

PDP-11 Symbolic Debugger COBOL-81 User's Guide

Order Number: AA-FA63A-TK

December 1985

Revision/Update Information:	This is a new manual.
Operating System and Version:	See the Preface for detailed information.
Software Version:	PDP-11 Symbolic Debugger Version 2.0

**digital equipment corporation
maynard, massachusetts**

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

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


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

Copyright ©1985 by Digital Equipment Corporation

All Rights Reserved.
Printed in U.S.A.

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:

DATATRIEVE	Micro/RSTS	RT
DEC	Micro/R SX	UNIBUS
DECmate	PDP	VAX
DECnet	P/OS	VAXcluster
DECUS	Professional	VMS
DECwriter	QBUS	VT
FMS	Rainbow	Work Processor
MASSBUS	RSTS	
MicroPDP-11	RSX	

ZK2957

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

Contents

PREFACE	vii
----------------	------------

CHAPTER 1 INCLUDING DEBUGGER SUPPORT	1-1
1.1 HOW TO INCLUDE THE DEBUGGER IN YOUR TASK	1-1
1.1.1 Using a Nonoverlaid Configuration _____	1-2
1.1.2 Replacing the Bundled Debugger _____	1-3
1.1.3 Using an Overlaid Configuration _____	1-5
1.1.4 Exiting the Debugger _____	1-6
1.2 FEATURES OF THE DEBUGGER	1-7

CHAPTER 2 CONTROLLING DEBUGGER INPUT AND OUTPUT	2-1
2.1 SETTING THE DEFAULT LANGUAGE	2-1
2.2 CHANGING THE DEFAULT OUTPUT	2-2
2.2.1 SET OUTPUT Command Parameters _____	2-2
2.2.2 The SHOW OUTPUT and CANCEL OUTPUT Commands _____	2-2
2.3 USING LOG FILES	2-2
2.3.1 Log File Example _____	2-3
2.3.2 The SHOW LOG Command _____	2-3
2.4 USING INDIRECT COMMAND FILES	2-3

CHAPTER 3	DEFINING SYMBOLS	3-1
3.1	KINDS OF SYMBOLS	3-1
3.1.1	Permanent Symbols _____	3-1
3.1.2	Program Symbols _____	3-2
3.1.3	Defined Symbols _____	3-3
3.2	MAKING SYMBOLS UNIQUE	3-3
3.3	ADJUSTING THE DEBUGGER'S SCOPE	3-4
3.3.1	SET SCOPE Command Parameter _____	3-5
3.3.2	The SHOW SCOPE and CANCEL SCOPE Commands	3-5

CHAPTER 4	CONTROLLING PROGRAM EXECUTION	4-1
4.1	DISPLAYING INFORMATION ON ACTIVE ROUTINE CALLS	4-1
4.2	THE EFFECTS OF BREAKPOINTS AND TRACEPOINTS	4-2
4.2.1	SET BREAK and SET TRACE Command Qualifiers ____	4-3
	4.2.1.1 The /AFTER:n Qualifier • 4-3	
	4.2.1.2 The /CALLS Qualifier • 4-4	
	4.2.1.3 The /RETURN Qualifier • 4-4	
4.2.2	SET BREAK and SET TRACE Command Parameters _	4-4
	4.2.2.1 The Address Parameter • 4-4	
	4.2.2.2 The DO Parameter • 4-5	
4.2.3	Commands Related to SET BREAK and SET TRACE _	4-5

CHAPTER 5	STARTING THE PROGRAM	5-1
------------------	-----------------------------	------------

5.1	EXECUTING A SPECIFIED NUMBER OF COMMANDS	5-1
5.1.1	STEP Command Qualifiers _____	5-2
5.1.2	STEP Command Parameter _____	5-2
5.2	CHANGING THE DEFAULT STEP CONDITIONS	5-3
5.2.1	SET STEP Command Parameters _____	5-3
5.2.2	The SHOW STEP and CANCEL STEP Commands ____	5-3
5.3	EXECUTING AN UNDETERMINED NUMBER OF COMMANDS	5-4

CHAPTER 6	MANIPULATING DATA	6-1
------------------	--------------------------	------------

6.1	DATA TYPES IN THE DEBUGGER	6-1
6.2	DEBUGGER MODES	6-2
6.3	DETERMINING THE VIRTUAL ADDRESS OF SYMBOLS	6-3
6.3.1	EVALUATE Command Qualifiers _____	6-4
6.3.2	EVALUATE Command Parameters _____	6-4
6.4	VALUE EXPRESSIONS	6-4
6.5	DISPLAYING MEMORY LOCATIONS	6-5
6.6	REFERENCING VARIABLE NAMES CONTAINING HYPHENS	6-5
6.7	ALTERING MEMORY LOCATIONS	6-6
6.7.1	DEPOSIT Command Qualifiers and Parameters ____	6-6
6.7.2	Depositing ASCII Strings _____	6-7

CHAPTER 7	COBOL-81 INTERACTIVE DEBUGGING EXAMPLE	7-1
------------------	---	------------

7.1	THE COBOL-81 PROGRAM	7-1
-----	----------------------	-----

7.2	THE SAMPLE DEBUGGING SESSION	7-3
-----	------------------------------	-----

INDEX

TABLES

3-1	Debugger Permanent Symbols	3-1
-----	----------------------------	-----

Preface

Intended Audience

This manual is intended for COBOL-81 programmers who have read and understand the *PDP-11 Symbolic Debugger User's Guide*, and know how to use the host operating system.

Operating Systems and Versions

The PDP-11 Symbolic Debugger runs on the following operating systems and versions:

- VAX/VMS Version 4.0 or higher
- VAX-11 RSX Version 2.0
- RSX-11M Version 4.1 or higher
- RSX-11M-PLUS Version 2.1 or higher
- Micro/RSX Version 1.1 or higher
- RSTS/E Version 9.0 or higher
- Micro/RSTS Version 2.0 or higher
- P/OS Version 2.0 with Professional Host Tool Kit Version 2.0 or higher
- P/OS Version 2.0 with PRO/Tool Kit Version 2.0 or higher

Structure of This Document

This manual is organized as follows:

- Chapter 1 explains how to include support for the debugger in your task and describes the commands that invoke the debugger. It also lists the major debugger features for all users as well as the debugger features that are specific to COBOL-81.
- Chapter 2 explains how to configure the debugger's default output, make a record of a debugging session, and use a command file to control the debugger.

- Chapter 3 describes the symbols the debugger recognizes and explains how to define your own symbols. It also discusses strategies for making symbols unique.
- Chapter 4 explains how to set breakpoints and tracepoints in your program.
- Chapter 5 describes two methods of executing your program in the debugger.
- Chapter 6 discusses the data types the debugger recognizes, the two modes of debugger operation, and a command that helps you determine memory addresses and perform arithmetic. It also explains how to examine and alter memory locations.
- Chapter 7 gives an example of debugger use with a COBOL-81 program

Associated Documents

The following list describes the content of each manual in the PDP-11 Symbolic Debugger documentation set:

- *PDP-11 Symbolic Debugger User's Guide*. This manual explains general use of the debugger with all supported languages.
- *PDP-11 Symbolic Debugger Installation Guide*. This manual explains the debugger installation procedure on all supported operating systems.
- *PDP-11 Symbolic Debugger Quick Reference Guide*. The quick reference manual lists the syntax of each debugger command and its qualifiers and parameters.
- *PDP-11 Symbolic Debugger FORTRAN-77 User's Guide*. This manual gives information to debugger users who program in FORTRAN-77.

NOTE

Where language-specific exceptions to the general case exist, the information given in this manual, specific to COBOL-81, takes precedence over general information presented elsewhere.

Conventions Used in This Document

The following conventions are followed throughout this manual:

Convention	Meaning
UPPERCASE	Uppercase words and letters in examples indicate that you type the word or letter exactly as shown.
lowercase	Lowercase words and letters in examples indicate that you substitute a word or value of your choice.
[]	Brackets in examples indicate optional elements.
n	A lowercase n indicates that you must substitute a value.
RSX-11	RSX-11 is used as a generic term for the RSX-11M, RSX-11M-PLUS, and Micro/RSX operating systems.
CTRL/a	The symbol CTRL/a indicates that you hold down the CTRL key while you simultaneously press the specified letter key. For example, CTRL/Z indicates that you hold down the CTRL key and press the letter Z.
RET	The RET symbol indicates that you press the RETURN key.

Including Debugger Support

The PDP-11 Symbolic Debugger is a powerful tool that helps you find logical and programming errors in a successfully compiled program that does not run correctly. When you are ready to use the debugger on a program, you must include it in your task. This chapter explains how to include debugger support in your task and what commands you issue to invoke the debugger. It summarizes the general debugger features listed in the *PDP-11 Symbolic Debugger User's Guide* and lists those debugger features that are available only for COBOL-81 users.

1.1 How to Include the Debugger in Your Task

The PDP-11 Symbolic Debugger supports both a nonoverlaid kernel and an overlaid kernel. You must include one of these two kernels in your task to use the debugger on your program. The nonoverlaid kernel occupies about 5000 bytes of user program space. The overlaid kernel occupies less than 4000 bytes of user program space. You may use the overlaid kernel unless your program is overlaid and you are loading your overlay segments manually. In this case, you must use the nonoverlaid debugger kernel because the overlaid debugger kernel uses autoloading, and you may not mix the two loading methods in one task.

There are three ways to include debugger support in your task:

1. You can include a nonoverlaid debugger kernel in your task. Refer to Section 1.1.1 for the commands to invoke the debugger with a nonoverlaid kernel.

2. If you want to use the overlaid debugger kernel, you can replace the debugger that is bundled with your COBOL-81 compiler with the PDP-11 Symbolic Debugger. See Section 1.1.2 for instructions on replacing the debugger that is bundled with your compiler.
3. If you want to use the overlaid debugger kernel, but still want to access the debugger bundled with your COBOL-81 compiler, refer to Section 1.1.3.

1.1.1 Using a Nonoverlaid Configuration

When you want to invoke the debugger with a nonoverlaid kernel, you must compile, link, and run your program. When you compile your program, you should create a listing file so you can reference source code line numbers and follow program flow during the debugging session. You may use the following commands with any COBOL-81 task to invoke the debugger with a nonoverlaid kernel:

FOR MCR USERS

```
>C81 myprog,myprog=myprog/DEB
>BLD myprog=myprog/DEB
>EDIT myprog.ODL
>TKB @myprog
>RUN myprog
```

FOR DCL USERS

```
$ COBOL/DEBUG/LIST myprog
$ MCR BLD myprog=myprog/DEB
$ EDIT myprog.ODL
$ LINK @myprog
$ RUN myprog
```

The PDP-11 Symbolic Debugger ODL file (PDPDBG.ODL) always invokes the overlaid kernel. If you want to use the nonoverlaid kernel and use the BLDODL utility contained in the COBOL-81 kit, you will have to

1. Remove the \$DALL co-tree from the ODL file
2. Replace the reference to \$DROOT with LB:[1,1]PDPDBG/DA
3. Include the appropriate COBOL-81 OTS
4. Remove the reference to "@C81DBx.ODL"

In response to the final command, the file MYPROG.CMD runs to produce a symbol table that the debugger requires for full symbolic capability.

Use the following commands instead of the preceding ones only if *both* the debugger and your COBOL-81 task are nonoverlaid:

FOR MCR USERS

```
>C81 myprog,myprog=myprog/DEB  
>TKB myprog,myprog=myprog,LB:[1,1]PDPDBG/DA,LB:[1,1]C81LIB/LB  
>RUN myprog
```

The TKB command in the preceding example contains two commas between the file names on the left side of the equal sign because one of the TKB command parameters has been omitted.

FOR DCL USERS

```
$ COBOL/DEBUG/LIST myprog  
$ LINK/DEBUG=LB:[1,1]PDPDBG/SYMBOL myprog,LB:[1,1]C81LIB/LIBR  
$ RUN myprog
```

NOTE

RSTS users must replace the RUN command with the DEBUG command as follows:

```
$ DEBUG myprog
```

Also, RSTS users should substitute all references to "LB"[1,1]" with "LB:."

1.1.2 Replacing the Bundled Debugger

If you do not want to use the debugger that comes bundled with your COBOL-81 compiler after you install the PDP-11 Symbolic Debugger and you want to use the overlaid PDP-11 Symbolic Debugger kernel, you may replace your bundled debugger with the PDP-11 Symbolic Debugger.

To replace your bundled debugger, copy the PDP-11 Symbolic Debugger kernel ODL file (LB:[1,1]PDPDBG.ODL) to the ODL file for the debugger bundled with your compiler by issuing the following command:

```
$ COPY LB:[1,1]PDPDBG.ODL LB:[1,1]C81DBx.ODL
```

You must complete the second file name by replacing the character x with one of three letters, depending on what kind of OTS and instruction set your compiler supports. There are three forms of the COBOL-81 ODL file:

C81DBG.ODL	Resident libraries, and either CIS (Commercial Instruction Set) or non-CIS
C81DBC.ODL	Nonresident libraries and CIS
C81DBN.ODL	Nonresident libraries and non-CIS

Once you copy the file PDPDBG.ODL, you may invoke the debugger by compiling, linking, and running your program. No edits are needed. You should create a listing file of your source code and refer to it during the debugging session to follow program flow and to reference source code line numbers. For example, to compile, link, and run a program called MYPROG.CBL, issue the following commands:

FOR MCR USERS

```
>C81 myprog,myprog=myprog/DEB
>BLD myprog=myprog/DEB
>TKB @myprog
>RUN myprog
```

After your program compiles and links, the command file myprog.CMD is run to produce the symbol table that the debugger requires for full symbolic capability.

FOR DCL USERS

```
$ COBOL/DEBUG/LIST myprog
$ MCR BLD myprog=myprog/DEB
$ LINK @myprog
$ RUN myprog
```

NOTE

RSTS users must replace the RUN command with the DEBUG command as follows:

```
$ DEBUG myprog
```

RSTS users should also substitute all references to "LB"[1,1]" with "LB:."

1.1.3 Using an Overlaid Configuration

If you want to use the overlaid PDP-11 Symbolic Debugger kernel and you still want to have access to the debugger that is bundled with your COBOL-81 compiler, you must create and edit an ODL file for your program. This section explains how to create and edit this ODL file and the commands you use to invoke the debugger.

To create an ODL file that includes debugger support for a program called MYPROG.CBL, type one of the following command lines:

FOR MCR USERS

```
>BLD myprog=myprog/DEB
```

FOR DCL USERS

```
$ MCR BLD myprog=myprog/DEB
```

The command that is appropriate to your CLI causes the BLDODL utility to create a command file and an ODL file for your program.

You must edit the ODL file that BLDODL creates to include the PDP-11 Symbolic Debugger instead of the debugger that comes bundled with your COBOL-81 compiler. For example, the following is a portion of an ODL file for the program called MYPROG.CBL:

```
@LB:[1,1]C81DBN
USROT$: .FCTR MYPROG-$DROOT
        .ROOT USROT$, $DALL
        .END
```

You must change the line @LB:[1,1]C81DBN because it refers to the COBOL-81 bundled debugger. Note that this file may be named C81DBG or C81DBC, depending on the configuration of your compiler. In either case, modify this line so that it appears as follows:

```
@LB:[1,1]PDPDBG
```

You must also modify the .FCTR directive so that it includes the COBOL-81 library. Edit it so that it appears as follows:

```
USROT$: .FCTR MYPROG-$DROOT-LB:[1,1]C81LIB/LB
```

The edited ODL file you create now correctly references the PDP-11 Symbolic Debugger.

Once you modify your ODL file, you must link and run your program to invoke the debugger. You should create a listing file of your source code and refer to it during the debugging session to follow program flow and to reference source code line numbers. For example, to compile, link, and run a program called MYPROG.CBL, issue the following commands:

FOR MCR USERS

```
>C81 myprog,myprog=myprog/DEB
>BLD myprog=myprog/DEB
>EDIT myprog.ODL
>TKB @myprog
>RUN myprog
```

FOR DCL USERS

```
$ COBOL/DEBUG/LIST myprog
$ MCR BLD myprog=myprog/DEB
$ EDIT myprog.ODL
$ LINK @myprog
$ RUN myprog
```

NOTE

RSTS users must replace the RUN command with the DEBUG command as follows:

```
$ DEBUG myprog
```

RSTS users should also substitute all references to "LB"[1,1]" with "LB:."

1.1.4 Exiting the Debugger

To leave the debugger, type the following command:

```
DBG>EXIT
```

This command causes orderly termination of the debugger on all operating systems.

1.2 Features of the Debugger

The PDP-11 Symbolic Debugger has important features that are available for all debugger users.

- It is interactive.
- It is symbolic.
- It supports overlaid programs.
- It gives online HELP.

In addition, the debugger supports the following features for COBOL-81 users:

- Configures the debugger for COBOL-81 users using the SET LANGUAGE command (see Chapter 2)
- Allows you to correctly qualify names of structured variable (see Chapter 3)
- Recognizes COBOL-81 data types (see Chapter 6)
- Allows you to refer to symbol names that contain hyphens (see Chapter 6)

Controlling Debugger Input and Output

This chapter explains how to set the default programming language to COBOL-81 and describes what happens when this default is set. It also explains how to configure aspects of the debugger input and output format that are not specific to programming in COBOL-81.

2.1 Setting the Default Language

When you enter the debugger, it displays a message indicating the programming language in which it expects your program to be written. If the message does not specify COBOL-81, issue the command

```
DBG>SET LANGUAGE COBOL
```

This command informs the debugger that your program is written in COBOL-81. The debugger uses this information to control how it interprets and displays information. For example, when the language is set to COBOL-81, the debugger interprets input and displays output in ASCII format by default. Also, structured variables are displayed by major row order; if you examine a structured variable, the first item displayed is the item in Row 1, Column 1; the second item is the item in Row 1, Column 2; the third item is in Row 1, Column 3; and so on.

2.2 Changing the Default Output

By default, the debugger's output configuration is NOLOG, TERMINAL, NOVERIFY. You change the default output with the SET OUTPUT command in the format:

```
SET OUTPUT parameter [,parameter [,parameter]]  
[NO]LOG  
[NO]TERMINAL  
[NO]VERIFY
```

2.2.1 SET OUTPUT Command Parameters

The parameters you use with the SET OUTPUT command configure the debugger's output. The [NO]LOG parameter determines whether or not a record of the debugging session is written in a log file. The [NO]TERMINAL parameter determines whether or not the debugger's output displays on your terminal. The [NO]VERIFY parameter determines if indirect commands are displayed on your terminal and/or recorded in your log file before they are executed.

2.2.2 The SHOW OUTPUT and CANCEL OUTPUT Commands

The SHOW OUTPUT command causes a message describing the debugger's current output configuration to be displayed. However, if you set the debugger's output to NOTERMINAL, no message is displayed on your terminal.

The CANCEL OUTPUT command returns the output configuration to the default of NOLOG, TERMINAL, NOVERIFY.

2.3 Using Log Files

When you issue the SET OUTPUT LOG command, the debugger begins logging to a log file called DEBUG.LOG. If you want the debugger to write log information to another file, issue the command

```
DBG>SET LOG filespec
```

This command causes the debugger to write log records to the file named by filespec.

2.3.1 Log File Example

The following is an example of a log file:

```
SHOW OUTPUT
!%DEBUG-I-OUTPUT: noverify, terminal, logging to "DISK$USER:[303,52]MYPROG.LOG;1"
SET LANGUAGE COBOL
SHOW LANGUAGE
!%DEBUG-I-CURRLANGC81, Current language is COBOL-81
SET LOG RECORD
```

This log file is closed after the command SET LOG RECORD is issued. The commands and responses that follow this command are written to a new log file called RECORD.LOG.

2.3.2 The SHOW LOG Command

You can display the name of the log file the debugger is currently using with the SHOW LOG command. If the output is set to NOLOG, the debugger displays a message informing you that it is not writing records to the current log file.

2.4 Using Indirect Command Files

Indirect command files are files that contain a series of debugger commands. Any valid debugger command can be included in an indirect command file, but none of them are checked for valid syntax before they are executed. Instead, the debugger issues an error message when it encounters invalid command syntax in the file and continues execution with the next line in the command file.

You execute an indirect command file as follows:

```
@filespec
```

You can invoke an indirect command file in response to the debugger prompt (DBG>) or in another indirect command file. The default file extension for indirect command files is CMD.

Defining Symbols

The PDP-11 Symbolic Debugger allows you to refer to memory locations and program data symbolically. This chapter explains the symbols the debugger recognizes and how to define symbols.

3.1 Kinds of Symbols

You use symbols to refer to memory locations without having to specify the virtual address of the location. The symbols that the debugger recognizes can be divided into three categories: permanent symbols, program symbols, and defined symbols.

3.1.1 Permanent Symbols

You can refer to the debugger's permanent symbols at any time during a debugging session. Table 3-1 lists these symbols and also describes what they represent.

Table 3-1: Debugger Permanent Symbols

Symbol	Definition
%R0 - %R5	General-purpose registers
%R6 or %SP	Stack pointer
%R7 or %PC	Program counter
%F0 - %F5	Floating-point registers

Table 3–1 (Cont.): Debugger Permanent Symbols

Symbol	Definition
%PS	Processor status word
%FS	Floating-point status word
%LINE nnn	Source code line number
%NAME	name
%SEGMENT name	Overlay segment name
\	Current value
.	Current location
RET	Logical successor
^	Logical predecessor

3.1.2 Program Symbols

When you build your program with debugger support, the task builder defines program symbols for you. These symbols are called program symbols because they refer to records in the symbol table file (STB) for your source code. In the STB file, the program symbol names are associated with virtual addresses.

The STB file contains symbol records for the following program symbols:

- Names of user written routines
- Variable names (but not routine parameter names)
- Source code line numbers

Source code line numbers are a special case because the STB file does not associate them with virtual addresses. Instead, source code line numbers are associated with the program counter (PC).

3.1.3 Defined Symbols

During a debugging session, you can use the DEFINE command to create a new debugger symbol or change an existing symbol, except that you cannot redefine permanent or program symbols. Symbols you define with the DEFINE command remain in effect until you redefine them with another DEFINE command, cancel the definition with the UNDEFINE command, or terminate the debugging session. The DEFINE command has the following format:

```
DEFINE symbol=address
```

The symbol parameter specifies the name you want to use to refer to program data or program addresses. The following restrictions apply to a debugger symbol name:

- It may be composed of only alphanumeric characters (the 26 letters A through Z and the numbers 0 to 9) and dollar signs (\$).
- It may not be more than six characters long.
- It may not begin with a number.

The address parameter identifies the portion of memory to which the symbol refers. It can be either a previously defined symbolic address or a virtual address denoted by a simple address or address expression.

3.2 Making Symbols Unique

When you refer to part of a structured variable in a debugger command, you must use a unique variable name. Two debugger keywords, OF and IN, allow you to qualify COBOL-81 structured variables.

Both OF and IN show the relationship between high-level variable names and the lower-level variable they compose.

For example, consider the following data division:

```
DATA DIVISION.  
FILE SECTION.  
FD  EMPLOYEE  
   LABEL RECORDS ARE OMITTED.  
01  OLDREC  
   05  NUMBER           PIC X(6)  
   05  HOURS            PIC Z9.9  
01  NEWREC  
   05  NUMBER           PIC X(6)  
   05  PAY              PIC $$$V99
```

To examine the contents of the variable NUMBER that is contained in the record OLDREC, issue one of the following commands:

```
DBG>EXAMINE NUMBER IN OLDREC  
DBG>EXAMINE NUMBER OF OLDREC
```

Both command lines are correct because IN and OF are synonyms. (Chapter 6 explains the EXAMINE command.)

If you do not qualify ambiguous names in structured variables, the debugger issues an error message. For example, an error message is issued in response to the following command:

```
DBG>EXAMINE NUMBER  
%DEBUG-E-NAMAMBIG, The data-name used in this command is ambiguous
```

The error occurs because there are two variables defined with the name NUMBER. You must supply a unique name before the debugger can determine which memory location the variable name references.

3.3 Adjusting the Debugger's Scope

If the program you are debugging consists of more than one routine, you must pay attention to the scope of symbols to which you refer because the debugger recognizes only symbols that are in the current scope. The scope of a symbol is the routine in which the symbol is declared.

The default scope is called the *PC scope*. At the beginning of a debugging session, the PC scope is the main routine. The PC scope, however, is dynamic; as you debug your program, the PC scope is always the routine you are currently executing.

If you do not want to use the default scope, you can specify scope by using the SET SCOPE command. The SET SCOPE command establishes the specified program unit as the one to be used for symbol interpretation.

The format of the SET SCOPE command is:

```
SET SCOPE pathname
```

3.3.1 SET SCOPE Command Parameter

The pathname parameter may be a scope prefix, the number 0, or the backslash character (\).

A scope prefix describes a location in terms of its overlay segment name (if any) and routine name. The format of a scope prefix is:

```
[segment-list\]routine
```

The *segment-list* element names the segment that contains the routine to which you are referring. The *routine* element names the routine to which you are setting the scope.

The pathname element routine can be replaced by two symbols: the number 0 or the backslash (\). The 0 symbol specifies that the scope be reset to the default, which is the PC scope. After you issue the command SET SCOPE 0, the scope is dynamic and is always the routine you are debugging. The backslash (\) symbol specifies that symbols referenced without pathnames be interpreted as global symbols.

You can also specify scope using a pathname. The use of pathnames is discussed in the *PDP-11 Symbolic Debugger User's Guide*.

3.3.2 The SHOW SCOPE and CANCEL SCOPE Commands

Two commands, SHOW SCOPE and CANCEL SCOPE, are useful when you are adjusting the scope of the debugger.

To determine the current scope, use the command

```
DBG>SHOW SCOPE
%DEBUG-I-SCOPE, scope: 0 [ = MAIN ]
```

To cancel the scope established by the SET SCOPE command, use the command

DBG>CANCEL SCOPE

The CANCEL SCOPE command causes symbols without scope prefixes to be interpreted as if they occurred in the routine that is currently executing. In its effect, the CANCEL SCOPE command is equivalent to the command SET SCOPE 0.

Controlling Program Execution

Controlling program execution is an important aspect of debugging. To do this effectively, you must know what code is executing and how your program transfers control from one part of your program to another. This chapter explains the commands that help you debug program execution and control.

4.1 Displaying Information on Active Routine Calls

The SHOW CALLS command provides information about the sequence of currently active routine calls. For each call, the debugger displays one line of information. The first line displays information about the current routine; the next line (if there is one) displays information about the routine that called the current routine. The listing ends with information on the routine that originated the path to the current routine.

Each line of information displayed by the debugger contains the following:

- The name of the calling module and routine.
- The name of the called routine.
- The line number of the call.
- The absolute and relative value of the PC in the calling routine at the time that control was transferred. Note that the PC values refer to the location of the instruction following the call.

The format of the SHOW CALLS command is:

```
SHOW CALLS [call-count]
```

The optional call-count parameter is a decimal integer in the range 1 through 32767 that specifies the number of calls to be displayed. If you do not specify the call count, or if the call count exceeds the current number of calls, information on all calls is displayed.

4.2 The Effects of Breakpoints and Tracepoints

Once you decide where the important points in your program are, you are ready to set either breakpoints or tracepoints. This section describes the effects of these event points so you can decide which program controller to use at a specific important program event.

A breakpoint is a program location where the debugger does the following:

1. Suspends program execution immediately before the instruction at the specified location is executed
2. Displays the name or the virtual memory location where execution has been suspended
3. Executes commands in a DO sequence if one was specified in the SET BREAK command (The SET BREAK command is discussed in Section 4.2.1.)
4. Issues its prompt

When a tracepoint is activated, the debugger does the following:

1. Suspends execution immediately before the instruction at the specified location is executed
2. Reports that execution has reached the traced location
3. Executes commands in a DO sequence if one was specified in the SET TRACE command (The SET TRACE command is discussed in Section 4.2.1.)
4. Resumes execution at the current program counter

These eventpoints remain in effect until the debugging session ends or until they are canceled or replaced.

To set a breakpoint, issue the SET BREAK command. This command has the following form:

```
SET BREAK [/qualifier] [address] [DO(action)]  
          /AFTER:n  
          /CALLS  
          /RETURN
```

To set a tracepoint, issue the SET TRACE command. This command has the following format:

```
SET TRACE [/qualifier] [address] [DO(action)]  
          /AFTER:n  
          /CALLS  
          /RETURN
```

4.2.1 SET BREAK and SET TRACE Command Qualifiers

This section explains the qualifiers you can use with both the SET BREAK and the SET TRACE commands. The qualifiers have the same effect on both commands.

4.2.1.1 The /AFTER:n Qualifier

If you specify the /AFTER:n qualifier, the debugger takes action at the nth activation of the specified location. It then takes action at each succeeding activation of the location. For example, if you specify a value of 3 for n, the breakpoint or tracepoint is activated when the debugger encounters the location more than two times; that is, on the third encounter, fourth encounter, and so on. The highest valid value of n is 255.

A special case exists. The /AFTER:0 qualifier has the same effect as the /AFTER:1 qualifier, which activates the breakpoint or tracepoint the first time the debugger encounters a location. However, the /AFTER:0 qualifier cancels the program controller once it has been activated. Therefore, /AFTER:0 allows you to set a program controller that you want to use only on the first encounter of a program location.

4.2.1.2 The /CALLS Qualifier

The /CALLS qualifier sets a breakpoint or tracepoint in two places for all commands that transfer control to a routine:

- After the calling instruction, but before the first instruction in a routine
- After the last instruction in a routine, but before the first instruction following a routine call

In other words, if you use the /CALLS qualifier to set a program controller, it is set at all JSR and RTS instructions, including those for system routines.

If you specify /CALLS, you cannot specify another qualifier in that command.

4.2.1.3 The /RETURN Qualifier

The /RETURN qualifier sets a breakpoint or tracepoint immediately after the last instruction in a calling routine, but before the first instruction following a routine call, that is, at an RTS command. You must specify the routine return you want to break or trace by using the address parameter explained in Section 4.2.2.1.

4.2.2 SET BREAK and SET TRACE Command Parameters

This section explains the command parameters you use with SET BREAK and SET TRACE. The effect of the parameters is the same for both commands.

4.2.2.1 The Address Parameter

The address parameter specifies the instruction address where you want to set a program controller. The specification may be in the form of a simple address or an address expression (these are explained in the *PDP-11 Symbolic Debugger User's Guide*, Section 3.2.1). You must specify this parameter if you do not use the /CALLS qualifier.

4.2.2.2 The DO Parameter

The DO parameter causes the debugger to execute one or more debugger commands when a breakpoint or tracepoint is activated. The action may be a single command, a list of commands separated by semicolons, or an indirect command file. The debugger executes DO action commands in the order in which they appear, but it does not check the syntax of these commands before they are executed. The number of levels to which you can nest indirect command files is limited only by the amount of dynamic storage currently available.

4.2.3 Commands Related to SET BREAK and SET TRACE

Four commands (SHOW BREAK, CANCEL BREAK, DISABLE BREAK, and ENABLE BREAK) are related to the SET BREAK command. Four other commands (SHOW TRACE, CANCEL TRACE, DISABLE TRACE, and ENABLE TRACE) are related to the SET TRACE command. This section describes the use of these commands.

To see which program controllers are in effect, issue either the SHOW BREAK or the SHOW TRACE command. The debugger responds to these commands with a message for either each breakpoint or each tracepoint that is set.

Once set, a program controller remains active for the duration of the debugging session unless you use the CANCEL BREAK or CANCEL TRACE command to cancel it or set another breakpoint or tracepoint at that program location. If you set a program controller in a location where one already exists, the second program controller set replaces the one set first.

The format of the CANCEL BREAK command is:

```
CANCEL BREAK[/qualifier][address]
           /ALL
           /CALLS
           /RETURN
```

The format of the CANCEL TRACE command is:

```
CANCEL TRACE[/qualifier][address]
           /ALL
           /CALLS
           /RETURN
```

The /ALL qualifier cancels either all breakpoints or all tracepoints currently set in a program. The /CALLS qualifier cancels either all the breakpoints or all the tracepoints at JSR and RTS instructions. The /RETURN qualifier cancels the program controller that is set at the RTS instruction of a routine. You must use the address parameter to specify which routine contains the program controller.

To prevent breakpoints from being activated, issue the DISABLE BREAK command; to prevent tracepoints from being activated, issue the DISABLE TRACE command. DISABLE commands do not cancel program controllers; they prevent program controllers from being activated until you enable them.

To enable program controllers, use the ENABLE BREAK or the ENABLE TRACE command. You do not have to respecify breakpoints or tracepoints when you use these commands.

Starting the Program

When you are ready to execute your COBOL-81 program, use either the STEP command or the GO command. This chapter explains how to use these commands.

5.1 Executing a Specified Number of Commands

To execute a specified number of commands in your program, use the STEP command. The STEP command causes the debugger to execute a single line or instruction, or a group of lines or instructions.

When you issue a STEP command, the debugger continues executing your program until one of the following occurs:

- A STEP sequence is complete.
- A breakpoint occurs.
- An error is detected in your program.
- Your program completes execution.
- You issue a control character command, such as CTRL/C.

A step sequence is considered complete only when the specified number of lines or instructions has been executed, regardless of intervening events.

The format of the STEP command is as follows:

```
STEP[/qualifier] [step-count]
    /INTO
    /OVER
    /INSTRUCTION
    /LINE
```

5.1.1 STEP Command Qualifiers

The `/INTO` and `/OVER` qualifiers control how the debugger treats called routines in your program. The `/INTO` qualifier specifies that the debugger step through the called routine. However, the `/OVER` qualifier specifies that the debugger stop stepping at a routine call, execute the called routine, and resume stepping when control is returned to the calling routine. Note that called routines can be either a routine you wrote or a system routine. Lines in called routines are not counted to satisfy a step-count when the `/OVER` qualifier is in effect.

The `/LINE` and `/INSTRUCTION` qualifiers determine what the debugger counts to satisfy a step-count. The `/LINE` qualifier specifies that the debugger count the execution of one line of your source program as a step. However, the `/INSTRUCTION` qualifier specifies that the debugger count each instruction in the PDP-11 machine code as a step. Therefore, if a line in your program translates to more than one PDP-11 machine code instruction, a single `STEP/INSTRUCTION` command does not execute that entire source program line.

Using these qualifiers with the `STEP` command overrides the default step conditions or conditions specified with the `SET STEP` command.

5.1.2 STEP Command Parameter

The step-count parameter specifies the number of source code lines or PDP-11 instructions (depending on how the step conditions are configured) you want the debugger to execute. A step-count must be given as a decimal integer.

Note that only executable lines, not comments or blank lines, are counted to satisfy a step-count.

5.2 Changing the Default Step Conditions

If you issue the STEP command without qualifiers when you start up the debugger, the debugger executes your program according to its default step conditions. By default, the debugger steps by line and counts only lines in the main routine to satisfy a step count (that is, /LINE/OVER).

Use the SET STEP command to change the default debugger step conditions. Once you change these conditions, the debugger executes the STEP command according to the conditions you set if you issue it without qualifiers.

The SET STEP command has the following format:

```
SET STEP parameter[,parameter]
      INTO
      OVER
      INSTRUCTION
      LINE
```

5.2.1 SET STEP Command Parameters

The SET STEP parameters have the same effect as that of the qualifiers to the STEP command; that is, the INTO and OVER parameters control whether the debugger steps into a called routine or suspends stepping to execute the called routine. The INSTRUCTION and LINE parameters control what the debugger counts to satisfy a step-count. INSTRUCTION tells the debugger to count all PDP-11 instructions, but LINE tells the debugger to count only source code lines.

5.2.2 The SHOW STEP and CANCEL STEP Commands

To display the current step conditions, issue the command

```
DBG>SHOW STEP
```

To restore step conditions to the debugger's default, issue the command

```
DBG>CANCEL STEP
```

This command returns the step conditions to their default of LINE and INSTRUCTION.

5.3 Executing an Undetermined Number of Commands

If you want to execute an undetermined number of commands in your program, use the GO command. The GO command instructs the debugger to execute your program until one of the following occurs:

- Your program terminates.
- A breakpoint is encountered.
- A pending STEP sequence is completed.
- An error is detected in your program.
- You issue a control character command, such as CTRL/C.

When you issue the GO command at debugger start-up, your program begins to execute as if you had built it without debugger support.

The GO command has the following format:

GO [address]

The address parameter allows you to specify an address at which to start program execution. It can be any legal simple address or address expression (see the *PDP-11 Symbolic Debugger User's Guide*, Section 3.2.1).

Manipulating Data

This chapter describes how to manipulate and alter data in your program using the EVALUATE, EXAMINE, and DEPOSIT commands. It also includes information on the concepts you must understand before using these commands.

6.1 Data Types in the Debugger

The debugger associates data types with literals, program symbols, and memory addresses. The data types of program symbols and memory addresses are assigned by the compiler. The data types of literals depend on the format of the literal. Following is a list of the literal data types the debugger supports and an explanation of how these data types are associated with literals:

- The data type integer is associated with literals that do not contain a decimal point.
- Literals that contain decimal points are associated with the data type real.
- The quoted string data type is associated with strings that are enclosed in quotation marks.

If a program symbol or memory address is not assigned a data type by the compiler, the debugger uses a default data type to interpret the location.

The default data type for COBOL-81 is ASCII. You can change the default data type with the SET TYPE command. This command has the format:

```
SET TYPE datatype
      BYTE
      INSTRUCTION
      PACKED
      WORD
```

Because the compiler normally assigns data types to your program locations, most users need not be concerned about the default debugger data type. Ordinarily, the debugger is unlikely to use its default data type to interpret any of your program locations.

To determine which default data type is currently in effect, you can issue the SHOW TYPE command. The debugger responds with a message showing the default.

6.2 Debugger Modes

The PDP-11 Symbolic Debugger supports radix modes and symbol modes. These modes work together to control the form in which the debugger interprets and displays information. The default debugger modes for COBOL-81 are a decimal radix mode and a symbolic symbol mode. When these modes are in effect, the debugger interprets and displays numbers in the numeric base 10. It also displays the symbol assigned to a virtual address instead of the address itself.

If you do not want to use these default modes, specify the mode you want to use by issuing the SET MODE command or by specifying a mode qualifier with the EXAMINE, EVALUATE, or DEPOSIT commands. The SET MODE command has the format

```
SET MODE mode [,mode]
      BINARY
      DECIMAL
      HEXADECIMAL
      OCTAL
      [NO]SYMBOL
```

The EXAMINE, EVALUATE, and DEPOSIT commands are explained later in this chapter.

The radix modes BINARY, DECIMAL, HEXADECIMAL, and OCTAL determine how integers in addresses and value-expressions are interpreted and displayed. For example, the address 1010 can refer to four different locations, depending on which radix mode is in effect when a command containing that address is issued.

[NO]SYMBOL determines whether symbols, such as variables names in your program, are displayed symbolically or by their numeric equivalents. It also determines how the processor status word (%PS) and floating point status word (%FS) are displayed. The default is SYMBOL. Note that [NO]SYMBOL only affects the debugger display because you can always enter data in either symbolic or numeric form.

To cancel modes established by the SET MODE command, issue the following command:

```
DBG>CANCEL MODE
```

This command returns the mode settings to their defaults.

To have the current modes displayed, issue the following command:

```
DBG>SHOW MODE
```

6.3 Determining the Virtual Address of Symbols

Before you examine and modify memory, you should obtain information to tell you what virtual addresses are associated with your program symbols. You can determine this association using the EVALUATE command. By using certain qualifiers with the EVALUATE command you can also determine the addresses of memory locations. This command has the following format:

```
EVALUATE[/qualifier] expression
        /ADDRESS      address
        /BINARY       value-expression
        /DECIMAL
        /HEXADECIMAL
        /OCTAL
```

6.3.1 EVALUATE Command Qualifiers

The /BINARY, /DECIMAL, /OCTAL, and /HEXADECIMAL qualifiers specify radix modes. If you specify a radix mode qualifier, integers in the expression parameter are interpreted in the specified radix and values are displayed in that radix.

6.3.2 EVALUATE Command Parameters

The expression parameter can be either an address or a value expression. If you want the debugger to determine the value of the expression using the address of the specified location, you must specify the /ADDRESS qualifier. If you do not use the /ADDRESS qualifier, the value of the expression is determined using the contents of the specified location. Note that you can evaluate only an expression that contains values that are resident.

If you issue the EVALUATE command with a simple address (one without operators) and without the /ADDRESS qualifier, the debugger displays the contents of the specified memory location.

6.4 Value Expressions

Value expressions may be specified with the EVALUATE and DEPOSIT commands. If a value in the expression refers to a memory location, the debugger performs the specified operations on the contents of the memory location, as opposed to the address of the location.

The following legal operators and delimiters in value expressions are listed in the order in which they are interpreted by the system:

1. Parentheses
2. Unary minus
3. Multiplication and division
4. Plus and minus

Quoted strings cannot be combined with debugger operators to form a value expression.

6.5 Displaying Memory Locations

The EXAMINE command lets you look at the contents of any virtual address or any resident memory location described by a debugger permanent symbol, a defined symbol, or a program symbol. The EXAMINE command has the following format:

```
EXAMINE [/qualifier] address
        /ASCII[:n]
        /BYTE
        /PACKED      (COBOL-81 only)
        /WORD
        /BINARY
        /DECIMAL
        /HEXADECIMAL
        /OCTAL
        /[NO]SYMBOL
```

The qualifiers you can use with the EXAMINE command are data type and mode qualifiers. These qualifiers control how the contents of the location you examine are displayed and how the address you specify is interpreted. They override the data type and mode specified with a SET TYPE or SET MODE command.

The address parameter specifies the location you want to display.

6.6 Referencing Variable Names Containing Hyphens

The debugger interprets hyphens in addresses as minus signs. Therefore, the reference to a variable defined as HOURS-1 in the following command line is interpreted as an arithmetic expression. (You are not permitted to perform arithmetic when language is set to COBOL.)

```
DBG>EXAMINE HOURS-1
%DEBUG-E-SYNTAXEXPR, syntax error in expression
```

To refer to a variable name that contains a hyphen, specify the %NAME keyword. The following example demonstrates the use of this keyword:

```
DBG>EXAMINE %NAME 'HOURS-1'
MAIN\HOURS-1: 40
```

This command causes the debugger to display the contents of the variable HOURS-1. Note that the variable specified with the %NAME keyword must be enclosed in apostrophes.

6.7 Altering Memory Locations

The DEPOSIT command changes the value of a location. You can deposit values into any resident program location. It has the following format:

```
DEPOSIT [/qualifier] address=value expression
        /ASCII[:n]
        /BYTE
        /PACKED      (COBOL-81 only)
        /WORD
        /BINARY
        /DECIMAL
        /HEXADECIMAL
        /OCTAL
```

6.7.1 DEPOSIT Command Qualifiers and Parameters

You can use mode and data-type qualifiers with the DEPOSIT command. The mode qualifiers determine what radix mode, or numerical base, the debugger uses to interpret the expressions you specify. The data-type qualifiers control how the debugger interprets the value you specify. These qualifiers let you deposit values in a data type other than the one associated with the memory location to which you are depositing, without altering the data type of that location. When you examine its contents later in your debugging session, your program treats the deposited value as if it were the location's data type, and the debugger still uses this data type to control its interpretation of the location. You can, however, instruct the debugger to change the location's data type to that of the value you deposited if you use a data type qualifier with the EXAMINE command.

The address parameter determines where a value is deposited. The value expression parameter gives the value you want deposited in that location.

6.7.2 Depositing ASCII Strings

To deposit an ASCII string, enclose the value expression in quotation marks or apostrophes. When the debugger encounters a string enclosed in quotation marks or apostrophes, it assumes that the string is of the data type ASCII. When the length of the string to be deposited is greater than the length associated with the address, the string is truncated from the right. However, when the length of the string is less than the length associated with the address, the debugger inserts ASCII blanks to the right of the last character in the string.

When you want to reference variable names of a character type other than ASCII, you use the /ASCII qualifier. If the string you are depositing is longer than two bytes, you must specify /ASCII:n, where n is the number of bytes in the string; otherwise, the debugger deposits only the first two bytes of your character string.

COBOL-81 Interactive Debugging Example

This chapter contains a sample debugging session for a COBOL-81 program. This debugging session demonstrates the most commonly used debugger commands.

7.1 The COBOL-81 Program

The COBOL-81 program example examined in this chapter calculates the square of the numbers 1 to 100 and the partial sum of their squares. It consists of two routines: the main routine, which is called MAIN; and a subroutine, which is called SQUARE.

```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID.          MAIN.
3
4      ENVIRONMENT DIVISION.
5
6      DATA DIVISION.
7      WORKING-STORAGE SECTION.
8      01  Messages.
9          02  Message1.
10             03  Line1 pic x(60) value "    Sums of Squares  (1:100)".
11             03  Empty-line pic x value " ".
12          02  Message2.
13             03  Line1 pic x(60) value "    I          I^2      Partial".
14             03  Line2 pic x(60) value "    Sum".
15          02  Message3 pic x(60) value " -----".
16
17      01  Partial pic 9(8).
18      01  I pic 9(4).
19      01  Isq pic 9(8).
20
21      01  Partial-disp pic z(7)9.
22      01  I-disp pic z(3)9.
23      01  Isq-disp pic z(7)9.
24
25      PROCEDURE DIVISION.
26      Par1.
27          Display Line1 of Message1.
28          Display Empty-line.
29          Display Line1 of Message2.
30          Display Line2 of Message2.
31
32          Move 0 to Partial.
33          Perform Par2
34              varying i from 1 by 1 until i > 100.
35
36          Display Message3.
37          Stop run.
38
39      Par2.
40          Call "MYSUB" using i, isq.
41          Add isq to Partial.
42          Move i to i-disp.
43          Move i to isq-disp.
44          Move partial to partial-disp.
45          Display i-disp, "    ", isq-disp, "    ", Partial-disp.
46

```



```

1      IDENTIFICATION DIVISION.
2      PROGRAM-ID.          MYSUB.
3
4      ENVIRONMENT DIVISION.
5
6      DATA DIVISION.
7      LINKAGE SECTION.
8      01 Num-in  pic 9(4).
9      01 Num-out pic 9(8).
10
11     PROCEDURE DIVISION USING Num-in, Num-out.
12     Par1.
13         Multiply Num-in by Num-in giving Num-in.
14         Exit program.

```

7.2 The Sample Debugging Session

The numbers displayed after executing the preceding program are incorrect. In each line of the display the variable I and the variable that contains the square of I are equal to each other. Also, the numbers displayed for Partial Sum are not equal to the sums of the squares of I.

The debugging session in this section locates the errors that caused the incorrect results in the preceding program. The comments in the examples briefly explain the commands. The callout number beside each comment corresponds to the numbered explanation that follows.

```

DBG>SET OUTPUT LOG                                !Ask for log file ①
DBG>SET BREAK %LINE 40                            !Break at routine call ②
DBG>GO                                              !Begin execution ③
%DEBUG-I-START, routine start at MAIN
%DEBUG-I-BREAKPOINT, breakpoint at MAIN\%LINE 40
DBG>EXAMINE I                                     !Check variable initialization ④
I: 1
DBG>GO                                              !Continue execution ⑤
%DEBUG-I-START, routine start a MAIN\%LINE 40
    Sums of Squares    (1:100)
        I          I2          Partial
                        Sum
-----
        1          1          0
%DEBUG-I-BREAKPOINT, breakpoint at MAIN\%LINE 40
DBG>GO                                              !Execute loop ⑥
%DEBUG-I-START, routine start at MAIN\%LINE 40
        2          2          0
%DEBUG-I-BREAKPOINT, breakpoint at MAIN\%LINE 40
DBG>EXAMINE ISQ                                   !Check squaring of I ⑦
ISQ: 0
DBG>SET TRACE %LINE 41 DO (DEPOSIT ISQ=9) !Set to test Partial Sum ⑧
DBG>GO                                              ⑨
%DEBUG-I-START, routine start at MAIN\%LINE 40
%DEBUG-I-TRACEPOINT, tracepoint at MAIN\%LINE 41
        3          3          9
%DEBUG-I-BREAKPOINT, breakpoint at MAIN\%LINE 40
DBG>GO                                              !Execute loop again for further testing ⑩
%DEBUG-I-START, routine start at MAIN\%LINE 40
%DEBUG-I-TRACEPOINT, tracepoint at MAIN\%LINE 41
        3          3          18
%DEBUG-I-BREAKPOINT, breakpoint at MAIN\%LINE 40
DBG>EXIT                                           !Exit debugger ⑪

```

- ① The SET OUTPUT command requests that a log file be maintained during the debugging session. A file called DEBUG.LOG is created and commands and debugger responses are written to this file during the session.
- ② A breakpoint is set at the call to the subroutine because the error causing incorrect output is thought to be in the subroutine.
- ③ The GO command begins program execution.
- ④ At this point, the initialization of the variable I is checked. Because it is correct, it is assumed that the Perform Par2 statement on lines 33 and 34 is correct.

- ⑤ The GO command executes the subroutine and the display commands following it once. The breakpoint at the routine call is activated again when it is encountered. The table printed between the START and BREAKPOINT messages is generated by the program, not by the debugger.
- ⑥ A second GO command is issued so that more of the table is displayed. It is determined by looking at the table that both ISQ and Partial Sum are being assigned incorrect values.
- ⑦ The debugger examines the variable ISQ to verify that it contains the value shown in the table. Because it does not, the debugger determines that the assignment of ISQ-DISP must contain an error. By checking the source code, the debugger recognizes the value of I as being assigned to ISQ-DISP. This error is marked for correction in the source code listing.

The contents of ISQ are also not correct, so there must be another error in the source code. The source code that computes the value passed back to ISQ is checked. The equation Multiply Num-in by Num-in giving Num-in is incorrect. The correct equation is Multiply Num-in by Num-in giving Num-out. This equation is marked for change in the source code listing.

- ⑧ Now that the reasons for the errors in the squaring of I have been determined, the only part of the program that remains to be debugged is the computation and display of Partial Sum. Because ISQ is not being computed correctly, a tracepoint is set at the command that initializes Partial Sum. The DO action specified in this command assigns the correct value to ISQ.
- ⑨ The GO command resumes execution of the program.
- ⑩ The value displayed for Partial Sum is correct during the first execution of the loop. However, to be sure that Partial Sum is being assigned the correct value, the loop is executed again with the GO command.
- ⑪ The table shows that Partial Sum contains a value of 18, which is correct because 9 was assigned to ISQ again by the SET TRACE command. Because all errors have been found, the EXIT command is issued.

INDEX

A

/AFTER:0 qualifier • 4-3
/AFTER:n qualifier • 4-3
Ambiguous references • 3-3
ASCII strings • 6-7

B

Breakpoint
 disabling • 4-6
 duration of • 4-2, 4-5
 effect of • 4-2
 enabling • 4-6
 setting • 4-2

C

Call-count • 4-1
/CALLS qualifier • 4-4
CANCEL BREAK command • 4-5
CANCEL MODE command • 6-3
CANCEL OUTPUT command • 2-2
CANCEL SCOPE command • 3-6
CANCEL STEP command • 5-3
CANCEL TRACE command • 4-5
COBOL-81 debugger, replacing bundled • 1-3
Command file
 See Indirect command file
Current location • 3-1
Current value • 3-1

D

Data type
 debugger • 6-1

Data type (cont'd.)
 default • 6-1
 with literals • 6-1
 with program symbols • 6-1
DCL commands
 invoking nonoverlaid kernel • 1-3
 invoking overlaid kernel • 1-4, 1-6
DEBUG command • 1-3, 1-4, 1-6
Debugger
 exiting • 1-6
 startup • 1-1
Debugger kernel
 nonoverlaid • 1-1
 overlaid • 1-1
Default debugger data type • 6-1
Default language
 effect of • 2-1
 setting • 2-1
Default output • 2-1
 changing • 2-1
DEFINE command • 3-3
Defined symbols • 3-3
DEPOSIT command • 6-5
 parameters • 6-6
 qualifiers • 6-6
DISABLE BREAK command • 4-6
Displaying memory • 6-4
DO action • 4-5

E

ENABLE BREAK command • 4-6
ENABLE TRACE command • 4-6
EVALUATE command • 6-3, 6-4
EXAMINE command • 6-4, 6-5

EXIT command • 1-6

F

File

See Indirect command file

See Log file

See ODL file

See STB file

@filespec command • 2-3

Floating-point status word • 3-1

G

GO command • 5-4

I

Indirect command file • 2-3

Invocation commands

nonoverlaid kernel • 1-2, 1-3

overlaid kernel • 1-4, 1-6

L

Line number • 3-1

Literal

with data types • 6-1

Log file • 2-2

default name • 2-2

example • 2-3

Logical predecessor • 3-1

Logical successor • 3-1

M

MCR commands

invoking nonoverlaid kernel • 1-2

invoking overlaid kernel • 1-4, 1-6

Memory

displaying • 6-4

Mode

canceling • 6-3

definition • 6-2

displaying • 6-3

radix • 6-2

symbol • 6-2

N

Nonoverlaid kernel • 1-1

invoking • 1-1, 1-2, 1-3

O

ODL file

creating • 1-5

example • 1-5

Operator

in value expressions • 6-4

Overlaid configuration

invoking • 1-4

Overlaid kernel • 1-1

invoking • 1-3, 1-4, 1-6

P

Pathname • 3-5

Permanent symbol • 3-1

Processor status word • 3-1

Program symbol • 3-2

and data types • 6-1

R

Registers • 3-1

/RETURN qualifier • 4-4

RSTS commands

invoking nonoverlaid kernel • 1-3

invoking overlaid kernel • 1-4, 1-6

S

Scope • 3-4

default • 3-4

of variables • 3-4

PC • 3-4

segment-list • 3-5

specifying • 3-5

Scope prefix • 3-5

Segment

overlay • 3-1

Segment-list

in pathname • 3-5

SET BREAK command • 4-3 to 4-5

- SET LANGUAGE command • 2-1
- SET LOG command • 2-2
- SET MODE command • 6-2
- SET OUTPUT command • 2-2
- SET SCOPE command • 3-5
- SET STEP command • 5-3
- SET TRACE command • 4-3 to 4-5
- SET TYPE command • 6-2
- SHOW BREAK command • 4-5
- SHOW CALLS command • 4-1
 - display • 4-1
- SHOW MODE command • 6-3
- SHOW OUTPUT command • 2-2
- SHOW SCOPE command • 3-5
- SHOW STEP command • 5-3
- SHOW TRACE command • 4-5
- SHOW TYPE command • 6-2
- STB file • 3-2
- STEP command • 5-1
- Step condition
 - changing • 5-2
 - default • 5-2
 - displaying • 5-3
 - restoring • 5-3
- STEP parameter • 5-2
- STEP sequence • 5-1
- Symbols • 3-1
 - creating • 3-3
 - defined • 3-3
 - in the debugger • 3-1
 - making unique • 3-3
 - pathname element routine replacement • 3-5
 - permanent • 3-1
 - program • 3-2
 - qualifying • 3-3

T

Tracepoint

- duration of • 4-2, 4-5
- effect of • 4-2
- enabling • 4-6
- setting • 4-2

V

Value expression • 6-4

Value expression (cont'd.)

- valid operators • 6-4
- Variable, hyphenated • 6-5

READER'S COMMENTS

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

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

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

Please indicate the type of user/reader that you most nearly represent:

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

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

Do Not Tear - Fold Here and Tape

digital



No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698



Do Not Tear - Fold Here

Cut Along Dotted Line