PDP–11 Symbolic Debugger User's Guide

Order Number: AA-FA60A-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

First Printing, December 1985

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/RSX	UNIBUS
DECmate	PDP	VAX
DECnet	P/OS	VAXcluster
DECUS	Professional	VMS
DECwriter	QBUS	VT
FMS	Rainbow	Work Processor
MASSBUS	RSTS	digital
MicroPDP-11	RSX	

ZK2955

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

Contents

PREFACE ix

CHAPTER 1 INTRODUCTION TO THE DEBUGGER 1-1

1.1 DEBUGGER FEATURES 1-1

1.2 ABBREVIATING KEYWORDS 1-2

1.3 STRATEGIES FOR USING THE DEBUGGER 1-2
1.3.1 Planning a Debugging Session 1-3
1.3.2 Debugging Optimized Code 1-3
1.3.3 Receiving Online Help 1-4

1.4 INVOKING THE DEBUGGER 1-4
1.4.1 How to Invoke the Debugger for Your MACRO–11 Task 1-5
1.4.1.1 Using the Overlaid Configuration • 1-5
1.4.1.2 Using the Nonoverlaid Configuration • 1-7
1.4.2 Invoking the Debugger with Instruction and Data Space (I- and D-space) Support 1-8
1.4.3 Exiting the Debugger 1-8

CHAPTER 2 CONFIGURING 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 SET OUTPUT Command Example 2-3
2.2.3 The SHOW OUTPUT and CANCEL OUTPUT Commands 2-3

2.3 USING LOG FILES 2-3
2.3.1 SET LOG Command Parameter 2-4
2.3.2 Example of a Log File 2-4
2.3.3 The SHOW LOG Command 2-5

2.4 USING INDIRECT COMMAND FILES 2-5
2.4.1 @ Command Parameter 2-6
2.4.2 @ Command Example 2-6

CHAPTER 3 DEFINING SYMBOLS 3-1

3.1 USING PERMANENT SYMBOLS 3-1

3.2 DEFINED SYMBOLS 3-3
3.2.1 Define Parameters 3-3
3.2.2 DEFINE Command Examples 3-5
3.2.3 The UNDEFINE Command 3-5

3.3 PROGRAM SYMBOLS 3-6
3.3.1 SET STB Command Parameter 3-7
3.3.2 The SHOW STB Command 3-7

3.4 REFERENCING PROGRAM SYMBOLS 3-7
3.4.1 Simple Pathnames 3-8
3.4.2 Pathname for Overlaid Programs 3-9
3.4.3 The SET SCOPE Command 3-10
3.4.3.1 SET SCOPE Parameter • 3-11
3.4.3.2 SET SCOPE Examples • 3-11
3.4.4 The SHOW SCOPE and CANCEL SCOPE Commands 3-12

CHAPTER 4 CONTROLLING PROGRAM EXECUTION 4-1

4.1 DISPLAYING INFORMATION ON ACTIVE ROUTINE CALLS 4-1
4.1.1 SHOW CALLS Command Parameter 4-2
4.1.2 SHOW CALLS Command Example 4-2

4.2 USING BREAKPOINTS AND TRACEPOINTS 4-3
4.2.1 SET BREAK and SET TRACE Command Qualifiers 4-4
4.2.1.1 The /AFTER:n Qualifier • 4-5
4.2.1.2 The /CALLS Qualifier • 4-5
4.2.1.3 The /RETURN Qualifier • 4-5
4.2.2 SET BREAK and SET TRACE Command Parameters 4-6
4.2.2.1 The Address Parameter • 4-6
4.2.2.2 The WHEN Parameter • 4-6
4.2.2.3 The DO Parameter • 4-6
4.2.3 SET BREAK and SET TRACE Command Examples 4-7
4.2.4 Commands Related to SET BREAK and SET TRACE 4-7

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 5-3
5.1.3 STEP Command Examples 5-3

5.2 CHANGING THE DEFAULT STEP CONDITIONS 5-4
5.2.1 SET STEP Command Parameters 5-5
5.2.2 SHOW STEP and CANCEL STEP Command Examples 5-5

5.3 EXECUTING AN UNDETERMINED NUMBER OF COMMANDS 5-5
5.3.1 GO Command Parameter 5-6
5.3.2 GO Command Examples 5-6

CHAPTER 6 MANIPULATING MEMORY 6-1

6.1 DATA TYPES IN THE DEBUGGER 6-1

6.2 CHANGING THE DEFAULT DATA TYPE 6-2
6.2.1 SET TYPE Command Parameter 6-3
6.2.2 SHOW TYPE and CANCEL TYPE Command Examples 6-4

6.3 DEBUGGER MODES 6-4
6.3.1 SET MODE Command Parameter 6-5
6.3.2 The SHOW MODE and CANCEL MODE Commands 6-5

6.4 VALUE EXPRESSIONS 6-6

6.5 DETERMINING THE VIRTUAL ADDRESS OF SYMBOLS 6-6
6.5.1 EVALUATE Command Qualifiers 6-7
6.5.2 EVALUATE Command Parameters 6-7
6.5.3 EVALUATE Command Examples 6-7

6.6 DISPLAYING MEMORY LOCATIONS 6-9
6.6.1 EXAMINE Command Qualifiers 6-9
6.6.2 EXAMINE Command Parameter 6-10
6.6.3 EXAMINE Command Examples 6-11

6.7 ALTERING MEMORY LOCATIONS 6-12
6.7.1 DEPOSIT Command Qualifiers 6-12
6.7.2 DEPOSIT Command Parameters 6-13
6.7.3 Depositing ASCII Strings 6-13
6.7.4 Depositing Radix–50 Strings 6-14
6.7.5 DEPOSIT Command Examples 6-14

CHAPTER 7 ENDING THE DEBUGGING SESSION 7-1

7.1 PREPARING TO LEAVE THE DEBUGGER 7-1

7.2 EXITING THE DEBUGGER 7-2

7.3 USING CONTROL COMMANDS 7-2

APPENDIX A PDP–11 SYMBOLIC DEBUGGER ERROR MESSAGES A-1

INDEX

TABLES

3–1 Debugger Permanent Symbols 3-1

6–1 Data Types Recognized By The Debugger 6-3

Preface

Intended Audience

This manual is intended for programmers who have no experience with the PDP-11 Symbolic Debugger. However, experienced debugger users can refer to it for explanations of commands they use infrequently.

This manual assumes that the reader understands programming in one of the supported languages and knows 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 gives an overview of the features of the debugger and some guidelines on how to use the debugger effectively. It also tells you where to look for the commands to invoke the debugger.
- 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 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 examining and altering memory locations.
- Chapter 7 explains how to exit from the debugger.
- Appendix A lists and explains debugger informational and error messages.

Associated Documents

The following briefly explains the contents of each manual in the PDP-11 Symbolic Debugger documentation set.

- *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.

- *PDP–11 Symbolic Debugger COBOL-81 User's Guide.* This manual gives information to debugger users who program in COBOL-81.

NOTE

There are language-specific exceptions to the general use of the debugger. Thus, wherever information given in a language-specific debugger guide conflicts with information presented here, the language-specific guide take precedence.

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.

Chapter 1

Introduction to the Debugger

The PDP-11 Symbolic Debugger helps you find logical and programming errors in successfully compiled programs that do not execute correctly. For example, the debugger can help you determine why a program terminates abnormally, produces incorrect output, or goes into an infinite loop. The debugger assists you in isolating and correcting your program's errors by allowing you to stop and start program execution where you want and to display and alter the contents of memory locations.

1.1 Debugger Features

PDP-11 Symbolic Debugger characteristics make it a powerful and flexible tool for finding logical and programming errors. You can use the debugger to do the following:

- Issue debugger commands from your terminal and see their effects immediately.
- Refer to program locations by symbols, rather than virtual addresses, and receive debugger output in symbolic form.
- Use the debugger on code written in FORTRAN-77, COBOL-81, and MACRO-11. You can also switch languages during a debugging session.
- Set breakpoints and tracepoints in overlay segments that are not currently resident.
- Invoke online help to get information about debugger commands and related features.

- Debug your programs on seven DIGITAL operating systems. These operating systems are RSX-11M, RSX-11M-PLUS, Micro/RSX, RSTS/E, Micro/RSTS, P/OS using the Pro/Tool Kit or the Professional Host Tool Kit, and VAX/VMS. (Note that the Symbolic Debugger task is native to VAX/VMS. All further references to VAX/VMS in this manual assume that the PDP-11 debugger kernel, which is linked to the task to be debugged, is in the compatibility mode for VAX-11 RSX.)

NOTE

COBOL-81 is not supported on VAX/VMS.

1.2 Abbreviating Keywords

You can abbreviate any keyword in a debugger command to its shortest unique abbreviation. For example, you can abbreviate the command SET BREAK to SE BR because no command keywords other than SET and BREAK begin with SE and BR. Symbol names must always be fully spelled out, however.

1.3 Strategies for Using the Debugger

Three strategies help you use the debugger effectively:

- Plan your debugging sessions carefully.
- Avoid debugging optimized code.
- Refer to the online HELP facility for information if you encounter problems during a debugging session.

The following sections discuss these strategies in more detail.

1.3.1 Planning a Debugging Session

To use the PDP–11 Symbolic Debugger efficiently, you should tailor the planning for your debugging session to the kind of abnormal behavior displayed by your program. Three common types of abnormal behavior are unexpected termination of the program, infinite looping within the program, and incorrect control transfers. This section gives suggestions for finding errors in programs that contain these types of abnormal behavior.

Unexpected termination of a program usually occurs shortly after the error that caused it. Use the debugger to examine the contents of key variables at the time of termination. If you note the approximate location of termination, use the debugger to suspend program execution before the termination point. Then execute the program in small, predetermined steps and examine the contents of variables as you proceed to find the instruction or instructions that produce the incorrect results.

If a program becomes caught in an infinite loop or if a pointer leads to an incorrect program location, stop the program before the loop or pointer and advance it in predetermined steps. By examining key locations as you proceed, you isolate unexpected output and, therefore, the error that caused it.

Once you locate your programming errors, you can make corrections in your source program; then you may compile (or assemble), link, and execute the corrected version.

1.3.2 Debugging Optimized Code

Debugger functions remain available when you debug optimized code; however, the results of debugger operations may be unpredictable. For example, a variable location in optimized code, particularly one within a loop, may not have a value in it; instead, the value may be in any one of several registers. Therefore, if you display the contents of such a variable, the value displayed is incorrect.

In addition, in an optimized program, lines do not necessarily correspond to instructions as in unoptimized code. With optimized code, the compiler rearranges the generated instructions to minimize the number of temporary values required. Thus, the instructions generated and subsequently executed may not correspond to the source code, and stepping through a program by line or by instruction may yield unexpected results.

To avoid these difficulties, debug only unoptimized programs.

1.3.3 Receiving Online Help

If you encounter problems during a debugging session, you can use the online help facility to get information about debugger commands and their format, qualifiers, and parameters. The help facility also provides information about debugger modes, data types, valid variable references, and termination procedures.

To use the help facility, issue the HELP command in the following format:

```
HELP [help-topic [subtopic]]
```

The help-topic parameter and the subtopic parameter specify the subject or command for which you want help. If you type HELP without parameters, you receive a list of all the topics on which information is available.

1.4 Invoking the Debugger

The way you invoke the debugger depends on the programming language and operating system you use. See one of the following manuals for information on invoking the debugger for use with a high-level programming language:

- *PDP–11 Symbolic Debugger FORTRAN–77 User's Guide*
- *PDP–11 Symbolic Debugger COBOL–81 User's Guide*

If you are invoking the debugger for use on a MACRO–11 program, see Section 1.4.1.

Note that the PDP–11 Symbolic Debugger only allows access to user mode space, not to supervisor mode or kernel mode (sometimes called executive mode). You can link to supervisor mode (I-CSSUP) routines without affecting Debugger use, except that you may not step through supervisor space (see Section 5.2), or set eventpoint in supervisor space (see Chapter 4).

1.4.1 How to Invoke the Debugger for Your MACRO-11 Task

To invoke debugger support, you can use either an overlaid or nonoverlaid debugger kernel in your task. An overlaid debugger kernel occupies less than 4000 bytes of user program space and can be included in your task by creating an overlay descriptor language (ODL) file that combines your program with the debugger. You then assemble, task-build, and run your program. You can use the overlaid debugger kernel unless your program is overlaid and you are loading your overlay segments manually. In this case you must include the nonoverlaid debugger kernel because the overlaid kernel uses autoloading, and you cannot mix the two loading methods in a single task. If you want to use the overlaid debugger kernel, refer to Section 1.4.1.1.

A nonoverlaid debugger kernel occupies about 5000 bytes of user space and can be included in your task by using certain switches when you assemble, link, and run your program. If you want to use the nonoverlaid kernel, refer to Section 1.4.1.2

1.4.1.1 Using the Overlaid Configuration

To use the overlaid debugger kernel, you must create an ODL file. The following example represents an ODL file that correctly includes the overlaid debugger in a user task. The source program to which this ODL file refers is called MYPROG.MAC.

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

As shown here, the ODL file you create for your task must include a factor (.FCTR) statement that concatenates your program with part of the debugger kernel ($DROOT). This factor statement must be declared in the ROOT statement as a co-tree with the rest of the debugger kernel ($DALL). Also, your ODL file must include PDPDBG.ODL, which is the debugger kernel ODL file, immediately before the END statement. Note that you can specify the elements in the FCTR statement in any order and that the kernel segment can be appended to an overlaid source program. For more information on ODL files and overlay structures, see the task builder manual for your operating system.

Once you create the ODL file, you assemble, link, and run your program to invoke the debugger. When you assemble your program, you should create a listing file and refer to it during the debugging session to follow program flow and to reference source code line numbers. You must also create a symbol table file for the debugger to have its full symbolic capability. The following example shows the MCR commands that you use to invoke the debugger.

FOR MCR USERS

```
>MAC myprog,myprog/-SP=myprog/EN:DBG
>TKB myprog,,myprog=myprog/MP
>RUN myprog
```

The task build (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 (the .MAP file) has been omitted.

The following example shows the DCL command you use to invoke the debugger.

FOR DCL USERS

```
$ MACRO/LIST myprog/ENABLE:DEBUG
$ LINK/SYMBOL_TABLE myprog/OVERLAY_DESCRIPTION
$ RUN myprog
```

NOTE

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

```
$ DEBUG myprog
```

VMS users must insert MCR in front of MAC in the compile command, and in front of TKB in the task build command, as in the following examples:

```
$ MCR MAC myprog,myprog/-SP=myprog/EN:DBG
$ MCR TKB myprog,,myprog=myprog/MP
```

1.4.1.2 Using the Nonoverlaid Configuration

When you want to invoke the debugger with the nonoverlaid kernel, you must assemble, link, and run your program. When you assemble your program, you should create a listing file and refer to it during the debugging session to follow program flow and to reference source code line numbers. You must also create a symbol table file for the debugger to have its full symbolic capability. If you are using MCR, type the following commands to invoke the debugger:

FOR MCR USERS

```
>MAC myprog,myprog/-SP=myprog/EN:DBG
>TKB myprog,,myprog=myprog,LB:[1,1]PDPDBG/DA
>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 (the .MAP file) has been omitted.

If you are using DCL, type the following commands to invoke the debugger:

FOR DCL USERS

```
$ MACRO/LIST myprog/ENABLE:DEBUG
$ LINK/DEBUG=LB:[1,1]PDPDBG/SYMBOL myprog
$ RUN myprog
```

NOTE

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

```
$ DEBUG myprog
```

Also, RSTS users should substitute LB: for LB:[1,1] in the examples given here.

1.4.2 Invoking the Debugger with Instruction and Data Space (I- and D-space) Support

RSX–11M–PLUS Version 2.1 or higher, RSTS/E Version 9.0 or higher, and Micro/RSX Version 3.0 or higher provide instruction and data space support. With this feature, you may be able to run significantly larger programs than is otherwise possible with a PDP–11.

To use the debugger with a task built in I- and D-space, take one of the following actions:

- MCR users should add /ID to the TKB command on the output file specification.
- DCL users should add /CODE:DATA_SPACE to the LINK command on the output file specification.

For more information on building tasks in I- and D-space, consult the *RSX–11M/M–PLUS Task Builder Manual* or the *RSTS/E Task Builder Reference Manual* as appropriate.

For information on linking the debugger with FORTRAN–77 tasks in I- and D-space, refer to *PDP–11 Symbolic Debugger FORTRAN-77 User's Guide.*

NOTE

I- and D-space support is not provided for COBOL–81.

1.4.3 Exiting the Debugger

To leave the debugger, type the following command in response to the debugger's prompt:

```
DBG>EXIT
```

Chapter 7 provides more information on exiting the debugger.

Chapter 2

Configuring Debugger Input and Output

The PDP-11 Symbolic Debugger can display and accept information in different formats. The format used primarily depends on the language in which your program is written. However, you have control over some parts of the debugger's input and output format. This chapter explains how to configure debugger input and output.

2.1 Setting the Default Language

When you invoke the debugger, it displays a message indicating the default programming language, that is, the programming language in which the debugger expects your program to be written. The default programming language controls key input and output configuration features. Therefore, if the debugger indicates a language other than the one you are using, you should change the default language with the SET LANGUAGE command.

How the SET LANGUAGE command affects debugger input and output depends on the programming language. When you are debugging a MACRO-11 program, the current language should be UNKNOWN. When the current language is UNKNOWN, the debugger expects input in word integer format and displays output in this format. If the current language is not UNKNOWN, issue the following command when you are debugging a MACRO-11 program:

```
DBG>SET LANGUAGE NONE
```

For an explanation of the effect of the SET LANGUAGE command when debugging a FORTRAN-77 or COBOL-81 program, consult *PDP-11 Symbolic Debugger FORTRAN-77 User's Guide* or *PDP-11 Symbolic Debugger COBOL-81 User's Guide* as appropriate.

You also use the SET LANGUAGE command when you debug a program written in more than one programming language. For example, if one routine is written in a language different from the other routines in your program, issue the SET LANGUAGE command when you enter the routine written in the new language. When you return to a routine written in the original programming language, issue the SET LANGUAGE command again.

2.2 Changing the Default Output

Some commands you issue to the debugger generate output. You can control how the debugger records and displays this output with the SET OUTPUT command. By issuing this command, you can change one or all aspects of the output configuration at any time during a debugging session.

The SET OUTPUT command has the following format:

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

2.2.1 SET OUTPUT Command Parameters

The [NO]LOG parameter controls debugger logging. LOG specifies that you want the debugger to record input and output in a log file. A log file is a file created by the debugger to record the commands you issue to it, and the responses it has to those commands. NOLOG specifies that you do not want a record of the debugging session. The default is NOLOG.

The [NO]TERMINAL parameter determines whether or not debugger output is displayed on your terminal. TERMINAL displays debugger output. NOTERMINAL specifies that debugger output not be sent to your terminal. The default is TERMINAL.

The [NO]VERIFY parameter controls the display of indirect commands to the debugger. VERIFY specifies that indirect commands be displayed on your terminal and/or recorded in your log file before they are executed. NOVERIFY specifies that the debugger not display indirect commands on your terminal or record them in your log file before they are executed. The default is NOVERIFY.

2.2.2 SET OUTPUT Command Example

Following is an example of using the SET OUTPUT command:

```
DBG>SET OUTPUT LOG,VERIFY
```

This example requests that the debugger create a log file and record the input and output from the debugging session in it. It also requests indirect command verification.

2.2.3 The SHOW OUTPUT and CANCEL OUTPUT Commands

To check your output configuration, issue the following command:

```
DBG>SHOW OUTPUT
%DEBUG-I-OUTPUT,output:verify,terminal,logging to "DB2:[303,52]DEBUG.LOG;1"
```

The debugger responds to the SHOW OUTPUT command with a message that shows how each of the output characteristics is set. It also gives the default device, the default directory, the name of the log file, and tells whether or not debugger input and output is being recorded in this file.

You can return to the default output characteristics (NOVERIFY, TERMINAL, NOLOG) by issuing the following command:

```
DBG>CANCEL OUTPUT
```

2.3 Using Log Files

As mentioned previously, a log file is a file created by the debugger to record the commands you issue to it and the responses it has to those commands. Output to a log file is identical to output to a terminal, with two exceptions:

- The DBG> prompt does not appear in a log file.
- Each line of debugger response in a log file is preceded by a comment delimiter (the exclamation point).

You can use a log file as an indirect command file. Section 2.4 tells you how to use indirect command files with the debugger.

At the beginning of each debugging session, the log file is called DEBUG.LOG. However, you can specify that the debugger write records to a file other than DEBUG.LOG with this command:

```
DBG>SET LOG filespec
```

2.3.1 SET LOG Command Parameter

The filespec parameter names the new file in which you want the debugger to record input and output. It has the following format:

```
   [node::device:[directory]]filename.ext.
```

Note that the brackets around "directory" do not indicate an optional element, but are required syntax. If you do not specify a file extension, the debugger uses the default file extension LOG; for example, MYPROG.LOG.

If the debugger is already writing to a log file when you issue the SET LOG command, the old file is closed and the new one opened. If you specify a file that already exists, a new version of that file is created if your operating system supports version numbers. If you specify an existing version of a file, the debugger opens the file you specify and appends the log of the debugging session to the end of the file. The debugger also appends the log of the debugging session to the end of an existing file on operating systems that do not support version numbers.

2.3.2 Example of a Log File

The following example shows part of a log file:

```
SET OUTPUT VERIFY
SHOW OUTPUT
!%DEBUG-I-OUTPUT, output: verify, terminal, logging to "DB2:[303,52]DEBUG.LOG;1"
SET LOG myprog
```

This log file shows that the output characteristic VERIFY is being set. This output configuration is then verified by the message following the SHOW OUTPUT command. The SET LOG command causes this file, which is called DEBUG.LOG, to be closed and a new file called MYPROG.LOG to be created. Any further input and output records are written to the file MYPROG.LOG, instead of DEBUG.LOG.

2.3.3 The SHOW LOG Command

To verify the name of your log file, issue the following command:

```
DBG>SHOW LOG
%DEBUG-I-LOGGING, logging to "DB2:[303,52]DEBUG.LOG;1"
```

In response to this command, the debugger displays the name of the log file including its device and directory. The output also specifies if records are being written in the log file.

2.4 Using Indirect Command Files

If a debugging session is interrupted unexpectedly, you may want to restart it with a partially created log file. You can execute a log file as an indirect command file to resume the debugging session where you left off.

However, indirect command files do not have to begin as log files. You can create an indirect command file using a text editor. The command file can then be used to execute a series of commands. For example, you can create an indirect command file that sets the output configuration of the debugger and execute it each time you enter the debugger.

To create an indirect command file, list debugger commands in the order you want them executed. You do not use any special syntax within an indirect command file. *Comments* are allowed in the file. They should be written to the right of debugger commands and preceded by an exclamation point. Section 2.4.2 contains an example indirect command file.

An indirect command file executes until an EXIT command or an end-of-file is reached even if it contains syntax errors. If the syntax of a command in an indirect command file is incorrect, the debugger issues an error message and proceeds with the next line in the command file.

Invoke an indirect command file with the following command:

```
DBG>@filespec
```

2.4.1 @ Command Parameter

You must use the full file specification if your indirect command file is on another machine or device, in another directory, or if it does not have the default file type CMD. A full file specification has the following format:

```
[node::device:[directory]]filename.ext.
```

The brackets around "directory" do not denote an optional element, but are required syntax.

You can also issue the @ command within another indirect command file. The number of levels to which you can nest indirect command files is limited only by the amount of dynamic storage currently available.

2.4.2 @ Command Example

The following is an example of the output generated by issuing the @ command. Note that the output characteristic VERIFY must be set for the debugger to generate this output.

The contents of an indirect command file called MYPROG.CMD are as follows:

```
SET OUTPUT LOG, VERIFY
SHOW OUTPUT
SET LOG MYPROG.LOG      !Name of log file
SHOW LOG
```

This indirect command file executes as follows:

```
DBG>@MYPROG
SHOW OUTPUT
%DEBUG-I-OUTPUT, output: verify, terminal, logging to "DB2:[303,52]DEBUG.LOG;1"
SET LOG MYPROG.LOG
SHOW LOG
%DEBUG-I-LOGGING, logging to "DB2:[303,52]MYPROG.LOG;1"
```

Chapter 3

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 you can define symbols. It also discusses the details of referencing program 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, defined symbols, and program symbols.

3.1 Using 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 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
%PS	Processor status word
%FS	Floating-point status word
%F0 - %F5	Floating-point registers

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

Symbol	Definition
%LINE nnn	Source code line number
%NAME name	Special symbol name
%SEGMENT name	Overlay segment name
\	Current value
.	Current location
^	Logical predecessor
RET	Logical successor

In the table, the debugger permanent symbols %R0 to %R5 refer to the processor registers that are for general use.

The %SP, %PC, %PS, and %FS debugger permanent symbols all refer to the registers that help the processor control program execution.

Symbols %F0 to %F5 refer to registers that are used for floating-point arithmetic.

The debugger permanent symbol %LINE nnn refers to a source code line number where nnn is a decimal integer that specifies the line number. The debugger always interprets the integer you specify with %LINE as decimal. (Note that this permanent symbol is meaningless when you are debugging a MACRO–11 program.)

The debugger permanent symbol %NAME allows MACRO–11 programmers to refer to symbols in their programs that contain periods. For example, to refer to the program symbol A.OR.B, you specify

```
%NAME 'A.OR.B'
```

Note that the program symbol must be enclosed in single quotation marks.

The %SEGMENT name symbol specifies an overlay segment where name is a string that specifies the segment to which you are referring.

The current value symbol (\) denotes the last value displayed by an EVALUATE command or denotes zero if no EVALUATE command has been issued. (The EVALUATE command is explained in Chapter 6.)

The current location symbol (.) represents either the last address used with an EXAMINE or DEPOSIT command or zero if neither of these commands have been issued. (The EXAMINE and DEPOSIT commands are explained in Chapter 6.)

The logical predecessor symbol (^) refers to the address immediately before the current location and is defined only for elements of arrays, memory locations that do not have an assigned data type, program instructions, and PDP-11 machine registers.

The logical successor symbol (RET) refers to the address immediately following the current location and is also only defined for array elements, untyped storage locations, instructions, and PDP-11 machine registers.

All debugger permanent symbols except current value, current location, logical predecessor, and logical successor must be preceded by the percent sign (%) when used in a debugger command line.

3.2 Defined Symbols

During a debugging session, you can create a new symbol or change an existing defined symbol by using the DEFINE command. Defined symbols are recognized only by the debugger. Although you can assign a new name to a location or symbol in your program, your program has no knowledge of the new name—it is known only to the debugger. These defined symbols remain in effect until you terminate the debugging session unless they are redefined or undefined.

The DEFINE command has the following format:

```
DEFINE symbol=address
```

3.2.1 Define Parameters

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

- It may be composed of only alphanumeric characters (the letters A through Z and the numbers 0 through 9) and dollar signs ($).
- It may not be more than 6 characters long.
- It may not begin with a number.

Note that uppercase and lowercase letters are not differentiated. Thus, the symbol name TECH and the symbol name Tech are treated as the same symbol name.

The parameter address identifies a portion of memory. Integers used in an address are interpreted in the current radix. For example, if the radix is set to octal, the integer 10 will be interpreted as a decimal 8. (See Section 6.3 for a further discussion of radixes.) An exception to this rule is source code line numbers. Line numbers are always interpreted as decimal integers. Addresses can be either a simple address or an address expression.

A simple address can be one of the following:

- An integer that denotes a program or memory location
- A previously defined symbol, as in a symbol created with the debugger command DEFINE
- A program symbol, such as a variable in your program
- A debugger permanent symbol (Table 3–1 lists and explains these symbols.)

The simple address can be combined with other simple addresses and operators to form an address expression. Debugger operators that can be used in address expressions are listed below in order of their precedence:

1. Parentheses
2. Unary minus or the negative symbol
3. Multiplication and division symbols
4. Addition and subtraction symbols

If two or more operators of equal precedence are used in the same expression, the order of evaluation is from left to right. When the debugger computes the value of an address expression, it performs the operation you specify on the value of the address of the specified location rather than on the value of the contents of the specified location. It also converts floating-point literals to integers.

3.2.2 DEFINE Command Examples

To define a symbol that refers to memory location 1064, issue the following command:

```
DBG>DEFINE SUM = 1064
```

Because this command assigns the symbolic name SUM to address 1064, you can now specify the defined symbol SUM to refer to memory location 1064.

To define a symbol that refers to a memory location 4 bytes after SUM, issue the following command:

```
DBG>DEFINE NEWSUM = SUM + 4
```

This command assigns the symbolic name NEWSUM to memory address 1068. Note that the debugger computes the address expression by adding the integer 4 and memory address of SUM, not the contents of memory at the location represented by SUM.

3.2.3 The UNDEFINE Command

You can cancel the definition of a symbol you have previously defined with the following command:

```
DBG>UNDEFINE symbol
```

The symbol parameter refers to the name of the defined symbol you want to undefine. This command makes the defined symbol named in the parameter invalid. Note that you can alter a defined symbol definition by issuing another DEFINE command for the symbol. The definition of a symbol is superseded if another DEFINE command for that defined symbol is issued.

3.3 Program Symbols

When you build your program, the task builder defines program symbols for you. These program symbols are defined in your source code's symbol table file (STB). In the STB, the program symbol names are associated with virtual addresses or program data and the current data type.

The STB file contains records for the following program symbols:

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

When you compile your program, the compiler generates PDP-11 machine code. Each machine instruction has a corresponding program counter (PC) value. Frequently, the compiler produces more than one PDP-11 instruction for each source code line, in which case the debugger interprets each source line as a *range* of PC values corresponding to the PDP-11 instructions generated for that line. (Note that MACRO-11 programs have only one instruction per line.) The information about the range of PC values that correspond to one of your source code lines is kept in the STB file.

NOTE

MACRO-11 does not generate the records required for the debugger to reference all program symbols. Local symbols, such as 10$, cannot be referenced.

You cannot display the contents of the STB file. However, you can create a map file for the STB file using TKB or determine if a given symbol is in the STB file by using the EXAMINE command. If you ask the debugger to examine a symbol that is not in the STB file, you receive an error message. Chapter 6 describes the EXAMINE command.

By default, the debugger looks for program symbol records in a file that has the same name as your source code file and the file extension STB. If the debugger is using the wrong STB file, you can specify the correct file with the following command:

```
DBG>SET STB filespec
```

3.3.1 SET STB Command Parameter

Filespec is the name of the STB file for your program. Occasionally, the debugger does not refer to the correct STB file because, for example, you specified an STB file name other than the default one in your TKB or LINK command or the STB file was moved after the TKB or LINK command was issued.

Filespec has the following format:

```
[node::device:[directory]]filename.ext
```

The brackets around "directory" do not indicate an optional item, but are required syntax. The default file extension for STB files is STB.

3.3.2 The SHOW STB Command

To verify that the debugger is using the correct STB file, instruct the debugger to display the name of the STB file by issuing the following command:

```
DBG>SHOW STB
%DEBUG-I-INISTBNAM, TKB-specified STB file is "DB2:[303,52]MYPROG.STB;1"
```

3.4 Referencing Program Symbols

When your program consists of more than one routine, the debugger searches in the routine you are currently debugging for any program symbols to which you refer. Therefore, if you refer to a program symbol that is not in the currently executing routine, you must specify that the debugger look in another routine for the symbol. Otherwise the debugger issues the following error message:

```
%DEBUG-E-NO-PATHNAME, unable to find SUBA\SUBA\VAR1 in STB file
```

The portion of your program in which a program symbol is known is called the scope of that symbol. For example, if a routine in your program called SUB1 contains the variable ARR(1), the scope of ARR(1) is the routine SUB1.

As mentioned previously, the debugger assumes that the scope of program symbols to which you refer is the routine you are currently executing. If you want to refer to a program symbol that is in another routine, you must specify its scope in one of the following three ways:

- Use a simple pathname.
- Use an extended pathname for an overlaid program.
- Use the SET SCOPE command.

These methods of specifying scope are discussed in the next sections.

3.4.1 Simple Pathnames

A pathname describes a program location. It consists of program location labels, such as routine names and source code line numbers, separated by the backslash character (\ ). Simple pathnames have the following format:

```
routine\routine[\%LINE nnn]
routine\routine[\symbol[(subscript-list)]]
```

The routine location label is the name of the routine in which the program symbol occurs. This label must be specified twice to signal to the debugger that the symbol to which you are referring is not in the current scope. If you do not specify the routine twice, the debugger interprets the routine name as a variable name and looks in the current scope for that variable.

The %LINE nnn label specifies a line number in the routine with nnn representing the decimal integer number of the line. Note that nnn refers only to compiler-generated line numbers.

The label symbol denotes a variable name or a symbol you defined previously for use in the routine. You use the subscript-list parameter when the symbol refers to an array, and you want to specify only certain elements of that array. Subscript lists can be expressions, but all integers they contain are interpreted in decimal radix, regardless of the default radix mode.

You must use an extended pathname for overlaid programs. Extended pathnames are discussed in Section 3.4.2.

NOTE

If you are debugging a MACRO–11 program, a pathname can contain only a module name and a symbol name, as follows:

```
module\symbol
```

The following are examples of valid pathnames for a nonoverlaid high-level language program that contains three routines: MAIN, SUB1, and SUB2. Remember that you use pathnames to specify locations that are not in the routine you are currently debugging.

For this example, the scope is the routine SUB1. If you want to refer to the twelfth element of an array called ARR contained in the routine called MAIN, give the following pathname:

```
MAIN\MAIN\ARR(12)
```

If you specify MAIN only once, the debugger looks in the current scope, SUB1, for a variable called MAIN. Because you are referring to ARR in the routine called MAIN, you must specify MAIN twice.

In the following example, the scope is still SUB1, but it refers to source code line 4 in SUB2.

```
SUB2\SUB2\%LINE 4
```

3.4.2 Pathname for Overlaid Programs

The pathname syntax for overlaid programs is an extended form of the simple pathname syntax. Because it is possible for a routine to appear at more than one place in an overlay tree, a method of uniquely identifying the routine is required. The extended pathname syntax contains a list of overlay segment names at the beginning of the pathname.

Valid pathname formats for an overlaid program are

```
segment-list\routine\routine[\%LINE nnn]
segment-list\routine\routine[\symbol[(subscript-list)]]
```

The segment-list specifies one or more segment names, in the following format:

```
%SEGMENT name[\%SEGMENT name...]
```

The keyword %SEGMENT must be specified when you reference one of two or more identical object files in your overlay structure that have the same name.

If you specify several segment names, specify them in order of segment branching, with the segment name nearest the program root specified first.

For an example of when to use the %SEGMENT keyword, consider the following ODL file:

```
       .ROOT MAIN-*(A,B)
MAIN:  .FCTR PROG.OBJ-LB:[1,1]PDPDBG.OBJ/DA
A:     .FCTR FILE1.OBJ-FILE2.OBJ
B:     .FCTR FILE3.OBJ-FILE2.OBJ
```

This ODL file contains three segments: segment PROG is labeled MAIN, segment FILE1 is labeled A, and segment FILE3 is labeled B. (Note that MAIN, A, and B are only labels and are not recorded in the STB file. Segment names are determined by the name of the first object file in the segment.) The segment FILE1 consists of two object files named FILE1.OBJ and FILE2.OBJ. The segment FILE3 consists of two object files named FILE3.OBJ and FILE2.OBJ. The contents of FILE2.OBJ are identical.

To reference a variable contained in the FILE2.OBJ, you must tell the debugger which segment contains the object file you want to reference. For example, the following pathname references a variable in FILE2.OBJ that is contained in segment FILE1.

```
%SEGMENT FILE1\MAIN\MAIN\VAR1
```

However, the following pathname references the same variable name in FILE2.OBJ that is contained in the segment FILE3.

```
%SEGMENT FILE3\MAIN\MAIN\VAR1
```

3.4.3 The SET SCOPE Command

The SET SCOPE command establishes the specified program unit as the one to be used for symbol interpretation. The scope established by the SET SCOPE command becomes the default for all symbols specified without a pathname. In other words, once you set the scope to a routine, the scope is no longer dynamic. The debugger looks in that routine for the symbols to which you refer without a pathname until the scope is set to another routine, canceled, or set to zero.

You issue the command as follows:

```
DBG>SET SCOPE pathname
```

3.4.3.1 SET SCOPE Parameter

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

The number 0 returns the scope to the debugger default, which is the currently active program unit. The scope is then dynamic and is always the routine you are currently debugging.

The backslash (\) specifies that symbols referenced without pathnames be interpreted as global symbols.

A scope prefix may be thought of as a truncated pathname. It describes a location in terms of its segment name (if any) and routine name. A scope prefix does not specify a particular line number, array reference, or symbol, as a pathname does. The format of a scope prefix is

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

The location label segment-list names the overlay segment that contains the routine to which you are referring. The label routine names the routine to which you are setting the scope.

3.4.3.2 SET SCOPE Examples

This section demonstrates several uses of the SET SCOPE command.

The first example sets the scope for an overlaid program.

```
DBG>SET SCOPE %SEGMENT ROOT\MAIN
```

This example uses a scope prefix with the SET SCOPE command to tell the debugger to interpret all symbols as lying within the main routine in the root segment.

You can also set the scope to overlay segments that you did not write, for example:

```
DBG>SET SCOPE %SEGMENT RMS\$GET
```

This command tells the debugger that symbols you refer to without a pathname are in the RMS routine $GET, which is in the segment named RMS.

Another use of SET SCOPE is to reset the scope to the currently active routine by issuing the following command:

```
DBG>SET SCOPE 0
```

This command makes the scope dynamic; the scope changes as you debug your program.

The backslash (\), when used with the SET SCOPE command, instructs the debugger to interpret symbols specified without pathnames as global symbols as in the following command line:

```
DBG>SET SCOPE \
```

You are most likely to set the scope to \ when you want to refer to symbols in a MACRO–11 program or symbols in high-level language routines compiled without the /DEBUG or /DB qualifier.

3.4.4 The SHOW SCOPE and CANCEL SCOPE Commands

SHOW SCOPE and CANCEL SCOPE are useful when you adjust the scope of the debugger.

To determine the current scope, use the following command:

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

The number 0 denotes that the scope is set to the routine currently executing. The routine name given in brackets tells you which routine is currently executing.

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

```
DBG>CANCEL SCOPE
```

As a result of the CANCEL SCOPE command, symbols without pathnames are interpreted as if they occurred in the routine that is currently executing. The CANCEL SCOPE command is equivalent in effect to the command SET SCOPE 0.

Chapter 4

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. FORTRAN–77 programs compiled with /TR:NAMES, /TR:BLOCKS, or /TR:ALL provide SHOW CALLS information. Since /TR:BLOCKS is the default switch, your program will supply traceback information unless you specify otherwise at compile time. (Note that /TR: switches are not used for MACRO–11 or COBOL–81. Also, SHOW CALLS is not a valid command when you are debugging a program written in MACRO–11.)

For each active 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 call path to the current routine (see Section 4.1.2).

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]
```

4.1.1 SHOW CALLS Command Parameter

The optional parameter call-count 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.

4.1.2 SHOW CALLS Command Example

The following example demonstrates the use of the SHOW CALLS command.

```
DBG>SET BREAK SUB2\%LINE 5
DBG>GO
%DEBUG-I-START, routine start at MAIN
%DEBUG-I-BREAKPOINT, breakpoint at SUB2\%LINE 5
DBG>SHOW CALLS

module name  routine name  line     rel PC        abs PC

  SUB2         SUB2           5     000036        060326
  SUB1         SUB1           9     000064        060224
  MAIN         MAIN          24     000262        002602
```

In this example, SUB2 is the routine that is currently being executed. SUB2 was called by SUB1, and SUB1 was called by MAIN.

The line numbers shown denote either the line after the call instruction (SUB1 and MAIN) or the line where execution of the program stopped (SUB2).

The relative PC value shows the number of bytes that have been executed in a routine. In other words, this value is the value of the PC relative to the beginning of a routine.

The absolute PC is the virtual address of the instruction immediately after the call or the instruction where execution of the program stopped.

NOTE

Traceback information for a routine call is not written to the traceback list immediately upon the execution of a call. Therefore, a SHOW CALLS command issued immediately after control is transferred to a routine can give incorrect information. To ensure that SHOW CALLS gives correct information, issue it only after the debugger has executed at least four PDP–11 machine instructions in the called routine.

4.2 Using Breakpoints and Tracepoints

Once you have identified the important points in your program, you can set either breakpoints or tracepoints at these points. To specify that you want program execution to stop at a particular location, use the SET BREAK command. Use the SET TRACE command if you want the debugger to report that it has reached a location without stopping program execution. The following material further describes the effects of breakpoints and tracepoints.

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

1. Suspends program execution immediately before the instruction at the specified location.
2. Tests the value of a WHEN condition if one was specified in the SET BREAK command. If the value is TRUE, the activation of the breakpoint continues; if the value is FALSE, your program resumes execution. (The WHEN parameter is not valid for COBOL–81.)
3. Displays the symbolic or virtual memory location at which execution has been suspended.
4. Executes commands in a DO sequence if one was specified in the SET BREAK command.
5. 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. Tests the value of a WHEN condition if one was specified in the SET TRACE command. If the value is TRUE, the activation of the breakpoint continues; if the value is FALSE, execution of your program resumes. (The WHEN parameter is not valid for COBOL–81.)
3. Reports that execution has reached the traced location.
4. Executes commands in a DO sequence if one was specified in the SET TRACE command.
5. Resumes execution at the current program counter.

These eventpoints remain in effect either until the debugging session ends or until they are canceled or replaced. See Section 4.2.4 for more information on the duration of eventpoints.

NOTE

> If you are using the overlaid debugger kernel, do not set a breakpoint in code that does overlay handling (for example, $AUTO, $MARKS, or $RDSEG). If a breakpoint is set in one of these system routines, the debugger enters an infinite loop when the breakpoint is activated.

The SET BREAK command has the following format:

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

The SET TRACE command has the following format:

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

4.2.1 SET BREAK and SET TRACE Command Qualifiers

The following sections explain 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 also takes action at each succeeding activation of the location. For example, if n equals 3, 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 /AFTER:1, 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, it must be the only qualifier.

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 the next section. You cannot use the /RETURN qualifier when you debug a MACRO-11 program because the MACRO assembler does not supply all the required information in the STB file.

4.2.2 SET BREAK and SET TRACE Command Parameters

The following sections explain the parameters for the commands 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 (I-space) address where you want a program controller set. It may be in the form of a simple address or an address expression. If you do not specify the /CALLS qualifier, you must specify this parameter.

4.2.2.2 The WHEN Parameter

The WHEN parameter allows you to control whether or not a program controller is activated by a condition specified by the expression. The debugger evaluates the expression and determines whether it is TRUE or FALSE. If the expression is TRUE, the debugger continues with activation of the program controller. If the value of the expression is FALSE, execution of the program resumes.

The expression you specify with the WHEN parameter can be any valid value expression as described in Section 6.4.

The WHEN parameter is not valid for all high-level programming languages. If the documentation for your high-level programming language does not contain a description of the WHEN parameter, you should not use it because the results of a command containing this parameter are unpredictable. The WHEN parameter is valid for use when you are debugging a MACRO-11 program.

4.2.2.3 The DO Parameter

The DO (action) parameter is used when you want the debugger to execute one or more debugger commands when a breakpoint or tracepoint is activated. Any valid debugger command, including another SET BREAK or SET TRACE command, can be specified in this parameter. 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 nesting of DO (action) commands is limited by the amount of dynamic storage available.

4.2.3 SET BREAK and SET TRACE Command Examples

This section demonstrates the use of the SET BREAK and SET TRACE commands.

To set a breakpoint at source code line 5 in the current scope, issue the following command:

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

This command will set a breakpoint at line 5 of the currently executing routine. When the debugger encounters this line in executing your source code, it stops program execution before it executes the instruction on line 5.

To set a tracepoint that is activated the fifth time an instruction is encountered, issue the following command:

```
DBG>SET TRACE/AFTER:5 %LINE 12 DO(SET OUTPUT LOG)
```

The preceding command sets a tracepoint at line 12 in the source code program. This tracepoint is activated immediately before the instruction on line 12 is executed for the fifth time. When this tracepoint is activated, the debugger begins writing records to a log file as specified by the DO sequence.

If you want to set an eventpoint in a routine that is not currently executing, specify a pathname, as in the following example:

```
DBG>SET TRACE SUB1\%LINE 5
```

This example shows how to set a breakpoint at line 5 in SUB1 when you are executing another routine.

4.2.4 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 of the following commands:

```
DBG>SHOW BREAK
%DEBUG-I-BREAKPONT, breakpoint at MAIN\%LINE 5
%DEBUG-I-BRK_ENABLED, the recognition of breakpoints is enabled
DBG>SHOW TRACE
%DEBUG-I-TRACEPOINT, tracepoint at MAIN\%LINE 12
%DEBUG-I-TRC_ENABLED, the recognition of tracepoints is enabled
```

The debugger responds to these commands with a message for each breakpoint or each tracepoint that is set.

Ordinarily, a program controller remains active for the duration of the debugging session. However, you can either cancel it with the CANCEL BREAK or CANCEL TRACE command 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 previous one.

The CANCEL BREAK command has the following format:

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

The CANCEL TRACE command has the following format:

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

For this command to operate correctly, you must specify either an address or the /ALL or /CALLS qualifier. (The /RETURN qualifier requires that an address be specified.)

The /ALL qualifier cancels all breakpoints or all tracepoints in a program. The /CALLS qualifier cancels all the breakpoints or all the tracepoints at JSR and RTS instructions. The /RETURN qualifier cancels the program controller set at the RTS instruction of a routine. You must specify which routine with the address parameter, which can be a simple address or an address expression.

To prevent breakpoints or tracepoints from being activated, issue either the DISABLE BREAK or the DISABLE TRACE command. These DISABLE commands do not cancel program controllers but they do prevent the 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 re-specify breakpoints or trace-points when you use these commands.

Chapter 5

Starting the Program

Once you have configured the debugger output and set breakpoints and tracepoints, you can execute your program. The PDP–11 Symbolic Debugger offers two methods of executing a program. The first method executes a number of instructions or lines depending on how you specify the command. The second method allows the program to run until one of several events is encountered.

5.1 Executing a Specified Number of Commands

To execute a specified number of lines 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.

NOTE

> The word "line" in this section refers to source code lines. The word "instruction" refers to compiler-generated PDP–11 machine instructions or MACRO–11 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.
- You issue a control character command, such as CTRL/C.
- Your program completes execution.

Although the debugger allows you to execute your program until it terminates, this use of the STEP command is not recommended. If you execute your program until its normal termination, the debugger background task may still be running after your program finishes. Therefore, you should always stop program execution before the end of your program and exit from the debugger in an orderly manner. (Exiting from the debugger is discussed in Chapter 7.)

A step sequence is complete only when the specified number of lines or instructions have been executed, regardless of intervening events. Therefore, if the debugger encounters a breakpoint while executing a step sequence, execution of the step sequence is merely suspended, not completed. Thus, any number of step sequences may be in effect at any one time; the sequences are executed independently and simultaneously. If multiple step sequences are completed at the same time, the debugger issues a message for each completed sequence.

The STEP command has the following format:

```
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 and that lines in called routines are not counted to satisfy a step count when the /OVER qualifier is in effect. The default condition is /OVER.

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, the command STEP/INSTRUCTION does not execute that entire source program line. The default condition is /LINE.

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. Step-count must be given as a decimal integer.

Note that neither comments nor blank lines are counted in satisfying a step count.

5.1.3 STEP Command Examples

The following example of using the STEP command demonstrates the effect of an intervening event on a step sequence. The scope in this example is a routine named MAIN.

```
DBG>SET BREAK %LINE 4
DBG>STEP/LINE 4
%DEBUG-I-START, routine start at MAIN
%DEBUG-I-BREAKPOINT, breakpoint at MAIN\%LINE 4
DBG>STEP/LINE
%DEBUG-I-START, routine start at MAIN\%LINE 4
%DEBUG-I-STEP, stepped to MAIN\%LINE 5
%DEBUG-I-STEP, stepped to MAIN\%LINE 5
```

The breakpoint is activated before the step sequence is complete, so the sequence is only completed when the second STEP command is issued. The "stepped to" message is issued twice because two step sequences (STEP/LINE 4 and STEP/LINE) are completed at the same time. Note that the instruction on line 5 of the routine called MAIN has not been executed at the end of this example because the debugger stops execution immediately before this instruction.

The next example shows how the /INSTRUCTION qualifier affects the STEP command. In this example, the scope is a routine named MAIN.

```
DBG>STEP/INSTRUCTION 4
%DEBUG-I-START, start at MAIN\%LINE 1
%DEBUG-I-STEP, stepped to MAIN\%LINE 1 + 16: MOV     %R1,-(%SP)
```

The STEP/INSTRUCTION 4 sequence starts at line 1 and ends after the fourth PDP–11 machine code instruction is executed. Because the four instructions require 16 bytes, the debugger steps to the beginning of the seventeenth byte of line 1.

The last example demonstrates how the /INTO and /OVER qualifiers affect the STEP command. In this example, the beginning scope is the routine MAIN. This routine calls two other routines, SUB1 and SUB2, in successive instructions.

```
DBG>STEP/INTO 3
%DEBUG-I-START, routine start at MAIN\%LINE 2
%DEBUG-I-STEP, stepped to SUB1\%LINE 2
DBG>STEP 5
%DEBUG-I-START, routine start at SUB1\%LINE 2
%DEBUG-I-STEP, stepped to MAIN\%LINE 3
DBG>STEP/OVER 2
%DEBUG-I-START, routine start at MAIN\%LINE 3
%DEBUG-I-STEP, stepped to MAIN\%LINE 5
```

In the first step sequence, the debugger counts the lines in the routine SUB1 in satisfying the step count for the STEP/INTO 3 and STEP 5 commands. In the second step sequence, the lines in SUB2 are not counted to satisfy the step count in STEP/OVER 2. Therefore, the step sequence is complete at the fifth line of the routine called MAIN.

5.2 Changing the Default Step Conditions

If you issue the STEP command without qualifiers at debugger start-up, 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.

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. In other words, the step conditions you set become the default step conditions for that debugging session. Note that you override the conditions you specify with the SET STEP command when you specify a parameter with the STEP command.

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 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 SHOW STEP and CANCEL STEP Command Examples

To display the current step conditions, issue the following command:

```
DBG>SHOW STEP
%DEBUG-I-STEPTYPE, step type: by line, over routine calls
```

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

```
DBG>CANCEL STEP
```

This command returns default conditions to step by line and step over routine calls.

5.3 Executing an Undetermined Number of Commands

If you want to execute an undetermined number of statements or instructions 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.
- Your program completes execution.

Although the debugger allows you to execute your program until it terminates, this use of the GO command is not recommended. If you execute your program until its normal termination, the debugger background task may still be running after your program finishes. Therefore, you should always stop program execution before the end of your program and exit from the debugger in an orderly manner. (Exiting from the debugger is discussed in Chapter 7.)

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

The GO command has the following format:

```
GO [address]
```

5.3.1 GO Command Parameter

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, as described in Section 3.2.1.

You should be cautious when using this parameter because the GO command does not alter register contents or the system stack; it simply transfers control to the designated location. Thus, if the program state at the specified address is not the same as the program state when you issue the GO command, the results are unpredictable. If you attempt to start the program at a location that is not currently resident, an error is reported. Also, if you specify an address that does not contain the beginning of an instruction, a run-time error may occur when the instruction including that byte is executed. (For I- and D-space tasks, the address parameter is an I-space address.)

5.3.2 GO Command Examples

When you use the GO command to resume program execution after suspension, execution resumes at the current program counter location, as in the following example:

```
%DEBUG-I-STEP, stepped to MAIN\%LINE 6
DBG>SET BREAK %LINE 8
DBG>GO
%DEBUG-I-START, routine start at MAIN\%LINE 6
%DEBUG-I-BREAKPOINT, breakpoint at MAIN\%LINE 8
```

Note that the breakpoint at line 8 ends the execution of the GO command. Another GO command must be issued for execution to continue.

This next example demonstrates the effect of an incomplete step sequence on the GO command.

```
DBG>SET BREAK %LINE 6
DBG>STEP 8
%DEBUG-I-START, routine start at MAIN
%DEBUG-I-BREAKPOINT, breakpoint at MAIN\%LINE 6
DBG>GO
%DEBUG-I-START, routine start at MAIN\%LINE 6
%DEBUG-I-STEP, stepped to MAIN\%LINE 8
```

When the step sequence is complete, execution of the GO command stops.

Chapter 6

Manipulating Memory

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

6.1 Data Types in the Debugger

Because different data types are interpreted and formatted in various ways, the debugger must associate a data type with all the values it accepts as input or displays as output. If the debugger cannot associate a data type with a value, it assigns the value a default data type.

A literal is any character string that is a constant but is not a symbol. The debugger supports three types of literals: integer, floating point, and quoted string. The debugger associates the data-type integer with a literal that does not contain a decimal point. If the literal contains a decimal point, the debugger associates the floating-point data type with the literal. A quoted-string data type is associated with a literal that is enclosed in either single or double quotation marks.

Because high-level programming language compilers assign data types to program symbols, it is unlikely that the debugger would have to use its default data type to interpret these symbols. Instead, the debugger associates the contents of high-level language program symbols with their compiler-generated data type. Also, the debugger assumes that input to a program symbol is given in the data type the compiler assigned to that symbol. Therefore, there is little need for you to be concerned with the default debugger data type when you are debugging a high-level language program. Your language documentation provides information on how the debugger treats the data types your programming language uses.

MACRO–11 programmers, however, must be concerned with the default debugger data type because symbols in MACRO–11 programs are untyped. Therefore, the debugger does use its default data type to interpret MACRO–11 program symbols. By default, the debugger uses the data-type word integer to interpret MACRO–11 program symbols.

The debugger associates a data type with a simple address or an address expression depending on three conditions:

- If you specify a type qualifier with a debugger command, an address in that command is interpreted as having the characteristics of the data type you specified.
- If you did not specify a type qualifier, but the address points to a program location that has a compiler-generated data type, that data type is used to interpret the address. (This is most likely to occur when you are debugging a program written in a high-level language.)
- If neither of the preceding conditions are met, the address is interpreted as having the default debugger data type, which depends on the programming language you are using. For MACRO–11, the default data type is word integer. Information on the default data type for high-level languages is found in the debugger documentation for that language. Note that you can change the default data type with the SET TYPE command, which is explained in Section 6.2.

6.2 Changing the Default Data Type

The default debugger data type is used when a literal, program symbol, or address cannot be associated with a data type in any other way.

If you do not want to use the default data type provided for your programming language, you can change the default type by issuing the SET TYPE command. The SET TYPE command has the following format:

```
SET TYPE datatype
         ASCII
         BYTE
         D_FLOAT
         FLOAT
         LONG
         INSTRUCTION
         PACKED       (COBOL-81 only)
         RAD50
         WORD
```

You can override the effect of the SET TYPE command by specifying a data type qualifier with the EXAMINE, DEPOSIT, and EVALUATE commands.

6.2.1 SET TYPE Command Parameter

The data type parameter determines what data type you want the debugger to use as the default type. Table 6–1 describes the data types the debugger recognizes.

Table 6–1: Data Types Recognized By The Debugger

Data Type	Definition
ASCII[:n]	ASCII of length n (where n = 2 by default)
BYTE	Byte integer
D_FLOAT	Double-precision floating point
FLOAT	Single-precision floating point
LONG	Longword integer
INSTRUCTION	Program instruction of variable length
PACKED	Packed decimal (valid only for COBOL–81)
RAD50	Radix–50
WORD	Word integer

Note that the default length for the datatype ASCII is 2 and n is always interpreted as a decimal integer. For more information on using ASCII, see the discussion in Section 6.6.1.

The data types that you can use depend on your programming language. The valid data types for MACRO–11 programmers are ASCII, BYTE, D_FLOAT, FLOAT, LONG, INSTRUCTION, RAD50, and WORD. See your high-level language documentation for a list of the data types you can use with a high-level programming language.

6.2.2 SHOW TYPE and CANCEL TYPE Command Examples

To determine which data type is in effect, you can issue the SHOW TYPE command as follows:

```
DBG>SHOW TYPE
%DEBUG-I-TYPE, type: word integer
```

To return the default data type to the one supplied by the debugger for you programming language, issue the following command:

```
DBG>CANCEL TYPE
```

6.3 Debugger Modes

The PDP-11 Symbolic Debugger supports two modes, radix and symbol. These modes control the form in which the debugger interprets and displays information. The default radix mode (the numerical base) is decimal, and the default symbol mode is symbolic.

If you do not want to use the default modes, you specify the modes you want to use by issuing the SET MODE command or by specifying a mode qualifier with the EXAMINE, EVALUATE, or DEPOSIT commands, which are explained later in this chapter.

The SET MODE command has the following format:

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

A radix mode specified as a qualifier to the DEPOSIT, EXAMINE, or EVALUATE command overrides the effect of the SET MODE command.

6.3.1 SET MODE Command Parameter

Mode determines how integers in address expressions and value expressions are interpreted and how memory locations are displayed. The BINARY, DECIMAL, HEXADECIMAL, and OCTAL modes change the numerical base the debugger uses to interpret and display information. For example, if the radix mode is set to BINARY, the number 1010 has a decimal value of 10. However, if the radix mode is set to HEXADECIMAL, this number has a decimal value of 4112. The radix mode does not control how data is stored in your program; it tells the debugger what numerical base to use to interpret your input and that it must translate output to this numerical base before it is displayed.

Note that the debugger always interprets *%LINE nnn* as decimal.

[NO]SYMBOL determines whether symbols, such as variable names in your program, are displayed symbolically or by their numeric equivalents. For example, if the symbol mode is set to NOSYMBOL, the virtual address of a memory location is displayed instead of the symbolic name that refers to that program location. [NO]SYMBOL also determines how the processor status word (%PS) and floating-point status word (%FS) are displayed (see Section 6.6.3). The default is SYMBOL. Note that [NO]SYMBOL affects only the debugger display because you can always enter data in either symbolic or numeric form.

6.3.2 The SHOW MODE and CANCEL MODE Commands

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

```
DBG>SHOW MODE
%DEBUG-I-MODES, modes: symbolic, decimal
```

This command causes the debugger to display a message describing the current modes.

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 of DECIMAL and SYMBOL.

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. These values have the data type associated with the memory location or the default debugger data type if no type is associated with the location.

Values are combined with operators and delimiters to form a value expression. The following legal operators and delimiters in value expressions are listed in order of precedence:

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.

NOTE

Value expressions are not valid for COBOL–81.

6.5 Determining the Virtual Address of Symbols

Before you examine and modify memory, you should understand how to determine which virtual addresses are associated with your program symbols. You can determine this association using the EVALUATE command. By adding or subtracting an offset you also can determine the addresses of higher and lower memory locations. You can only evaluate an expression that contains resident values.

The EVALUATE command has the following format:

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

6.5.1 EVALUATE Command Qualifiers

If you issue the EVALUATE command with no qualifiers and a simple expression (one without operators), the debugger displays the contents of the specified memory location.

You use the /ADDRESS qualifier when you want the debugger to determine the virtual address of a symbol or to use the address of a location you specify to determine the value of an expression parameter.

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.

6.5.2 EVALUATE Command Parameters

The expression parameter can either be 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 or, if the expression does not refer to a memory location, by performing the operations specified on the literals in the expression.

6.5.3 EVALUATE Command Examples

The following example demonstrates the use of the /ADDRESS qualifier.

```
DBG>DEFINE X = 15346
DBG>EVALUATE X
6083
DBG>EVALUATE/ADDRESS X
15346
```

In this example, the symbol X is defined to be memory location 15346. The response to the EVALUATE command that is issued without the /ADDRESS qualifier displays the contents of the memory location the symbol X references. The response to the EVALUATE command with the /ADDRESS qualifier displays the virtual address of the symbol X.

Another use of the EVALUATE command is to perform address arithmetic, as in the following example:

```
DBG>EVALUATE/ADDRESS (X + 2)/2
7674
```

This example demonstrates that the expression specified with the /ADDRESS qualifier is treated as an address, not as a value expression. Here the debugger adds the value of the memory location denoted by X (15346) and the integer 2. It then divides this sum by 2 to get the value 7674.

NOTE

> You cannot perform address arithmetic with the language set to COBOL. If you want to do this kind of arithmetic while debugging a COBOL program, first set the language to FORTRAN and then SET LANG COBOL when you finish the arithmetic exercise.

Finally, you can use the EVALUATE command to perform arithmetic on value expressions. Consider this command:

```
DBG>EVALUATE X + 10
6093
```

In this example, the debugger treats the expression as a value expression and adds 10 to the contents of location X to get the sum 6093. Note that you could also have specified that the debugger add 6083 and 10 as follows:

```
DBG>EVALUATE 6083 + 10
6093
```

The debugger treats values in EVALUATE expressions as literals unless one of them is a symbol or the /ADDRESS qualifier is specified.

6.6 Displaying Memory Locations

You can display the contents of memory by using the EXAMINE command. This command allows you to display the contents of any virtual address or any resident memory location described by a debugger permanent symbol, defined symbol, or program symbol.

The EXAMINE command lets you look at the contents of a memory location. It has the following format:

```
EXAMINE[/qualifier]  address
        /ASCII[:n]
        /BYTE
        /D_FLOAT
        /D_SPACE
        /FLOAT
        /LONG
        /INSTRUCTION
        /I_SPACE
        /RAD50
        /WORD
        /BINARY
        /DECIMAL
        /HEXADECIMAL
        /OCTAL
        /PACKED       (COBOL-81 only)
```

6.6.1 EXAMINE Command Qualifiers

The qualifiers you can use with the EXAMINE command are data-type and mode qualifiers. These qualifiers control how the contents of a memory location are displayed. For example, if you specify a data-type qualifier, the contents of the location you examine are displayed in the format of that data type. Likewise, specifying a mode qualifier causes the contents of the location to be displayed in that mode.

The data-type qualifiers are useful when you examine an untyped storage location because the debugger associates its default data type with untyped storage locations. If you examine a location that does not contain a value that is of the default data type, you must translate the result to the correct data type, so you can understand it. For example, if the default data type is ASCII and you examine a Radix–50 value, you must translate the ASCII string that is displayed to its Radix–50 equivalent before you know if the memory location contains the correct value. However, if you specify the /RAD50 qualifier when you examine the location, the value displayed is a Radix–50 value.

Because high-level language compilers associate data types with memory locations, you need not specify a data-type qualifier with the EXAMINE command to have values displayed in the correct data type when you are debugging a high-level language program. These qualifiers are used primarily for debugging MACRO-11 programs because they do not have data types associated with memory locations.

When examining a task in I- and D-space, if you do not use a qualifier, or use any qualifier except /INSTRUCTION or /I_SPACE, the debugger will examine a D-space address. For example, the following command causes the debugger to examine line 4 of the program code in **D-space**:

```
DBG>EXAMINE %LINE 4
```

To examine an I-space address, you must use /INSTRUCTION or /I_SPACE with the EXAMINE command.

6.6.2 EXAMINE Command Parameter

The address parameter specifies the location you want to display. It can be a simple address or an address expression, and it can contain any valid symbolic reference.

MACRO-11 and FORTRAN-77 programmers can examine a range of locations by specifying the following:

```
EXAMINE address:address
```

This command causes the contents of a range of locations to be displayed starting with the first address specified, up to and including the second address.

COBOL programmers can use the same command, but they cannot use symbol names as part of the address range. Also, you cannot use arithmetic operations in COBOL in conjunction with user symbols. For example, you can use the command EXAMINE VAR1 but not EXAMINE VAR1+10.

6.6.3 EXAMINE Command Examples

You can EXAMINE memory locations by specifying a virtual address and by specifying a symbol as in the following example:

```
DBG>DEFINE SUM = 15346
DBG>EXAMINE SUM
15346:6083
```

Both EXAMINE commands display the contents of the same memory location.

In the following example, which shows the effect of the [NO]SYMBOL qualifier when used with the EXAMINE command, X is a program symbol.

```
DBG>EXAMINE/SYMBOL X
MYPROG\X:  6083
DBG>EXAMINE/NOSYMBOL X
15346:  6083
```

Both EXAMINE commands display the same memory location. The difference in the results is in how the debugger displays the output. In the first command, the debugger displays the name of the memory location as a symbol; in the second command, the debugger displays the name of the memory location as a virtual address.

Only the debugger display is affected by mode values.

The symbol mode also affects the way the processor status word (PS) and the floating-point status word (FS) are displayed. When the mode is symbolic, the debugger gives you a formatted display of the contents of the PS and the FS. For example:

```
DBG>EXAMINE/SYMBOL %PS
%PS:     CURMOD PREMOD REGS xxx IPL T N Z V C
           00     00    0   000 000 0 0 0 0 0
DBG>EXAMINE/SYMBOL %FS
%FS:     ER ID IUV IU IV IC LF LI T MM N Z V C
          0  0  0   0  0  0  0  0 0  0 0 0 0 0
```

If the mode is nonsymbolic, the display of the PS and FS is not formatted. For example:

```
DBG>EXAMINE/NOSYMBOL %PS
%PS:    0
DBG>EXAMINE/NOSYMBOL %FS
%FS:    0
```

6.7 Altering Memory Locations

The debugger allows you to alter the contents of memory locations with the DEPOSIT command. You can deposit values into any resident program location.

The DEPOSIT command has the following format:

```
DEPOSIT [/qualifier] address=value-expression
        /ASCII[:n]
        /BYTE
        /D_FLOAT
        /D_SPACE
        /FLOAT
        /LONG
        /INSTRUCTION
        /I_SPACE
        /RAD50
        /WORD
        /BINARY
        /DECIMAL
        /HEXADECIMAL
        /OCTAL
        /[NO]SYMBOL
```

6.7.1 DEPOSIT Command Qualifiers

When you issue the DEPOSIT command without qualifiers, the debugger converts the value denoted by the value expression parameter to the data type associated with the address. It then stores that value in the designated address. If the address does not have an associated type, the debugger interprets the value-expression as being of the default data type.

The qualifiers you use with the DEPOSIT command are mode and data-type qualifiers. The mode qualifiers determine what radix mode is used to interpret the address and the value expression specified with the command. The data-type qualifiers control how the debugger interprets the value expression. Therefore, when you issue the DEPOSIT command with a data-type qualifier, the debugger ignores the data type associated with the memory location, and deposits the value using the data type specified by the qualifier. However, the debugger does not associate the data type specified with a DEPOSIT command qualifier with the memory location. After the value is deposited, the location is still interpreted as having its compiler-generated type or, if it is an untyped location, the default debugger type. Therefore, if you use the EXAMINE command without qualifiers to look at the location, the value displayed will be in the format of the compiler-generated type or the default debugger type. If the value stored in that location must be represented in some other type to be meaningful, you must specify a data-type qualifier with the EXAMINE command. Also, your program always treats the values stored in program symbols in their compiler-generated type; the debugger cannot change the types of variables in your program.

If you are altering memory locations of a task in I- and D-space, and use no qualifier at all, or any qualifier except /I_SPACE, the debugger alters a D-space address. To deposit a value in I-space, you must use the /I_SPACE qualifier with the DEPOSIT command.

6.7.2 DEPOSIT Command Parameters

The address parameter specifies the memory location to which you want to deposit a value. It can be a symbol or a virtual address. The value-expression parameter specifies the value you want to deposit in the memory location.

MACRO–11 and FORTRAN–77 programmers can use a single DEPOSIT command to deposit more than one value by listing the expressions to be deposited on the right side of the equal sign and separating them with commas. The debugger deposits the first value expression into the location denoted by the address, then deposits subsequent value expressions into the logical successors of that memory location.

6.7.3 Depositing ASCII Strings

To deposit an ASCII string, you must 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 overwrites the existing string leaving characters beyond the end of the new string unchanged (MACRO–11 and FORTRAN–77 programs) or padding the excess bytes with ASCII blanks (COBOL–81 programs).

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 characters in the string; otherwise, the debugger deposits only the first two bytes of your character string.

6.7.4 Depositing Radix-50 Strings

You must use the /RAD50 qualifier with the DEPOSIT command to deposit a value expression that is a Radix-50 string. This qualifier identifies the value expression as being of the data type Radix-50. The value expression must be delimited by quotation marks or apostrophes. If the length of the quoted string is not a multiple of three characters, it is padded on the right with blanks when it is stored because three Radix-50 characters occupy two bytes in memory and the debugger stores all Radix-50 strings in two-byte increments.

6.7.5 DEPOSIT Command Examples

In the following example, two successive locations are examined in Radix- 50, and then the value of the first location is changed with the DEPOSIT command.

```
DBG>EXAMINE/RAD50 ARR(1):ARR(2)
MAIN\ARR(1)<0,16>:  HAL
MAIN\ARR(2)<0,16>:  E
DBG>DEPOSIT/RAD50 ARR(1) = "GEG"
DBG>EXAMINE/RAD50 ARR (1)
MAIN\ARR(1)<0,16>:  GEG
```

The angle brackets enclose numbers that show the starting point and number of bits being examined. The location is displayed starting at bit 0 through bit 15. These angle brackets appear only when you display a Radix-50 value.

The following example shows how to deposit an 18-character ASCII string in the variable NAME, which is in the routine called SUB1.

```
DBG>DEP/ASCII:18 SUB1\NAME = "The patient's name"
```

The following example shows the effect of using a data type qualifier with the DEPOSIT command.

```
DBG>EXAMINE VAR
VAR: B
DBG>DEPOSIT/BYTE VAR=65
DBG>EXAMINE VAR
VAR: A
DBG>EXAMINE/BYTE VAR
VAR: 65
```

The location VAR has the type ASCII associated with it. When the DEPOSIT/BYTE command is issued, the debugger stores the integer 65 in the memory location referred to by VAR. The first EXAMINE command displays this location with its associated type of ASCII. However, when the /BYTE qualifier is specified, the debugger displays the contents of VAR as an integer.

Chapter 7

Ending the Debugging Session

When you find errors in your program you want to correct, you must leave the debugger and correct them in your source code. However, before you leave the debugger, it is a good idea to check and see if you have looked at all the program locations in which you suspect your program contains errors. This chapter explains how you display all your eventpoints and several ways you can safely exit from the debugger.

7.1 Preparing to Leave the Debugger

Before you leave the debugger, issue the following SHOW command:

```
DBG>SHOW ALL
%DEBUG-I-NOBRKSET, there are no breakpoints currently set
%DEBUG-I-BRK_ENABLED, the recognition of breakpoints is enabled
%DEBUG-I-NOTRCSET, there are no tracepoints currently set
%DEBUG-I-TRC_ENABLED, the recognition of tracepoints is enabled
%DEBUG-I-CURR_NOLANG, Current language is unknown
%DEBUG-I-MODES, modes: symbolic, decimal
%DEBUG-I-OUTPUT, output: noverify, terminal, not logging to "DB2:[303,52]DEBUG.LOG"
%DEBUG-I-SCOPE, scope: 0 [ = MAIN ]
%DEBUG-I-TYPE, type: word integer
%DEBUG-I-STEPTYPE, step type: by line, over routine calls
```

This command causes the debugger to display all the breakpoints and tracepoints you have set. If you have not used some of the program controllers that you set, you may not have found all the errors in your program.

In addition to breakpoints and tracepoints, the SHOW ALL command displays the current settings of mode, output, type, and step conditions. You can use this command when you want to see how the debugger is configured.

7.2 Exiting the Debugger

To leave the debugger, issue the EXIT command as follows:

```
DBG>EXIT
```

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

After you terminate a debugging session, you must run your program again to restart the debugger.

7.3 Using Control Commands

You can use the control commands CTRL/C and CTRL/Z to interrupt the debugger, but you should verify that the control command you use has the effect you expect before you use it.

The CTRL/C command does the following:

- On RSX-11M-PLUS, RSX-11M, and Micro/RSX, pressing CTRL/C initiates one of the following reponses:
  - When your terminal is set at NO CONTROL C (NOCTRLC), pressing CTRL/C interrupts your task and returns control to your operating system.
  - When your terminal is set at CONTROL C (CTRLC), pressing CTRL/C aborts your task and returns you to your operating system. You must then rerun your program to reenter the debugger.
- On the P/OS, pressing CTRL/C aborts your task. You must then rerun your program to reenter the debugger.
- On RSTS/E and Micro/RSTS, pressing CTRL/C interrupts program execution and returns control to the debugger.
- On a VAX/VMS system, pressing CTRL/C interrupts the debugging session and returns you to DCL. If you enter the CONTINUE command before issuing any other commands, you are returned to the DBG> prompt. However, if you issue a command other than the CONTINUE or SPAWN command at the DCL prompt, you must run your program again to restart the debugger. If you enter the SPAWN command upon exiting the subprocess, you can type CONTINUE to resume debugging.

If you interrupt the debugger using the CTRL/C command on VAX/VMS, the debugger background task runs after you have received the DCL prompt. You should issue the following SHOW command:

```
$ SHOW PROCESS/SUBPROCESS
```

The name of the debugger background task process appears in response to this command. To stop the background task from running, issue the STOP command as follows:

```
$ STOP process-name
```

This situation also occurs if you let your program run out of the debugger. You should always set a breakpoint immediately before the end of your program to avoid this problem.

The CTRL/Z command causes orderly termination of the debugger on all systems. After you terminate a debugging session using the CTRL/Z command, you must run your program again to restart the debugger.

Appendix A

PDP–11 Symbolic Debugger Error Messages

The PDP–11 Symbolic Debugger generates error messages and informational messages while you are using it. These messages are displayed on your terminal and in your log file, if one exists. The message format is as follows:

```
%DEBUG-CODE-MSGNAM, message text
```

The percent character (%) identifies the line as a message. DEBUG signals that the message is from the debugger. The CODE of the message determines its class. The four classes of error messages are described below, in order of greatest to least severity.

Code	Description
F	Fatal; must be corrected before the debugger will perform the requested operation correctly.
E	Error; should be corrected because the operation has been requested incorrectly.
W	Warning; should be investigated because the requested operation may not be designed for use in the way it was specified.
I	Information; no action necessary because the message is only providing information.

The MSGNAM is the name of the message. This part of the message is followed by the message text, which explains why the error message occurred.

This appendix describes only the error messages generated by the debugger, that is, those with a severity of F, E, or W. Informational messages are generally self-explanatory.

The error messages in the following list are arranged in alphabetical order by their message name. The message text, further explanation of the message, and possible corrective action are given for each error message generated by the debugger.

AFTTOBIG, AFTER count too large

Explanation. The delay specification in breakpoints and tracepoints is limited to a maximum of 255.

User Action. Reduce the delay specification to a valid value.

AMBIG, ambiguous keyword

Explanation. The keyword specified is an abbreviation for several keywords that are valid at this point.

User Action. Make the abbreviation unique by increasing its length.

ASCTOOBIG, ASCII type must be less than 256 bytes

Explanation. The number of characters in an ASCII string must be from 1 through 255.

User Action. Reduce the number of characters.

BADINDEX, index cannot be an array

Explanation. An unsubscripted array name was used as an array subscript.

User Action. Correct the expression and reenter the command.

BADINS, illegal opcode

Explanation. The debugger encountered an unexpected error.

User Action. File a Software Performance Report.

BADPACK, The /PACKED qualifier is not valid with this data item

Explanation. The /PACKED qualifier cannot be used with non-numeric data.

User Action. Check the validity of the data item.

BADRAD50, the string "'string'" contains non-RAD50 characters

Explanation. As specified, the string must contain only valid Radix–50 characters.

User Action. Check the contents of the string or use another data type.

BADRANGE, address range 'address' to 'address' is invalid (not ascending)

Explanation. The addresses must be specified in order from lowest to highest.

User Action. Correct the specification.

BADREGUSE, register name may not be used in this context

Explanation. Register names are invalid in certain address contexts. Examples are execution start addresses, and breakpoint and tracepoint addresses.

User Action. Specify an address that is not a register.

BAD_INSTRUCT, unable to decode PDP–11 instruction at 'address'

Explanation. 'Address' does not start a valid PDP–11 instruction.

User Action. Check the validity of the address. It may begin on an odd-byte boundary or refer to an address containing data.

BAD_NAME, bad file name

Explanation. The file system has detected a syntax error in the file specification.

User Action. Correct the file specification.

BAD_PATH, "'pathname'" does not begin a valid pathname

Explanation. The pathname is incorrectly specified.

User Action. Check the validity of the pathname.

BAD_SCOPE, invalid numeric scope

Explanation. The only numeric scope recognized by the debugger is 0-scope, as in SET SCOPE 0.

User Action. Check the validity of the specified scope.

BAD_STAMP, time stamp does not match

Explanation. The time stamp written in the STB file does not match the time stamp written in the task image being debugged. This indicates that the two files were not created by the same task build.

User Action. Use SET STB 'file-name' to access the correct STB file.

BAD_STB, invalid STB file

Explanation. The file specified in the SET STB command has an invalid format.

User Action. Verify that the file name refers to an STB file and specify the proper file name in the SET STB command.

BAD_STB, invalid nesting of FORTRAN routines

Explanation. The debugger has detected an error in the symbol table file.

User Action. Recompile the program or file a Software Performance Report.

BAD_SYMBOL, deposit past end of 'symbol'

Explanation. An attempt was made to deposit too long a value into one of the debugger permanent symbols.

User Action. Check the validity of the data.

BAD_SYMBOL, examine past end of 'symbol'

Explanation. An attempt was made to examine past the end of one of the debugger permanent symbols.

User Action. Correct and reenter the command.

BAD_TBIT, unexpected T-bit trap

Explanation. A CTRL/C was typed during execution of a debugger command.

User Action. None.

BAD_WHEN, WHEN expression is of invalid type

Explanation. The expression in a WHEN clause must be of either LOGICAL type or INTEGER type.

User Action. Check the validity of the command.

BREAK_BAD_ADDR, address is unknown to debugger

Explanation. An error occurred searching the STB file. Additional message(s) will appear giving more details.

User Action. None.

BRK_DISABLED, the recognition of breakpoints is disabled

Explanation. A DISABLE BREAK command is in effect.

User Action. Use the ENABLE BREAK to reverse the effects of the DISABLE BREAK command.

BRK_ENABLED, the recognition of breakpoints is enabled

Explanation. An ENABLE BREAK command is in effect.

User Action. Use the DISABLE BREAK to reverse the effects of the ENABLE BREAK commands.

COBOLERR, COBOL detected run time error

Explanation. The COBOL compiler discovered a fatal run-time error in your program.

User Action. Refer to your COBOL–81 user's manual for an explanation of the run-time error message.

COBOLNOOP, Current language is COBOL - no FORTRAN operators

Explanation. When the current language is COBOL–81, the FORTRAN logical operators (.GT., .LT., and so on) are invalid.

User Action. Do not use the FORTRAN logical operators.

CONVERR, index could not be converted for dimension 'number'

Explanation. A conversion error occurred evaluating a subscript expression. Additional message(s) will appear giving more details.

User Action. None.

CTRLC, unexpected CTRL/C

Explanation. A CTRL/C was typed during execution of a debugger command.

User Action. None.

CURRLANGC81, Current language is COBOL–81

Explanation. The debugger is configured to operate on a COBOL–81 program.

User Action. If your program is not written in COBOL–81, use the SET LANGUAGE command to change the current language.

CURRLANGF77, Current language is FORTRAN–77

Explanation. The debugger is configured to operate on a FORTRAN–77 program.

User Action. If your program is not written in FORTRAN– 77, use the SET LANGUAGE command to change the current language.

CURR_NOLANG, Current language is unknown

Explanation. The debugger is configured to operate on a MACRO–11 program, which is the default debugger current language.

User Action. If your program is not written in MACRO–11, use the SET LANGUAGE command to change the current language.

DATA_LOST, previously opened LOG file cannot be reopened, data will be lost

Explanation. The debugger was unable to open the LOG file. Additional message(s) will appear giving more details.

User Action. None.

DEPTOOLONG, DEPOSIT buffer overflow

Explanation. The DEPOSIT command is limited to a total of 100 bytes of data.

User Action. Divide the DEPOSIT command into several deposit commands.

DIMENERR, number of indexes given does not match dimension

Explanation. An array reference contains an incorrect number of dimensions, that is, too many or too few subscripts.

User Action. Check the validity of the number of dimensions.

EMT, illegal EMT

Explanation. The user program contains incorrect emulator trap (EMT) instructions or the debugger encountered an unexpected error

User Action. Check the EMT instruction or file a Software Performance Report.

EXTOOLNG, you can only EXAMINE up to 'number' bytes at once

Explanation. A single EXAMINE command may only display 100 bytes of data.

User Action. Break the command into two or more separate EXAMINE commands.

FCS_ERR, FCS error code is 'number'

Explanation. The FCS file system has detected an error.

User Action. Consult the *RSX-11M/M-PLUS and Micro/RSX I/O Operations Reference Manual* for an explanation of the error code.

FILE_ERR, error on 'type of' file 'file-operation'

Explanation. An error occurred in a file operation. Additional message(s) will give details on the error encountered.

User Action. None.

FLTDIV, floating point divide by 0

Explanation. The debugger attempted to divide a floating-point number by 0.

User Action. Correct the expression and reenter the command.

FLTOVF, floating point overflow

Explanation. A floating-point overflow occurred during evaluation of an expression.

User Action. Correct the expression and reenter the command.

FLTUND, floating point underflow

Explanation. A floating-point underflow occurred during evaluation of an expression.

User Action. Correct the expression and reenter the command.

FPUND, undefined floating point variable

Explanation. An expression contained a variable whose value was undefined.

User Action. Correct the expression and reenter the command.

FP_ERR, floating point error

Explanation. The debugger encountered an unexpected error.

User Action. Avoid the use of floating point or file a Software Performance Report.

HELP_FMT, length error in following keyword:

Explanation. The debugger HELP file is corrupt.

User Action. Avoid the use of the HELP command and file a Software Performance Report.

ILL_NUM, "'numeric-string'" is not a valid base-'radix' number

Explanation. The 'numeric-string' is specified in the wrong radix.

User Action. Use the SHOW MODE command to determine the correct radix and respecify 'numeric-string' or change the radix mode.

INPCONERR, input conversion error

Explanation. A floating-point number must be a real constant with a decimal point.

User Action. Check the validity of the input.

INTDIV, integer divide by 0

Explanation. The debugger attempted to divide an integer by 0.

User Action. Correct the expression and reenter the command.

INTERROR, 'message text'

Explanation. The debugger has detected an internal coding error.

User Action. File a Software Performance Report.

INTOVF, integer overflow

Explanation. An integer overflow occurred during evaluation of an expression.

User Action. Correct the expression and reenter the command.

INTRPT, unexpected interrupt

Explanation. An interrupt has occurred during execution of a debugger command. Additional message(s) will appear giving more details.

User Action. None.

INVCHAR, illegal character

Explanation. A command line contained an invalid character.

User Action. Check the validity of the command line.

INVDIM, subscript error, 'array name' has dimension 'dimension list'

Explanation. An array reference has one or more dimensions out of range.

User Action. Check the validity of the array reference.

INVLINE, invalid line number 'line'

Explanation. Line numbers must be less than 32768.

User Action. Break your program into smaller pieces or respecify the command, giving the correct line number.

INVTYPE, invalid data type

Explanation. The compiler corrupted the STB file.

User Action. Recompile your program to create a new STB file or file a Software Performance Report for your compiler.

IN_USE, file is already in use

Explanation. The specified file is locked by another task.

User Action. Ensure that the file is available for use.

IOT, IOT instruction executed

Explanation. The debugger encountered an unexpected error.

User Action. File a Software Performance Report.

LINNOTFND, search failed for %LINE 'line-number'

Explanation. No such line number exists in your program.

User Action. Verify the validity of the line number.

LINTOBIG, command line too long

Explanation. A command, including all continuation lines, is limited to 256 characters.

User Action. Simplify the command.

MEM_PROT, memory protect error

Explanation. An attempt was made to execute the contents of system memory.

User Action. Check your program or file a Software Performance Report.

MIXED_TYPE, unable to EXAMINE an address range of mixed type

Explanation. Entities in the specified address range must be either program addresses or register addresses, but not both.

User Action. Specify an address range of one type.

NAMAMBIG, The data-name used in this command is ambiguous

Explanation. The specified data-name appears more than once in the program.

User Action. Specify more COBOL qualification.

NAMTOBIG, the name "'name'" has 'length' characters

Explanation. Symbol name must not exceed six characters.

User Action. Specify the name correctly.

NONNUMLIT, Only a nonnumeric literal can be moved to an alphanumeric item

Explanation. A number cannot be moved into a storage location that expects alphanumeric data.

User Action. Enclose the literal in quotation marks or specify numeric type for the storage location by using a qualifier with the DEPOSIT command.

NOPRED, 'address' has no predecessor

Explanation. Logical predecessor is defined only for elements of arrays, untyped storage locations, instructions, and PDP–11 machine registers.

User Action. Reference the address by its symbolic name or virtual address.

NOQUOTE, string literal missing closing quote

Explanation. String literals must be enclosed by quotation marks.

User Action. Respecify with the closing quote.

NOSUCC, 'address' has no successor

Explanation. Logical successor is defined only for elements of arrays, untyped storage locations, instructions, and PDP-11 machine registers.

User Action. Reference the address by its symbolic name or virtual address.

NOSTEP, unable to STEP/OVER, stepping INTO instead

Explanation. The debugger cannot obtain from the COBOL-81 compiler the information needed to step over routine calls.

User Action. Instead of stepping through all of COBOL's OTS routines as well as your own, set breakpoints and issue the GO command. This problem should be fixed with the next COBOL-81 update.

NOSUCDEP, COBOL-81 does not support DEPOSITs to successive memory locations

Explanation. You cannot store multiple values with a single DEPOSIT command

User Action. Divide the command so that only one value is stored each time you issue the DEPOSIT command.

NOTARRAY, subscripted variable "'name'" is not an array

Explanation. A subscripted variable must be an array.

User Action. Correct the expression and reenter the command.

NOTDEFINE, you do not have a defined symbol "'name'"

Explanation. The debugger does not recognize 'name' as a symbol.

User Action. Use the DEFINE command to define 'name'.

NOTLINKED, STB record has not been processed by TKB

Explanation. The STB file contains records that have not been processed by the bask builder.

User Action. File a Software Performance Report for your task builder.

NOTRES, 'address' is not resident

Explanation. An address specified in an EXAMINE or DEPOSIT command refers to an overlay segment not currently in memory.

User Action. None.

NOT_GSD, unable to locate "'name'" as GSD

Explanation. The specified 'name' could not be found as a global symbol.

User Action. Check the validity of the name or reset your scope so that it is no longer global.

NO_ACTION, unable to link in action "'action'"

Explanation. The debugger could not obtain sufficient storage to execute the indicated action.

User Action. Regain dynamic memory by eliminating unneeded eventpoints and defined symbols.

NO_CALLS, there are no active call frames

Explanation. There are no subprograms currently active.

User Action. None.

NO_CONV, unable to convert datatypes

Explanation. A DEPOSIT command attempted to convert numeric data to character, or character data to numeric.

User Action. Check the validity of the data.

NO_DIR, directory does not exist

Explanation. The directory specified as part of the file specification does not exist.

User Action. Check the validity of the directory specification.

NO_FILE, file does not exist

Explanation. The specified file does not exist.

User Action. Correct the file specification and reenter the command.

NO_FLOAT, COBOL-81 does not support floating point

Explanation. It is invalid to refer to a COBOL-81 storage location as floating point.

User Action. Change the data type specification to one that is legal for COBOL-81.

NO_HIERARCHY, only COBOL supports hierarchical data types

Explanation. The OF and IN keywords are defined only for COBOL-81 users.

User Action. Use pathname syntax to refer to FORTRAN-77 variables that could be ambiguous.

NO_LOG, unable to open log file "'file name'"

Explanation. The debugger is unable to open the LOG file. Additional message(s) will appear giving more details.

User Action. None.

NO_PACKED, FORTRAN-77 does not support packed decimal types

Explanation. It is invalid to refer to a FORTRAN-77 program location as packed.

User Action. Specify a valid FORTRAN-77 data type.

NO_PATHNAME, unable to find 'name' in STB file

Explanation. The debugger does not recognize 'name' as a symbol.

User Action. Check the validity of the name.

NO_READ, no read access to address 'address'

Explanation. The debugger was unable to read the specified address.

User Action. Check the validity of the address.

NO_REAL, logical operations are not legal for REAL

Explanation. Logical operations can be performed only on logical or integer expressions.

User Action. Check the validity of the expression.

NO_SCOPE, 0-SCOPE is invalid for this PC

Explanation. The current PC is not within a program or subprogram compiled with the DEBUG option.

User Action. Specify a pathname for the symbol.

NO_SPACE, internal storage exhausted

Explanation. The debugger could not obtain sufficient dynamic memory to perform the command.

User Action. Regain dynamic memory by eliminating unneeded eventpoints and defined symbols.

NO_STB_FILE, no STB file is currently set

Explanation. A SHOW STB command was issued when no STB file was set.

User Action. Use the SET STB command.

NO_SUBS, Subscripts cannot be specified here

Explanation. Unstructured variable references cannot contain subscripts.

User Action. Check the validity of the variable reference.

NO_WHEN, unable to link in condition "'when clause'"

Explanation. The debugger could not obtain sufficient storage to evaluate the specified condition. The condition is treated as though it evaluated to TRUE.

User Action. Regain dynamic memory by eliminating unneeded eventpoints and defined symbols.

NO_WRITE, no write access to address 'address'

Explanation. The debugger was unable to write to the specified address.

User Action. Check the validity of the address.

NUMERICLIT, Only a numeric literal can be moved to a numeric item

Explanation. Nonnumeric data cannot be moved to a storage location that expects a number.

User Action. Specify a nonnumeric type for the storage location.

NUMTOOLONG, A numeric literal cannot have more than 18 digits

Explanation. Numeric literals must be shorter than 18 digits.

User Action. Shorten the numeric literal.

ODDBPT, you may not set an 'event' at odd address 'address'

Explanation. You cannot set a breakpoint or tracepoint at an odd address.

User Action. Correct the address and respecify it.

ODD_ADDR, odd address trap

Explanation. An attempt was made to begin execution at an odd address.

User Action. Check your program or file a Software Performance Report.

OPERSTOVFL, operand stack overflow

Explanation. The parser has exhausted its operand stack.

User Action. Reduce the complexity of the command.

OPSTOVFL, operator stack overflow

Explanation. The parser has exhausted its operator stack.

User Action. Reduce the complexity of the command.

OUTCONERR, output conversion error

Explanation. The debugger was unable to format the value requested.

User Action. File a Software Performance Report.

OUTOFBNDS, The subscript value specified is out of the legal range

Explanation. The largest value in the array range is smaller than the specified subscript value.

User Action. Check the validity of the subscript value.

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

Explanation. The SET OUTPUT command has been specified with both NOLOG and NOTERM parameters.

User Action. Specify the SET OUTPUT command with either the LOG or TERM parameter.

PERM_DEF, you cannot redefine the permanent symbol 'name'

Explanation. A permanent symbol cannot be defined with the DEFINE command.

User Action. Check the validity of the symbol name.

PROTECTED, file has restricted access

Explanation. The file specified in the current command is protected against access by the debugger.

User Action. Correct the file protection.

PRSTOVFL, parse stack overflow

Explanation. The parser has exhausted its parse stack.

User Action. Reduce the complexity of the command.

RECTOBIG, record was truncated

Explanation. An indirect command file contained a command line of more than 255 bytes in length.

User Action. Correct the command file.

RETURN_BREAKs, can only refer to a routine

Explanation. The address specified with SET BREAK/RETURN must be within a main routine or a called routine.

User Action. Correct the address and respecify it.

RMS_ERR, RMS error code is 'number'

Explanation. The RMS file system has encountered an error.

User Action. Consult the *RMS-11 MACRO-11 Reference Manual* or the *VAX-11 Record Management Services Reference Manual* for an explanation of the error code.

SEE_CALLS, the number_of_frames argument ('count') must be greater than 0

Explanation. A zero argument to SHOW CALLS is invalid.

User Action. Correct the command and reenter it.

SHOTOBIG, SHOW CALL count too large

Explanation. The SHOW CALLS count is a decimal integer in the range 1 through 32767.

User Action. Correct the call-count parameter and reissue the SHOW CALLS command.

STNNAME, the SET STB_FILE command handles STB file names up to 'length' characters

Explanation. The file name specified is longer than allowed by the host file system.

User Action. Correct the file name and reenter the command.

STPTOBIG, STEP count too large

Explanation. The maximum value of the step count is 32767.

User Action. Correct the command and reenter it.

STRTRUNC, string truncated

Explanation. The string being deposited was longer than the length associated with the location deposited into.

User Action. None.

SYNTAX, command syntax error at or near "string"

Explanation. A syntax error was detected in the debugger command.

User Action. Correct the command, and reenter it.

SYNTAXEXPR, Syntax error in expression

Explanation. The arithmetic expression in the debugger command contains a syntax error.

User Action. Check the validity of the arithmetic expression.

TBIT, unexpected T-bit trap

Explanation. Either you typed a CTRL/C during the execution of a STEP command or your program set the T-bit in the processor status word.

User Action. None.

TERM_LOST, error during write of log output to terminal

Explanation. An I/O error occurred during terminal I/O.

User Action. File a Software Performance Report.

TOOFEWSUBS, Command specifies too few subscripts for "string"

Explanation. The structured variable has more dimensions than specified.

User Action. Check the validity of the variable reference.

TOOLONG, file name too long

Explanation. The file name specified is longer than allowed by the host file system.

User Action. Correct the file name and reenter the command.

TOMANSUBS, Too many subscripts specified for this item

Explanation. The item is specified with more subscripts than the structured variable is declared with.

User Action. Check the validity of the variable reference.

TRAP, TRAP instruction executed

Explanation. The debugger encountered an unexpected error.

User Action. File a Software Performance Report.

TRSTOVFL, tree storage overflow

Explanation. The parser has exhausted its tree storage.

User Action. Reduce the complexity of the command.

UNEXPEOF, unexpected end of file

Explanation. The debugger has detected an internal coding error.

User Action. File a Software Performance Report.

UNMATCHED, unmatched end-of-routine record in STB

Explanation. The debugger encountered a format error in the symbol table file.

User Action. File a Software Performance Report.

VARERR, symbol "'name'" is not an array variable

Explanation. The variable was specified with subscripts, but is not dimensional.

User Action. Specify the variable without a subscript list.

WHEN_ERR, error in evaluation of WHEN

Explanation. An error was encountered attempting to evaluate a WHEN clause. The WHEN clause is treated as though it evaluated to TRUE.

User Action. Correct the WHEN clause.

INDEX

0 symbol • 3-11
%FS symbol • 3-2
%LINE symbol • 3-2
  in pathnames • 3-8
%NAME symbol • 3-2
%PC symbol • 3-2
%PS symbol • 3-2
%SEGMENT symbol • 3-2
  in pathnames • 3-9
%SP symbol • 3-2
@ Command • 2-5
  example • 2-6

A

Address
  as address expression • 3-4
  as parameter • 3-4
  as simple address • 3-4
  computation of • 3-4
  data types • 6-2
  integers in • 3-4
  line numbers in • 3-4
  operators in • 3-4
  permanent symbol in • 3-4
  program symbol in • 3-4
Address arithmetic • 6-8
/AFTER:n qualifier • 4-5
ASCII strings
  default length • 6-14
  depositing • 6-13
$AUTO • 4-4

B

Backslash (\) symbol • 3-11
Breakpoint
  disabling • 4-8
  displaying • 4-8
  duration of • 4-8
  effect of • 4-3
  enabling • 4-8
  setting • 4-4

C

/CALLS qualifier • 4-5
CANCEL ALL command • 7-1
CANCEL BREAK command • 4-8
CANCEL MODE command • 6-5
CANCEL OUTPUT command • 2-3
CANCEL SCOPE command • 3-12
CANCEL STEP command • 5-5
CANCEL TRACE command • 4-8
CANCEL TYPE command • 6-4
Command file
  See Indirect command file
Comment
  in indirect command file • 2-5
  in log file • 2-3
Control commands • 7-2
CTRL/C • 7-2
CTRL/Z • 7-3
Current location symbol • 3-2
Current scope, displaying • 3-12
Current value symbol • 3-2

D

Data type • 6-1
- changing the default • 6-2
- compiler-generated • 6-1
- debugger default • 6-2
- displaying default • 6-4
- for addresses • 6-2
- recognized by the debugger • 6-3

Debugger
- displaying configuration • 7-1
- displaying modes • 6-5
- interrupting • 7-2
- invoking • 1-4
  - MACRO-11 command • 1-5
  - ODL file • 1-5
  - with nonoverlaid kernel • 1-5, 1-7
  - with overlaid kernel • 1-5

Debugger features • 1-1
Debugger start-up • 1-4
Debugger termination • 1-8, 7-2
Debugging session
- online help • 1-4
- optimized code • 1-3
- planning • 1-2

Default language
- setting • 2-1

DEFINE command • 3-3, 3-5
Defined symbol • 3-3
Deposit
- successive • 6-13

DEPOSIT command • 6-12, 6-13, 6-14
DISABLE BREAK command • 4-8
DO (action) parameter • 4-6

E

ENABLE BREAK command • 4-8
EVALUATE command • 6-6, 6-7
EXAMINE command • 6-9, 6-10
- displaying a range • 6-10

Exclamation point
- in indirect command file • 2-5
- in log file • 2-3

EXIT command • 1-8, 7-2
Extended pathname • 3-9
- and %SEGMENT • 3-9

Extended pathname (cont'd.)
- example • 3-10
- syntax • 3-9

F

File
- see Indirect command file
- See Log file
- See STB file

Filespec
- format • 2-4
- full • 2-6
- in @ command • 2-6
- in log file • 2-4

%FS symbol • 3-2

G

GO command • 5-5, 5-6

H

HELP command • 1-4
Help facility • 1-4

I

Indirect command file • 2-5
- and log file • 2-5
- creating • 2-5
- example • 2-6
- execution • 2-5
- including comments • 2-5
- invoking • 2-5
- nesting • 2-6
- VERIFY parameter • 2-2

Integer
- in addresses • 3-4
- in pathnames • 3-8

L

Line number
- as permanent symbol • 3-2
- in addresses • 3-4
- in STB file • 3-6

%LINE symbol • 3-2
Literal • 6-1
Log file • 2-3
  as indirect command file • 2-3, 2-5
  comment in • 2-3
  creating new version • 2-4
  default extension • 2-4
  displaying name • 2-5
  example • 2-4
  name of • 2-4
Logical predecessor symbol • 3-3
Logical successor symbol • 3-3

M

$MARKS • 4-4
Mode
  canceling • 6-5
  definition of • 6-4
  radix • 6-4
  setting • 6-4
  symbol • 6-4
    and FS • 6-11
    and PS • 6-11

N

%NAME symbol • 3-2

O

Operator
  in addresses • 3-4
  in value expressions • 6-6
Optimized code, debugging • 1-3
Output configuration
  canceling • 2-3
  default • 2-2
  displaying • 2-3
  example of setting • 2-3
  setting • 2-2
Overlay and pathname • 3-9

P

Pathname
  See also Extended pathname
Pathname (cont'd.)
  See also Simple pathname
  for MACRO–11 • 3-8
  in SET SCOPE command • 3-11
  truncated • 3-11
%PC symbol • 3-2
Period • 3-2
Permanent symbol • 3-1
  for registers • 3-2
  in addresses • 3-4
Program symbol • 3-6
  data types • 6-1
  in MACRO–11 • 3-6
  referencing • 3-7
%PS symbol • 3-2

R

Radix–50 strings
  default length • 6-14
  depositing • 6-14
$RDSEG • 4-4
Register, permanent symbols for • 3-2
/RETURN qualifier • 4-5
Routine calls
  displaying active • 4-1

S

Scope • 3-7
  canceling • 3-12
  displaying • 3-12
  prefix • 3-11
    syntax • 3-11
  setting • 3-10
  specifying • 3-8
%SEGMENT symbol • 3-2
SET BREAK command • 4-4, 4-6, 4-7
SET LANGUAGE command • 2-1
  default for MACRO–11 • 2-1
  for MACRO–11 • 2-1
SET LOG command • 2-4
SET MODE command • 6-4
  overriding • 6-4
SET OUTPUT command • 2-2, 2-3
SET SCOPE command • 3-10, 3-11
  pathname in • 3-11

SET STB command • 3-6
SET STEP command • 5-4, 5-5
SET TRACE command • 4-4, 4-6, 4-7
SET TYPE command • 6-2
    overriding • 6-3
SHOW ALL command • 7-1
SHOW BREAK command • 4-8
SHOW CALLS command • 4-1, 4-2
    explanation of display • 4-1
SHOW LOG command • 2-5
SHOW MODE command • 6-5
SHOW OUTPUT command • 2-3
SHOW SCOPE command • 3-12
SHOW STB command • 3-7
SHOW STEP command • 5-5
SHOW TRACE command • 4-8
SHOW TYPE command • 6-4
Simple pathname • 3-8, 3-9
%SP symbol • 3-2
STB file • 3-6
    contents • 3-6
    default extension • 3-7
    name of • 3-6
STEP command • 5-1, 5-2, 5-3
    duration of • 5-1, 5-2
    sequence of • 5-1
Step conditions
    canceling • 5-5
    changing • 5-4
    default • 5-4
    displaying • 5-5
Symbol
    ambiguous reference to • 3-7
    as defined symbol • 3-3
    as permanent symbol • 3-1
    as program symbol • 3-6
    backslash (\) • 3-11
    containing period • 3-2
    creating • 3-3
    current location • 3-2
    current value • 3-2
    for registers • 3-2
    in pathnames • 3-8
    %LINE • 3-2
        in pathnames • 3-8
    logical predecessor • 3-3
    logical successor • 3-3
    making unique • 3-7
Symbol (cont'd.)
    number 0 • 3-11
    %SEGMENT • 3-2
    types of • 3-1
Symbol table file
    See STB file
System routines • 4-4

T

Traceback information • 4-3
Tracepoint
    disabling • 4-8
    displaying • 4-8
    duration of • 4-8
    effect of • 4-3
    enabling • 4-8
    setting • 4-4

U

UNDEFINE command • 3-5

V

Value expression • 6-6
    operators • 6-6
    quoted strings in • 6-6
Variable name
    in STB file • 3-6

W

WHEN parameter • 4-6

PDP–11 Symbolic Debugger
User's Guide
AA–FA60A–TK

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