
/DIRECT qualifier • CD–195
DISABLE AST command • 8–17, CD–50
Display
See also Source display
attribute • 6–3, 6–16, CD–92, CD–189
canceling • 6–9, CD–16
contracting • 6–10, CD–71
creating • 6–10, CD–103
current • 6–3, 6–16, CD–92
default configuration • 6–2
defined • 6–2
expanding • 6–10, CD–71
extracting • 6–19, CD–73
Display (cont'd.)
hiding • 6–9, CD–53, CD–104
identifying • 6–9, CD–169
kind • 6–3, 6–12, C–1
list • 6–3, CD–169, C–6
moving • 6–9, CD–82
pasteboard • 6–3, CD–54, CD–105
predefined • 6–4, C–3
removing • 6–9, CD–53, CD–105
saving • 6–19, CD–86
scrolling • 6–8, CD–87
selecting • 6–16, CD–92
showing • 6–9, CD–51
window • 6–2, 6–11, C–7
DISPLAY command • 6–9, CD–51
DO clause
example • 2–15
exiting • CD–69, CD–84
format • CD–4
DO display • 6–13, C–1
/DOWN qualifier • CD–71, CD–82, CD–87
DST (debug symbol table)
creating • 4–4
shareable image • 4–13
source line correlation • 5–1
Dynamic mode
image setting • 4–13
module setting • 4–6
Dynamic module setting • CD–120
/DYNAMIC qualifier • CD–52, CD–104

E

/ECHO qualifier • CD–37
EDIT command • CD–55
/EDIT qualifier • CD–23, CD–136, CD–191
ENABLE AST command • 8–17, CD–57
/ERROR qualifier • 6–17, CD–92
EVALUATE/ADDRESS command • 2–13, 2–19, 3–12, CD–60
EVALUATE command • 3–5, CD–58
Event facility, setting • CD–109
Eventpoint
See Breakpoint
See Tracepoint
See Watchpoint
/EVENT qualifier • 2–16, CD–14, CD–25, CD–97, CD–148
EXAMINE command • 3–2, CD–62

EXAMINE/INSTRUCTION command • 3–19, 6–6, C–5
EXAMINE/SOURCE command • 5–4, 6–4, C–4
Exception breakpoint or tracepoint
  canceling • 8–11, CD–14, CD–25
  qualifying • 8–15, D–8
  resuming execution at • 8–11
  setting • 8–11, CD–97, CD–148
Exception condition • 8–10
Exception handler
  debugger as • 2–22
  debugging • 8–10
/EXCEPTION qualifier • 8–10, CD–14, CD–25, CD–97, CD–148, CD–208
Exclamation point (!)
  comment delimiter • CD–4
  log file • 7–5
%EXC_FACILITY • 8–15, D–8
%EXC_NAME • 8–15, D–8
%EXC_NUMBER • 8–15, D–8
%EXC_SEVERITY • 8–15, D–8
Execution
  as controlled by debugger • 2–22
  discrepancies caused by debugger • 2–23
  interrupting with CTRL/Y • 2–3, 2–5, CD–30
  monitoring with SHOW CALLS command • 1–13, CD–166
  monitoring with tracepoint • 2–10, CD–147
  resuming after exception break • 8–11
  starting or resuming with CALL command • 7–11, CD–10
  starting or resuming with GO command • 1–11, CD–77
  starting or resuming with STEP command • 2–7, CD–208
  suspending with breakpoint • 2–10, CD–96
  suspending with exception breakpoint • 8–11, CD–97
  suspending with watchpoint • 2–17, CD–156
$EXIT • 8–16
EXIT command • 2–4, 8–16, CD–69
Exit handler
  debugging • 8–16, CD–69
  execution sequence of • 8–16
  identifying • 8–16, CD–172
EXITLOOP command • 7–10, CD–70
/EXIT qualifier • CD–55
EXPAND command • 6–10, CD–71
Expression
  See Address expression
  See Language expression
EXTRACT command • 6–19, CD–73

F

File
  see Command procedure
  see Initialization file
  see Log file
  see Source file
Final handler • 8–13
/FLOAT qualifier • CD–45, CD–63
FOR command • 7–9, CD–75
%FP • 3–22, D–2
/FULL qualifier • CD–198

G

/G_FLOAT qualifier • CD–45, CD–63
/GENERATE qualifier • CD–53
Global symbol
  see Symbol
Global symbol table
  see GST
GO command • 1–11, CD–77
GST (global symbol table)
  creating • 4–4
  shareable image • 4–12

H

/H_FLOAT qualifier • CD–45, CD–63
Handler
  condition • 8–13
Help
  online • CD–79
HELP command • 1–7, CD–79
%HEX • 3–12, D–4
/HEXADECIMAL qualifier • 3–11, CD–58, CD–60, CD–63
/HIDE qualifier • CD–53, CD–104
/HOLD qualifier • CD–142, CD–198
Hyphen (-)
  line-continuation character • CD–4
  subtraction operator • D–6

I

Identifier
  search string • 5–6
/IDENTIFIER qualifier • 5–6, CD–90
IF command • 7–10, CD–81
/IF_STATE qualifier • 7–9, CD–38
Image
  see also Shareable image
  privileged, securing • 4–5
  shareable, debugging • 4–11
Indirection operator
  See Contents-of operator
Initialization
  debugging session • 2–1, 8–7
Initialization code • 8–9
Initialization file
  debugger • 7–4, D–1
Input, debugger
  DBG$INPUT • 8–5, D–1
/INPUT qualifier • 6–17, CD–92, CD–129, CD–206
Instruction
  depositing • 3–19, 3–21
  display (INST) • 6–6, C–5
  display kind • 6–13, C–1
  examining • 3–19
  operand • 3–19, CD–64, CD–120
  replacing • 3–21
/INSTRUCTION qualifier • 6–6, 6–17, CD–14, CD–25, CD–45, CD–64, CD–92, CD–98, CD–148, CD–208
Integer type • 3–14, 3–24, 3–26
Interrupt
  debugging session • 2–5, CD–30
  program • 2–3, CD–30
/INTO qualifier • CD–98, CD–149, CD–156, CD–209
Invoking
  debugger • 2–1

J

/JSB qualifier • 2–14, CD–98, CD–149, CD–209

K

Key definition
  creating • 7–8, CD–37
  debugger predefined • B–1
  deleting • 7–8, CD–42
  displaying • 7–8, CD–174
Keypad mode • 7–8, CD–37, CD–120, CD–174, B–1
Key state • 7–8, CD–37, CD–174, B–1

L

%LABEL • 2–11, D–5
Language
  current • 3–10, CD–113
  identifying • CD–176
  multilanguage program • 8–7
  setting • 3–10, CD–113
  support by debugger • E–1
Language expression
  compared to address expression • 3–7
  DEPOSIT command • 3–3, CD–44
  EVALUATE command • 3–5, CD–58
  FOR command • 7–9, CD–75
  IF command • 7–10, CD–81
  REPEAT command • 7–10, CD–85
  WHEN clause • 2–15
  WHILE command • 7–10, CD–216
Language-Sensitive Editor • CD–55
Last-chance handler • 8–13
/LEFT qualifier • CD–71, CD–82, CD–87
Lexical function
  see Built-in symbol
LIB$INITIALIZE • 8–9
%LINE • D–5
  EXAMINE command • 3–19
  EXAMINE/SOURCE command • 5–4
  GO command • CD–77
  SET BREAK command • 2–11
  SET TRACE command • 2–11
  STEP command • 2–7
Line mode • CD–120
Line number
  see also %LINE
  source display • 5–1, 5–3, 5–4
  traceback information • 1–13, 4–3

/LINE qualifier • 2–14, CD–14, CD–25, CD–64, CD–98, CD–149, CD–209
LINK command • 4–4, 5–1
  shareable image • 4–11
/LIST qualifier • 5–1
/LOCAL qualifier • 7–6, CD–35, CD–40, CD–195
Local symbol
  see Symbol
/LOCK_STATE qualifier • CD–38
Log file
  as command procedure • 7–5
  debugger • 7–5, CD–126
  name • 7–5, CD–115, CD–177
Logical name
  debugger • D–1
Logical predecessor • 3–8, 3–19, D–4
Logical successor • 3–8, 3–19, D–4
/LOG qualifier • CD–38, CD–42
/LONGWORD qualifier • CD–46, CD–64

M

Margin
  source display • 5–8, CD–116, CD–178
/MARK_CHANGE qualifier • CD–53, CD–104
Memory
  effect of debugger • 2–23
MicroVAX
  see VAXstation
Mode
  CANCEL MODE • CD–18
  dynamic • 4–6, 4–13
  SET MODE [NO]DYNAMIC command • 4–6, 4–13, CD–120
  SET MODE [NO]G_FLOAT command • CD–120
  SET MODE [NO]KEYPAD command • 7–8, CD–121
  SET MODE [NO]LINE command • CD–121
  SET MODE [NO]OPERANDS command • 3–19, CD–121
  SET MODE [NO]SCREEN command • 6–1, CD–121
  SET MODE [NO]SCROLL command • CD–121
  SET MODE [NO]SEPARATE command • 8–5, CD–120
  SET MODE [NO]SYMBOLIC command • 3–13, CD–120
  SHOW MODE • CD–180
/MODIFY qualifier • CD–98, CD–149
Module
  see also Shareable image
  canceling • 4–6, CD–19
  information about • 4–6, CD–181
  setting • 4–5, CD–123
  traceback information • 4–3
/MODULE qualifier • CD–23, CD–136
MOVE command • 6–9, CD–82
Multilanguage program
  debugging • 8–7

N

%NAME • D–3
%NEXTDISP • C–6
%NEXTINST • C–7
%NEXTLOC • 3–8, D–4
Next location
  See Logical successor
%NEXTOUTPUT • C–7
/NEXT qualifier • 5–6, CD–90
%NEXTSCROLL • C–7
%NEXTSOURCE • C–7
%NEXT_TASK • D–9
Nonstatic variable • 2–19, 3–1
/NOOPTIMIZE qualifier • 4–2, 8–1
NOP (No Operation) instruction • 3–21

O

Object code • 8–1
Object module • 4–2, 5–1
%OCT • 3–12, D–4
/OCTAL qualifier • 3–11, CD–58, CD–60, CD–64
/OCTAWORD qualifier • CD–46, CD–64
Operand
  instruction • 3–19, CD–64, CD–120
/OPERANDS qualifier • 3–19, CD–64, CD–120
Operator
  address expression • D–5
  language expression • E–1
Optimization
  effect on debugging • 8–1
/OPTIMIZE qualifier • 4–2, 8–1
/OPTIONS qualifier • 4–11
Output
  configuration, displaying • 7–2, 7–5, CD–184

Output (cont'd.)
configuration, setting • 7–2, 7–5, CD–126
debugger, DBG$OUTPUT • 8–5, D–1
display (OUT) • 6–5, C–4
display kind • 6–14, C–1
/OUTPUT qualifier • 6–17, CD–93, CD–129, CD–206
/OVER qualifier • CD–99, CD–149, CD–157, CD–209
/OVERRIDE qualifier • 3–25, CD–21, CD–27, CD–129, CD–154, CD–185, CD–203
Override type • 3–25

P

/PACKED qualifier • CD–46, CD–64
%PAGE • C–6
/PAGE qualifier • 6–20, CD–145
Parameter
debugger command procedure • 7–2, CD–32
%PARCNT • 7–2, D–3
Pasteboard • 6–3
Path name
abbreviating • 4–9
numeric • 4–9
relation to symbol • 4–8
symbol search • 4–7
syntax • 4–8
to specify debugger scope • 4–8
%PC
see PC
PC (program counter)
built-in symbol (%PC) • 3–22, D–2
content of • 1–10, 3–19
EXAMINE/INSTRUCTION command • 6–6, 6–14, C–5
EXAMINE/OPERANDS command • 3–19
EXAMINE/SOURCE command • 5–4, 6–4, 6–16, 6–18, C–4
scope • 4–7
SHOW CALLS display • 1–13, CD–166
Period (.)
contents-of operator • 3–6, 3–19, D–6
current entity • 3–8, D–4
Pointer type • 3–18
/POP qualifier • CD–53, CD–105
debugger window (VAXstation) • CD–128
Predecessor
See Logical predecessor
Previous location
See Logical predecessor
%PREVLOC • 3–8, D–4
Primary handler • 2–22, 8–13
/PRIORITY qualifier • CD–143, CD–199
Program
display kind • 6–16, C–1
Program counter
see PC
/PROGRAM qualifier • 6–17, CD–93
Prompt
debugger (DBG>) • 1–6, CD–128
display (PROMPT) • 6–5, C–4
/PROMPT qualifier • 6–18, CD–93
Pseudo-display name • C–6
%PSL • 3–22, D–2
PSL (processor status longword) • 3–23
/PSL qualifier • CD–64
/PSW qualifier • CD–64
/PUSH qualifier • CD–53, CD–105

Q

/QUADWORD qualifier • CD–46, CD–64
QUIT command • 2–4, CD–84
Quotation mark (")
ASCII string delimiter • 3–16
instruction delimiter • 3–21

R

Radix
canceling • CD–21
conversion • 3–10, D–4
current • 3–10, CD–129
displaying • CD–185
multilanguage program • 8–8
setting • CD–129
specifying • 3–10
Range
colon (:) • 3–17, CD–62
Real type • 3–14
Record
source line correlation • 5–1
Record type • 3–18
/REFRESH qualifier • CD–53
Register
DEPOSIT command • 3–22

Register (cont'd.)
display (REG) • 6–7, C–5
display kind • 6–15, C–1
EXAMINE command • 3–22
PSL • 3–23
symbol • D–2
variable • 2–19, 3–1
/RELATED qualifier • CD–19, CD–123, CD–181
/REMOVE qualifier • CD–53, CD–105
REPEAT command • 7–10, CD–85
/RESTORE qualifier • CD–143
RETURN key
logical successor • 3–8, D–4
/RETURN qualifier • CD–99, CD–150, CD–209
/RIGHT qualifier • CD–71, CD–82, CD–87
Routine
calling • 7–11, CD–10
EXAMINE/SOURCE command • 5–4
multiple invocations of • 4–9
SET BREAK command • 2–11
SET TRACE command • 2–11
SHOW CALLS command • 1–13
traceback information • 4–3
RST (run-time symbol table) • 4–5
and symbol search • 4–7
deleting symbol records in • 4–6, CD–19
displaying modules in • 4–6, CD–181
displaying symbols in • 4–8, CD–195
inserting symbol records in • 4–6, CD–123
shareable image • 4–13
RUN command • 2–1, 2–2, 4–4
see also Execution
shareable image • 4–12
Run-time symbol table
see RST

S

SAVE command • 6–19, CD–86
Scalar type • 3–14
Scope
canceling • 4–10, CD–22
current • 4–10, CD–131
displaying • 4–10, CD–186
PC • 4–7
SEARCH command • 5–6, CD–89
search list • 4–7, 4–10, CD–131, CD–186
setting • 4–10, CD–131
specifying with path name • 4–8
Scope (cont'd.)
TYPE command • 5–4, CD–214
Screen display
see Display
Screen management
debugging screen oriented program • 8–5
Screen mode • 6–1, CD–120
summary reference information • C–1
Screen oriented program
debugging • 8–5
Screen size
displaying • 6–20, CD–201
%PAGE, %WIDTH symbols • C–6
setting • 6–20, CD–145
/SCREEN_LAYOUT qualifier • CD–73
SCROLL command • 6–8, CD–87
Scroll mode • CD–120
/SCROLL qualifier • 6–18, CD–93
SEARCH command • 5–6, CD–89
displaying default qualifiers for • 5–7, CD–188
setting default qualifiers for • 5–7, CD–134
Search list
scope • 4–7, 4–10, CD–131, CD–186
source file • 5–2, CD–23, CD–136, CD–191
Security
image • 4–5
SELECT command • 6–16, CD–92
Semicolon (;)
command separator • CD–4
Separate window
debugger (VAXstation) • 8–5, CD–120
SET ATSIGN command • 7–2, CD–95
SET BREAK command • 2–10, 5–7, 8–10, CD–96
SET DEFINE command • 7–6, CD–102
SET DISPLAY command • 6–10, CD–103
SET EDITOR command • CD–107
SET EVENT_FACILITY command • CD–109
SET IMAGE command • 4–14, CD–110
effect on symbol definitions • CD–36
SET KEY command • 7–9, CD–112
SET LANGUAGE command • 3–10, CD–113
SET LOG command • 7–5, CD–115
SET MARGINS command • 5–8, CD–116
SET MAX_SOURCE_FILES command • 5–3, CD–119
SET MODE command • CD–120
SET MODE [NO]DYNAMIC command • 4–6, 4–13, CD–120
SET MODE [NO]G_FLOAT command • CD–120
SET MODE [NO]KEYPAD command • 7–8, CD–120, B–1

SET MODE [NO]LINE command • CD–120
SET MODE [NO]OPERANDS command • 3–19, CD–120
SET MODE [NO]SCREEN command • 6–1, CD–120
SET MODE [NO]SCROLL command • CD–120
SET MODE [NO]SEPARATE command • 8–5, CD–120
SET MODE [NO]SYMBOLIC command • 3–13, CD–120
SET MODULE command • 4–6, CD–123
SET OUTPUT command • CD–126
SET OUTPUT [NO]LOG command • 7–5, CD–126
SET OUTPUT [NO]SCREEN_LOG command • 7–5, CD–126
SET OUTPUT [NO]TERMINAL command • CD–126
SET OUTPUT [NO]VERIFY command • 7–2, CD–126
SET PROMPT command • CD–128
SET RADIX command • 3–10, 8–8, CD–129
SET SCOPE command • 4–10, 5–4, CD–131
SET SEARCH command • 5–7, CD–134
SET SOURCE command • 5–2, CD–136
SET STEP command • 5–7, CD–139
SET TASK command • CD–142
SET TERMINAL command • 6–20, CD–145
SET TRACE command • 2–10, 5–7, 8–10, CD–147
SET TYPE command • 3–24, CD–153
SET TYPE/OVERRIDE command • 3–25, CD–153
SET WATCH command • 2–17, 5–7, CD–156
SET WINDOW command • 6–12, CD–161
/SET_STATE qualifier • 7–9, CD–38
Shareable image
  see also Module
  CANCEL IMAGE command • 4–13, CD–17
  debugging • 4–11
  SET BREAK/INTO command • 2–14, CD–99
  SET IMAGE command • 4–14, CD–110
  SET STEP INTO command • 2–9, CD–140
  SET TRACE/INTO command • 2–14, CD–150
  SHOW IMAGE command • 4–13, CD–173
  STEP/INTO command • CD–209
/SHAREABLE qualifier • 4–11
/SHARE qualifier • 2–14, CD–99, CD–150, CD–181, CD–209
SHOW AST command • 8–17, CD–163
SHOW ATSIGN command • 7–2, CD–164
SHOW BREAK command • 2–11, CD–165
SHOW CALLS command • 1–13, 2–3, 2–5, 8–11, 8–17, CD–166
SHOW DEFINE command • 7–6, CD–168
SHOW DISPLAY command • 6–9, CD–169
SHOW EDITOR command • CD–170
SHOW EVENT_FACILITY command • 2–16, CD–171
SHOW EXIT_HANDLERS command • 8–16, CD–172
SHOW IMAGE command • 4–13, CD–173
SHOW KEY command • 7–8, CD–174
SHOW LANGUAGE command • 3–10, CD–176
SHOW LOG command • 7–5, CD–177
SHOW MARGINS command • 5–9, CD–178
SHOW MAX_SOURCE_FILES command • 5–3, CD–179
SHOW MODE command • CD–180
SHOW MODULE command • 4–6, CD–181
SHOW OUTPUT command • 7–2, 7–5, CD–184
SHOW RADIX command • 3–10, CD–185
SHOW SCOPE command • 4–10, CD–186
SHOW SEARCH command • 5–7, CD–188
SHOW SELECT command • 6–18, CD–189
SHOW SOURCE command • 5–2, CD–191
SHOW STACK command • 8–12, CD–193
SHOW STEP command • 2–8, CD–194
SHOW SYMBOL command • 4–8, CD–195
SHOW SYMBOL/DEFINED command • 7–6
SHOW TASK command • CD–198
SHOW TERMINAL command • 6–20, CD–201
SHOW TRACE command • 2–11, CD–202
SHOW TYPE command • 3–24, CD–203
SHOW WATCH command • 2–17, CD–204
SHOW WINDOW command • 6–12, CD–205
/SILENT qualifier • 2–15, CD–99, CD–150, CD–157, CD–209
/SIZE qualifier • CD–54, CD–105
Slash (/)
  division operator • D–6
SMG$
  debugging screen oriented program • 8–5
Source directory
  displaying • 5–2, CD–191
  search list • 5–2, CD–23, CD–136
Source display • 1–7, 5–1, 6–1
  discrepancies in • 8–1
  display kind • 6–15, C–1
  EXAMINE/SOURCE command • 5–4, 6–4, 6–15, C–4
  line-oriented • 5–3
  margins in • 5–8, CD–178
  not available • 1–10, 5–1, 6–4, CD–136, C–4
  SEARCH command • 5–6, CD–89
  SET BREAK command • 5–7
  SET STEP command • 5–7, CD–139
  SET TRACE command • 5–7

Source display (cont'd.)
SET WATCH command • 5–7
SRC, predefined • 6–4, C–4
STEP command • 5–7
TYPE command • 5–3, CD–214
Source file
correct version of • CD–136, CD–191
defined • 5–2
file specification • 5–2
location • 5–2, CD–23, CD–136, CD–191
maximum number • 5–3, CD–119, CD–179
not available • 5–2, CD–136
Source line correlation • 5–1
/SOURCE qualifier • 5–4, 5–8, 6–4, 6–18, CD–65, CD–93, CD–99, CD–150, CD–157, CD–209
%SOURCE_SCOPE • 6–16, C–4
%SP • 3–22, D–2
SPAWN command • 2–6, CD–206
SRC source display • 6–4, C–4
SS$_DEBUG condition • CD–30, D–1
Stack
see also Call stack
variable • 2–19, 3–1
/START_POSITION qualifier • CD–107
/STATE qualifier • 7–8, CD–43, CD–112, CD–175, CD–199
/STATIC qualifier • CD–157
Static variable • 2–19, 3–1
/STATISTICS qualifier • CD–199
STEP command • 2–7, 5–7, CD–208
displaying default qualifiers for • CD–194
setting default qualifiers for • CD–139
STOP command • 2–4
/STRING qualifier • 5–6, CD–90
String type • 3–16, 3–26
Successor
See Logical successor
Symbol
see also DST, GST, RST
built-in • C–6, D–2
compiler generated type • 3–4
defining • 7–6, CD–36
displaying • 4–8, 7–6, CD–36, CD–195
global • 4–3, 4–9
image setting • 4–13
local • 4–3
module setting • 4–5
multiply-defined • 4–7
not in symbol table • 4–5, 4–14
not unique • 4–8
relation to address expression • 3–4
Symbol (cont'd.)
relation to path name • 4–8
search conventions • 4–7
shareable image • 4–13
SHOW SYMBOL command • 4–8
traceback information • 4–3
universal • 4–5, 4–11
Symbolic mode • 3–13, CD–120
/SYMBOLIC qualifier • 3–13, CD–65
SYMBOLIZE command • 2–13, 3–13, CD–212
Symbol record
see Symbol
Symbol table
see DST, GST, RST
/SYSTEM qualifier • 2–14, CD–99, CD–150, CD–209
System space
SET BREAK command • CD–99
SET STEP command • CD–140
SET TRACE command • CD–150
STEP command • CD–209

T

%TASK • D–9
Tasking
SET TASK command • CD–142
SHOW TASK command • CD–198
/TASK qualifier • CD–46, CD–65
/TEMPORARY qualifier • CD–99, CD–150, CD–157
Terminal
debugger, input/output • 8–5
Terminal screen size
see Screen size
/TERMINATE qualifier • 7–8, CD–38
Termination
debugging session • 2–4, CD–69, CD–84
execution of handlers at • 8–16
/TIME_SLICE qualifier • CD–143, CD–199
/TOP qualifier • CD–87
Traceback
compiler option • 4–3
link option • 4–4
SHOW CALLS display • 1–13
/TRACEBACK qualifier • 2–3, 4–4, 4–5
shareable image • 4–12
Tracepoint
canceling • 2–17, CD–25
defined • 2–10

Tracepoint (cont'd.)
delayed triggering of • 2–15, CD–147
displaying • CD–202
DO clause • 2–15
exception • 8–10, CD–147
setting • 2–10, CD–147
source display at • 5–7
WHEN clause • 2–15
Transfer address • 2–1, 8–7
Type
address expression • 3–4, 3–24
array • 3–16
ASCII string • 3–16, 3–26
compiler generated • 3–4, 3–14
conversion, numeric • 3–7
current • 3–24, CD–153, CD–203
displaying • CD–203
integer • 3–14, 3–26
override • 3–25, CD–153
pointer • 3–18
real • 3–14
record • 3–18
scalar • 3–14
SET TYPE command • 3–24, CD–153
symbolic address expression • 3–4
VAX instruction • 3–19
TYPE command • 5–3, CD–214
Type override • 3–25, CD–27, CD–154, CD–203
/TYPE qualifier • 3–27, CD–46, CD–65, CD–196

U

Universal symbol
see Symbol
/UP qualifier • CD–71, CD–82, CD–87
/USE_CLAUSE qualifier • CD–196

V

/VALUE qualifier • 7–6, CD–35
Variable
as override type • 3–27
examining and depositing • 3–14
initialized • 3–1
nonstatic • 2–19, 3–1
optimized code • 8–1
register • 2–19, 3–1
stack local • 2–19, 3–1
Variable (cont'd.)
static • 2–19
uninitialized • 2–23
Variable name
address expression • 3–7
DEPOSIT command • 3–3
EXAMINE command • 3–2
language expression • 3–6
SET WATCH command • 2–17
VAX Language-Sensitive Editor • CD–55
VAXstation
debugger commands for • CD–5
debugging screen oriented program • 8–5
popping debugger window • CD–128
screen size • 6–20, CD–145
separate debugger window • 8–5, CD–120
Verify
SET OUTPUT VERIFY command • CD–126
Virtual memory address
examining • 3–13
obtaining • 2–13, 3–12
specifying eventpoint • 2–13
symbolizing • 3–13
/VISIBLE qualifier • CD–143
%VISIBLE_TASK • D–9

W

/WAIT qualifier • CD–206
Watchpoint
aggregate • 2–18
canceling • CD–28
defined • 2–17
displaying • CD–204
nonstatic (stack or register) variable • 2–19
setting • 2–17, CD–156
source display at • 5–7
static variable • 2–19
WHEN clause
example • 2–15
format • CD–4
WHILE command • 7–10, CD–216
%WIDTH • C–6
/WIDTH qualifier • 6–20, CD–145
Window
debugger, popping (VAXstation) • CD–128
debugger, separate (VAXstation) • 8–5, CD–120
screen mode, creating definition for • 6–12, CD–161

Window (cont'd.)
screen mode, defined • 6–2
screen mode, deleting definition of • 6–12, CD–29
screen mode, identifying • 6–12, CD–205
screen mode, predefined • CD–205, C–7
screen mode, specifying • 6–11
/WORD qualifier • CD–46, CD–65
Workstation
see VAXstation

Reader's Comments

VMS Debugger Manual
AA–LA59A–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

-- 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

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here ----------