PDP–11 Symbolic Debugger MACRO-11 User's Guide

Order Number: AA–HL43A–TK

February 1986

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

February 1986

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

ZK3224

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

Intended Audience

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

Operating Systems and Versions

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

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

Structure of This Document

This manual is organized as follows:

- Chapter 1 explains how to include support for the debugger in your task and describes the commands you use to invoke the debugger. It also lists the major debugger features.
- Chapter 2 explains how to configure the debugger's default output, make a record of a debugging session, and use a command file to control the debugger.
- Chapter 3 describes the symbols the debugger recognizes and explains how to define your own symbols. It also discusses strategies for making symbols unique.

- Chapter 4 explains how to set breakpoints and tracepoints in your program.
- Chapter 5 describes two methods of executing your program in the debugger.
- Chapter 6 discusses the data types the debugger recognizes, the two modes of debugger operation, and a command that helps you determine memory addresses and perform arithmetic. It also explains how to examine and alter memory locations.

Associated Documents

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

- *PDP–11 Symbolic Debugger User's Guide.* This manual explains general use of the debugger with all supported languages.
- *PDP–11 Symbolic Debugger Installation Guide.* This manual explains the debugger installation procedure on all supported operating systems.
- *PDP–11 Symbolic Debugger Quick Reference.* The quick reference manual lists the syntax of each debugger command and its qualifiers and parameters.
- *PDP–11 Symbolic Debugger Information for FORTRAN-77 Users.* This manual gives information of interest to debugger users who program in FORTRAN-77.
- *PDP–11 Symbolic Debugger Information for COBOL-81 Users.* This manual gives information of interest to debugger users who program in COBOL-81.

NOTE

Where language-specific exceptions to the general case exist, the information given in this manual, specific to MACRO–11 takes precedence over general information presented elsewhere.

Conventions Used in This Document

The following are conventions that are followed throughout this manual:

Convention	Meaning
UPPERCASE	Uppercase words and letters used in examples indicate that you type the word or letter exactly as shown.
lowercase	Lowercase words and letters used 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

Including Debugger Support

The PDP-11 Symbolic Debugger helps you find logical and programming errors in a successfully compiled program that does not run correctly. When you are ready to use the debugger on a program, you must include it in your task. This chapter explains how to include debugger support in your task and describes the commands you issue to invoke the debugger. It also explains how to exit from the debugger and summarizes the general debugger features listed in the *PDP-11 Symbolic Debugger User's Guide*.

1.1 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, read Section 1.1.1.

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 1.1.2

1.1.1 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
```

1.1.2 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. The following are the MCR commands you use 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.

The following are the DCL commands you use 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 preceeding examples.

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

The following operating systems provide instruction and data space support:

- RSX–11M–PLUS Version 2.1 or higher
- RSTS/E Version 9.0 or higher
- Micro/RSX Version 3.0 or higher
- Micro/RSTS Version 2.0 or higher

With this feature, you may be able to run significantly larger programs than is otherwise possible on 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.

1.1.4 Exiting the Debugger

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

```
DBG>EXIT
```

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

1.2 Features of the Debugger

The PDP–11 Symbolic Debugger supports the following features:

- It is interactive.
- It is symbolic.
- It supports overlaid programs.
- It supports I- and D-Space programs on the operating systems on which I- and D-Space is available.
- It gives online HELP.

Chapter 2

Controlling Debugger Input and Output

This chapter explains how to set the default programming language to MACRO–11 and describes what happens when this default is set. It also explains how to configure aspects of the debugger input and output format that are not specific to programming in MACRO–11.

2.1 Setting the Default Language

When you enter the debugger, an informational message is displayed, indicating the programming language in which the debugger expects your program to be written. If this message does not say UNKNOWN, issue the following command:

```
DBG>SET LANGUAGE NONE
```

This command informs the debugger that your program is written in MACRO–11. When the current language is UNKNOWN, the debugger expects input in word integer format and displays output in this format.

2.2 Changing the Default Output

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

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

2.2.1 SET OUTPUT Command Parameters

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

2.2.2 The SHOW OUTPUT and CANCEL OUTPUT Commands

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

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

2.3 Using Log Files

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

```
SET LOG filespec
```

This command causes the debugger to write information in the file named by filespec.

2.3.1 Log File Example

The following is an example of a log file.

```
SHOW OUTPUT
!%DEBUG-I-OUTPUT: noverify, terminal, logging to "SY:[33,62]MYPROG.LOG;1"
SET LANGUAGE NONE
SHOW LANGUAGE
!%DEBUG-I-NOCURRLANG, Current language is UNKNOWN
SET LOG RECORD
```

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

2.3.2 The SHOW LOG Command

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

2.4 Using Indirect Command Files

Indirect command files are files that contain a series of debugger commands. Any valid debugger command can be included in an indirect command file, but none of them are checked for valid syntax before they are executed. Instead, the debugger issues an error message when it encounters commands with invalid syntax and continues execution with the next line in the command file. You can include comments in your indirect command file if you preface them with an exclamation mark (!).

You execute an indirect command file as follows:

```
@filespec
```

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

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 to define symbols.

3.1 Kinds of Symbols

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

3.1.1 Permanent Symbols

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

Table 3–1: Debugger Permanent Symbols

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

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

Symbol	Meaning
%PS	Processor status word
%FS	Floating-point status word
%NAME	Program symbol name follows
%SEGMENT	Overlay segment name follows
\	Current value
.	Current location
RET	Logical successor
^	Logical predecessor

The %NAME debugger permanent symbol allows you to refer to symbols in your program that contain periods. For example, to refer to the A.OR.B program symbol you specify the following:

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

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

See the *PDP-11 Symbolic Debugger User's Guide* for more information on the other debugger permanent symbols.

3.1.2 Program Symbols

When debugging a MACRO–11 program, you can refer to global symbols. However, all the records required for the debugger to resolve references to local symbols, such as 10$, are not generated. Therefore, references to local symbols may generate unexpected results.

3.1.3 Defined Symbols

During a debugging session, you can create a new debugger symbol or change an existing symbol by using the DEFINE command. These symbols remain in effect until you terminate the debugging session. The DEFINE command has the following format:

```
DEFINE symbol=address
```

The symbol parameter specifies what name you want to use to refer to program data or program addresses. A debugger symbol name must:

- Be composed of only alphanumeric characters (the letters A to Z and the numbers 0 through 9) and dollar signs ($)
- Be no more than 6 characters long
- Not begin with a number

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

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 control program execution.

4.1 The Effects of Breakpoints and Tracepoints

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

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

1. Suspends program execution immediately before the instruction at the specified location is executed.
2. Tests the value expression in the WHEN clause if one was specified in the SET BREAK command (see Section 4.1.2.2). If this value expression is false, program execution continues. However, if the value expression is true, activation of the breakpoint continues as described in Step 3.
3. Displays the name or the virtual memory location where execution has been suspended.
4. Executes commands in a DO sequence if one was specified in the SET BREAK command (see Section 4.1.2.3).
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 expression in the WHEN clause if one was specified in the SET TRACE command (see Section 4.1.2.2). If this value expression is false, program execution continues. However, if the value expression is true, activation of the tracepoint continues as described in Step 3.
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 (see Section 4.1.2.3).
5. Resumes execution at the current program counter.

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

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

```
SET BREAK [/qualifier] [address] [WHEN(value-expr)] [DO(action)]
          /AFTER:n
          /CALLS
```

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

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

4.1.1 SET BREAK and SET TRACE Qualifiers

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

4.1.1.1 The /AFTER:n Qualifier

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

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.

4.1.1.2 The /CALLS Qualifier

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

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

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

If you specify /CALLS, you cannot specify any other qualifier in the command.

4.1.2 SET BREAK and SET TRACE Parameters

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

4.1.2.1 The Address Parameter

The address parameter specifies the instruction 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.

4.1.2.2 The WHEN Parameter

The WHEN parameter allows you to control whether or not a program controller is activated based on a condition specified by the value-expr parameter.

4.1.2.3 The DO Parameter

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

4.1.3 Commands Related to SET BREAK and SET TRACE

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

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

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

The format of the CANCEL BREAK command is:

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

The format of the CANCEL TRACE command is:

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

The /ALL qualifier cancels either all breakpoints or all tracepoints currently set in a program. The /CALLS qualifier cancels either all the breakpoints or all the tracepoints at JSR and RTS instructions.

To prevent breakpoints and tracepoints from being activated, issue either the DISABLE BREAK or the DISABLE TRACE command. DISABLE commands do not cancel program controllers, they prevent the activation of program controllers until you enable them.

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

Chapter 5

Starting the Program

When you are ready to execute your MACRO–11 program, use either the STEP command or the GO command. This chapter explains how to use these commands.

5.1 Executing a Specified Number of Commands

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

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

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

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

The syntax of the STEP command is:

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

5.1.1 STEP Commmand 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. 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. Lines in called routines are not counted to satisfy a step-count when the /OVER qualifier is in effect. Note that called routines can be either a routine you wrote or a system routine.

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

5.1.2 STEP Command Parameter

The step-count parameter specifies the number of MACRO–11 instructions you want the debugger to execute. Step-count must be given as a decimal integer.

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

5.2 Changing the Default Step Conditions

If you issue the STEP command without qualifiers when you start up the debugger, the debugger executes your program according to its default step conditions. By default, the debugger counts only instructions 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.

The format of the SET STEP command follows:

```
SET STEP parameter
        INTO
        OVER
```

5.2.1 SET STEP Command Parameters

The INTO parameter specifies that the debugger step through called routines. The OVER parameter 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.

5.2.2 The SHOW STEP and CANCEL STEP Commands

To display the current step conditions, issue the command:

```
SHOW STEP
```

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

```
CANCEL STEP
```

5.3 Executing an Undetermined Number of Commands

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

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

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

The GO command has the following syntax:

```
GO [address]
```

The address parameter allows you to specify an address at which to start program execution. It can be any legal simple address or address expression.

Chapter 6

Manipulating Data

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

6.1 Data Types in the Debugger

The debugger associates data types with literals, program symbols, and memory addresses.

The data type of a literal depends on the format of the literal. The following list of the literal data types that the debugger supports explains how these data types are associated with literals.

- The data type integer is associated with literals that do not contain a decimal point.
- Literals that contain decimal points are associated with the data type floating point or double-precision floating point (D_FLOAT).
- The quoted string data type is associated with strings that are enclosed in quotation marks.

The data types of program symbols and memory addresses are word integer by default. You can change the data type the debugger uses to interpret memory addresses with the SET TYPE command. This command has the following format:

```
SET TYPE datatype
        ASCII[:n]
        BYTE
```

```
D_FLOAT
FLOAT
LONG
INSTRUCTION
RAD50
WORD
```

The datatype parameter determines what data type the debugger uses as the default data type. All memory addresses are interpreted as being that data type until another SET TYPE command is issued. You can override the default data type by specifying a data type qualifier on the EXAMINE and DEPOSIT command lines. (The EXAMINE and DEPOSIT commands are explained later in this chapter.)

To determine the data type in effect, you can issue the SHOW TYPE command. This command causes the debugger to display an informational message that tells you the currently active data type.

6.2 Debugger Modes

The PDP–11 Symbolic Debugger supports radix modes and symbol modes. These modes work together to control the form in which the debugger interprets and displays information. The default debugger modes for MACRO–11 are a decimal radix mode and a symbolic symbol mode. When these modes are in effect, the debugger interprets and displays numbers as decimals. It also displays the symbol that refers to a memory address, instead of the address itself.

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

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

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

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

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

```
DBG>CANCEL MODE
```

This command returns the mode settings to their defaults.

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

```
DBG>SHOW MODE
```

6.3 Determining the Virtual Address of Symbols

Before you examine and modify memory, you should know how to determine what 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. This command has the following format:

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

6.3.1 EVALUATE Command Qualifiers

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

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

6.3.2 EVALUATE Command Parameters

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

6.4 Value Expressions

Value expressions can 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.

The following are legal operators and delimiters in value expressions listed in order from highest to lowest precedence.

- Parentheses
- Unary minus
- Multiplication and division
- Plus and minus

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

6.5 Displaying Memory Locations

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

```
EXAMINE [/qualifier]  address
        /ASCII[:n]
        /BYTE
        /D_FLOAT
        /FLOAT
```

```
/INSTRUCTION
/LONG
/RAD50
/WORD

/BINARY
/DECIMAL
/HEXADECIMAL
/OCTAL
/[NO]SYMBOL

/D_SPACE
/I_SPACE
```

6.5.1 EXAMINE Command Qualifiers

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

If your system supports I- and D-space, you can use the /I_SPACE and /D_SPACE qualifiers with the EXAMINE command when you know that the item you are examining is stored in either the I-space (instruction storage) or D-space (data storage) portion of memory. Using no qualifier, or any qualifier except /INSTRUCTION or /I_SPACE, with EXAMINE causes the debugger to 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. For more information on I-space and D-space, see your operating system documentation.

6.5.2 EXAMINE Command Parameter

The address parameter specifies the location you want to display. It can be a simple address or an address expression.

6.6 Altering Memory Locations

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

```
DEPOSIT [/qualifier] address=value expression[,value expression]
        /ASCII[:n]
        /BYTE
        /D_FLOAT
        /FLOAT
        /LONG
        /RAD50
        /WORD

        /BINARY
        /DECIMAL
        /HEXADECIMAL
        /OCTAL

        /D_SPACE
        /I_SPACE
```

6.6.1 DEPOSIT Command Qualifiers

You can use mode and data type qualifiers with the DEPOSIT command. The mode qualifiers determine what radix mode the debugger uses to interpret the expressions you specify. The data type qualifiers control how the debugger interprets the value you specify.

MACRO–11 users who have I- and D-space support on their systems can use the /I_SPACE and /D_SPACE qualifiers. You use these qualifiers when you know that the item you are depositing should be stored in the I-space portion of memory or in the D-space portion of memory. The default is the /D_SPACE qualifier. For more information on I-space and D-space see your operating system documentation.

6.6.2 DEPOSIT Command Parameters

The address parameter specifies the location in memory to which you want to deposit a value. The value expression parameter specifies the value you want to deposit in the memory location. If you specify more than one value expression, the debugger deposits the first value at the location denoted by address expression and deposits subsequent value expressions at locations denoted by logical successors to address expression.

6.6.3 Depositing ASCII Strings

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

When you want to deposit an ASCII string at an address represented by non-ASCII characters, 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.

6.6.4 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 value expression as being of the data type Radix-50. The string 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. The default length of Radix-50 values is two bytes.

INDEX

PDP–11 Symbolic Debugger
Macro–11 User's Guide
AA–HL43A–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

PDP–11 Symbolic Debugger
Macro–11 User's Guide
AA–HL43A–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