PDP-11 Symbolic Debugger FORTRAN-77 User's Guide

Order Number: AA-FA64A-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

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	

ZK2958

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 FORTRAN-77 programmers who have read and understand the *PDP-11 Symbolic Debugger User's Guide* and know how to use the host operating system.

Operating Systems and Versions

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

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

Structure of This Document

This manual is organized as follows:

- Chapter 1 explains how to include support for the debugger in your task and describes the commands you use to invoke the debugger. It also lists the major debugger features for all users as well as the debugger features that are specific to FORTRAN-77.
- Chapter 2 explains how to configure the debugger's default output, make a record of a debugging session, and use a command file to control the debugger.

- Chapter 3 describes the symbols the debugger recognizes and explains how to define your own symbols. It also discusses strategies for making symbols unique.
- Chapter 4 explains how to set breakpoints and tracepoints in your program.
- Chapter 5 describes two methods of executing your program in the debugger.
- Chapter 6 discusses the data types the debugger recognizes, the two modes of debugger operation, and a command that helps you determine memory addresses and perform arithmetic. It also explains how to examine and alter memory locations.
- Chapter 7 gives an example of debugger use with a FORTRAN-77 program.

Associated Documents

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

- *PDP–11 Symbolic Debugger User's Guide.* This manual explains general use of the debugger with all supported languages.
- *PDP–11 Symbolic Debugger Installation Guide.* This manual explains the debugger installation procedure on all supported operating systems.
- *PDP–11 Symbolic Debugger Quick Reference Guide.* The quick reference manual lists the format of each debugger command and its qualifiers and parameters.
- *PDP–11 Symbolic Debugger COBOL-81 User's Guide.* This manual gives information 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 FORTRAN-77, 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 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.
$\boxed{\text{RET}}$	The $\boxed{\text{RET}}$ 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 summarizes the general debugger features listed in the *PDP–11 Symbolic Debugger User's Guide* and lists those debugger features that are available only to FORTRAN-77 users.

1.1 How to Include the Debugger in Your Task

You can include debugger support in your task, either as an overlaid kernel or as a nonoverlaid kernel.

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 file (ODL) that combines your program with the debugger. You then compile, link, 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 is loaded automatically, 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 qualifiers when you link your program. If you want to use the nonoverlaid kernel, refer to Section 1.1.2

1.1.1 Invoking the Overlaid Configuration

To use the overlaid debugger kernel, you must create an ODL file. The following example shows 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.FTN.

```
        .ROOT   USROT$,$DALL
USROT$: .FCTR   MYPROG-$DROOT-LB:[1,1]F4POTS/LB
@LB:[1,1]PDPDBG
        .END
```

As shown here, the ODL file you create for your task must include an FCTR statement that concatenates your program with part of the debugger kernel ($DROOT) and the FORTRAN-77 library (F4POTS). This FCTR statement must be declared in the ROOT statement as a co-tree with the rest of the debugger kernel ($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 and library 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 must compile, link, and run your program to invoke the debugger. When you compile 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.

The following MCR and DCL commands are used to invoke the debugger.

FOR MCR USERS

```
>F77 myprog,myprog/-SP=myprog/DB/-OP/TR
>TKB myprog,,myprog=myprog/MP
>RUN myprog
```

The TKB command in the preceding example contains two commas between the file names on the left side of the equal sign because one of the TKB command parameters has been omitted. This command specifies that TKB create only an object file and a symbol table file. You must create a symbol table file for the debugger to have its full symbolic capability.

FOR DCL USERS

```
$ FORTRAN/F77/DEBUG/NOOP/TRACEBACK/LIST myprog
$ 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
```

Also, RSTS users should substitute "LB:" for "LB:[1.1].

VMS users must use the MCR commands; that is, they must insert "MCR" in front of "F77" in the compile command and in front "TKB" in the task build command. For example:

```
$ MCR F77 myprog,myprog/-SP=myprog/DB/-OP/TR
$ MCR TKB myprog,,myprog=myprog/MP
```

1.1.2 Invoking the Nonoverlaid Configuration

When you want to invoke the debugger with the nonoverlaid kernel, you must compile, link, and run your program. When you compile 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. The following MCR and DCL commands are used to invoke the debugger.

FOR MCR USERS

```
>F77 myprog,myprog/-SP=myprog/DB/-OP/TR
>TKB myprog,,myprog=myprog,LB:[1,1]PDPDBG/DA,LB:[1,1]F4POTS/LB
>RUN myprog
```

The TKB command in the preceding example contains two commas between the file names on the left side of the equal sign because one of the TKB command parameters has been omitted. This command specifies that TKB create only an object file and a symbol table file. You must create a symbol table file for the debugger to have its full symbolic capability.

FOR DCL USERS

```
$ FORTRAN/F77/DEBUG/NOOP/TRACEBACK/LIST myprog
$ LINK/DEBUG=LB:[1,1]PDPDBG/SYMBOL myprog,LB:[1,1]F4POTS/LIBR
$ 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].

VMS users must use the MCR commands; that is, they must insert "MCR" in front of "F77" in the compile command and in front "TKB" in the task build command. For example:

```
$ MCR F77 myprog,myprog/-SP=myprog/DB/-OP/TR
$ MCR TKB myprog,,myprog=myprog,LB:[1,1]PDPDBG/DA,LB:[1,1]F4POTS/LB
```

1.1.3 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 has important features that are available for all debugger users.

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

In addition, the debugger supports the following features for FORTRAN-77 users:

- Configures the debugger for FORTRAN-77 users using the SET LANGUAGE command (see Chapter 2)
- Recognizes FORTRAN-77 data types (see Chapter 6)
- Supports two new qualifiers, /I_SPACE and /D_SPACE, for use with the EXAMINE and DEPOSIT commands (see Chapter 6)

Chapter 2

Controlling Debugger Input and Output

This chapter explains how to set the default programming language to FORTRAN-77 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 FORTRAN-77.

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 FORTRAN, issue the following command:

```
DBG>SET LANGUAGE FORTRAN
```

This command informs the debugger that your program is written in FORTRAN-77. The debugger uses this information to control how it interprets and displays information.

When the language is set to FORTRAN, the debugger does the following:

- Interprets input and displays output in decimal integer format by default
- Recognizes the FORTRAN logical operators .GT., .LT., .GE., .LE., .EQ., .NE., .NOT., .AND., .OR., .XOR., and .EQV.
- Displays structured variables in column major order; if you examine a structured variable, the first item displayed is the item in Row 1, Column 1; the second item is the item in Row 2, Column 1; the third item is in Row 3, Column 1; and so on

2.2 Changing the Default Output

By default, the debugger's output configuration is NOLOG, TERMINAL, NOVERIFY. 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 displays at your terminal.

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

2.3 Using Log Files

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

```
SET LOG filespec
```

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

2.3.1 Log File Example

The following is an example of a log file.

```
SHOW OUTPUT
!%DEBUG-I-OUTPUT: noverify, terminal, logging to "SY:[33,52]MYPROG.LOG;1"
SET LANGUAGE FORTRAN
SHOW LANGUAGE
!%DEBUG-I-CURRLANGF77, Current language is FORTRAN-77
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 (!).

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

3.1.2 Program Symbols

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

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

- Names of user written routines
- ENTRY statement names
- Routine entry point labels
- Variable names (but not routine parameter names)
- Source code line numbers

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

NOTE

If you do not name the main module of your program using a FORTRAN program statement, the debugger assigns the symbol .MAIN. to it. Therefore, you must use this symbol to refer to the main module of your program if it is unnamed. Also, the debugger always displays .MAIN. as the name of an unnamed module.

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. The following restrictions apply to a debugger symbol name:

- It may be composed of only alphanumeric characters (the letters A to 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.

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

3.2 Making Symbols Unique

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

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

If you want to refer to symbols that are in a scope other than the default scope, you can specify a different scope in one of three ways:

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

These three methods of specifying scope are discussed in the following sections.

3.2.1 Simple Pathnames

A pathname describes a program location. It consists of program location labels separated by the backslash character (\). A program location label may be the name of a routine, a line number, an array reference, or a symbol. Valid formats for a nonoverlaid program are:

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

The pathname element routine is the name of the routine in which the symbol occurs. You must specify this pathname element 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 assumes that the routine name is a variable and looks in the current scope for that variable.

The element %LINE nnn specifies a line number in the routine with nnn representing the decimal integer number of the line, and the pathname element symbol denotes a program symbol or a symbol you defined previously for use in the routine.

The subscript-list is used when the symbol refers to an array, and you want to specify only one element of that array. Subscript-lists can be expressions, but all integers in them are interpreted in decimal radix, regardless of the default radix mode.

3.2.2 Pathnames in Overlaid Programs

The pathname syntax for overlaid programs is an extended form of the pathname syntax for nonoverlaid programs. 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)]]
```

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

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

The keyword %SEGMENT is optional, but it must be specified when you reference one of two or more segments in your overlay structure that have the same name.

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

3.2.3 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.

The SET SCOPE command has the following format:

```
SET SCOPE pathname
```

3.2.3.1 SET SCOPE Command Parameters

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

A scope prefix may be thought of as a truncated pathname. It describes a location in terms of its overlay 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
```

Segment-list names the overlay segment that contains the routine to which you are referring. Routine names the routine to which you are setting the scope. Note that in this case you need not specify the routine twice because the command SET SCOPE tells the debugger to expect a routine name.

Instead of naming the routine to which you want to set the scope, you can use one of two symbols, the number 0 or the backslash (\). The 0 symbol specifies that the scope be reset to the default, which is the PC scope. In other words, after issuing the command SET SCOPE 0, the scope is dynamic and is always the routine you are debugging. The backslash symbol specifies that symbols referenced without pathnames be interpreted as global symbols.

3.2.3.2 The SHOW SCOPE and CANCEL SCOPE Command

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

To determine the current scope, use the command:

```
DBG>SHOW SCOPE
```

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

```
DBG>CANCEL SCOPE
```

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

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. For each call, the debugger displays one line of information. The first line displays information about the current routine; the next line (if any) 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.

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 SHOW CALLS command has the following format:

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

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

4.2 The Effects of Breakpoints and Tracepoints

Once you decide where the important points in your program are, you are ready to set either breakpoints or tracepoints. This section describes the effects of these 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 4.2.2.2). 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 4.2.2.3).
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 4.2.2.2). 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 4.2.2.3).
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
           /RETURN
```

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
           /RETURN
```

4.2.1 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.

4.2.1.1 The /AFTER:n Qualifier

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

A special case exists. The /AFTER:0 qualifier has the same effect as /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, you cannot specify any other qualifier in the command.

4.2.1.3 The /RETURN Qualifier

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

4.2.2 SET BREAK and SET TRACE Parameters

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

4.2.2.1 The Address Parameter

The address parameter specifies the instruction address where you want 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 based on a condition specified by the value-expr parameter. For example, if you want a breakpoint to be activated only when a variable in your program called ADD is equal to 1, issue the following command:

```
DBG>SET BREAK %LINE 5 WHEN(ADD.EQ.1)
```

This command causes the debugger to test the contents of the variable ADD before it activates the breakpoint. If ADD is not equal to 1, the breakpoint is not activated.

4.2.2.3 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.

4.2.3 Commands Related to SET BREAK and SET TRACE

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

To see 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 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
```

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

To prevent breakpoints 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 cr the ENABLE TRACE command. You do not have to re-specify breakpoints or tracepoints when you use these commands.

Chapter 5

Starting the Program

When you are ready to execute your FORTRAN-77 program, use either the STEP command or the GO command. This chapter explains how to use these commands.

5.1 Executing a Specified Number of Commands

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

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

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

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

The 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. Lines in called routines are not counted to satisfy a step-count when the /OVER qualifier is in effect.

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

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

5.1.2 STEP Command Parameter

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

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

5.2 Changing the Default Step Conditions

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

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

The SET STEP command has the following format:

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

5.2.1 SET STEP Command Parameters

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

5.2.2 The SHOW STEP and CANCEL STEP Commands

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

```
SHOW STEP
```

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

```
GO [address]
```

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

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 types of program symbols and memory addresses are assigned by the compiler. 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 literal.

- 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.

If a program symbol or memory address is not assigned a data type by the compiler, the debugger uses a default data type to interpret location. The default data type for FORTRAN-77 is word integer. You can change the default data type with the SET TYPE command. This command has the following format:

```
SET TYPE datatype
             ASCII[:n]
             BYTE
             D_FLOAT
             FLOAT
             LONG
             INSTRUCTION
             RAD50
             WORD
```

Note that, in most cases, you need not be concerned about the default debugger data type because the compiler assigns data types to your program locations. Therefore, it is unlikely that the debugger will use its default data type to interpret any of your program locations. Table 6–1 shows how the debugger interprets the data types used by the FORTRAN-77 compiler.

Table 6–1: FORTRAN-77 and DEBUG Data Types

FORTRAN-77 Data Type	Equivalent DEBUG Data Type
BYTE	BYTE
LOGICAL	WORD
LOGICAL*1	BYTE
LOGICAL*2	WORD
LOGICAL*4	LONG
INTEGER	WORD
INTEGER*2	WORD
INTEGER*4	LONG
REAL	FLOAT
REAL*4	FLOAT
REAL*8	D_FLOAT
DOUBLE PRECISION	D_FLOAT
COMPLEX	Use two FLOAT numbers
COMPLEX*8	Use two FLOAT numbers

Table 6–1 (Cont.): FORTRAN-77 and DEBUG Data Types

FORTRAN-77 Data Type	Equivalent DEBUG Data Type
CHARACTER*len[1]	ASCII[:n][1]
Radix–50	RAD50
MACRO–11 Instruction	INSTRUCTION

[1]Len and n both represent the same thing: the number of characters specified. This number can be any integer from 1 through 255.

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 FORTRAN-77 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 following format:

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

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

[NO]SYMBOL determines whether symbols, such as 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.

Two new qualifiers, /I_SPACE and /D_SPACE, have been defined for use by FORTRAN-77 users who have I- and D-space support on their systems. Consult your system manager to determine if your operating system provides I- and D-space support.

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, or numerical base, the debugger uses to interpret the expressions you specify. The data-type qualifiers control how the debugger interprets the value you specify.

Note that although these qualifiers allow you to deposit values in a data type other than the one associated with the memory location to which you are depositing, they do not alter the data type of the location. Therefore, your program treats the value as if it were of the data type associated with the location, and the debugger still uses this data type to control its interpretation of the location if you examine its contents later in the debugging session. The debugger can be instructed to treat the location in the data type of the value you deposited if you use a data type qualifier with the EXAMINE command.

FORTRAN-77 users who have I- and D-space support on their systems can use the new /I_SPACE and /D_SPACE qualifiers. (See your system manager to find out if your operating system provides I- and D-space support.) 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. 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.

Chapter 7

Example Debugging Session

This chapter contains sample debugging sessions that demonstrate the most commonly used debugger commands. The program being debugged consists of a main program and three subroutines. The example sessions find errors in the subroutines. In each example, a compiler listing of each subroutine is given, followed by the debugging session for that subroutine. Comments are written beside each debugger command in the example and further explanation follows each example.

The main procedure of this program, which is called MAIN, is a series of calls to other routines. This routine executes correctly; the bugs in this program are in its subroutines. The following is a compiler listing of the main routine:

```
C PERFORM SOME SIMPLE STATISTICS ON A SET OF NUMBERS
      PROGRAM MAIN

      PARAMETER (ICOUNT=10)
      REAL MEDIAN
      DIMENSION VECTOR(ICOUNT)
      DATA VECTOR/10.,12.,8.,14.,6.,16.,4.,18.,2.,20/

C  COMPUTE THE AVERAGE
      PRINT *,AVERGE(VECTOR,ICOUNT)

C  SORT THE VALUES INTO ASCENDING ORDER
      CALL SORT(VECTOR,ICOUNT)

C  COMPUTE THE MEDIAN
      PRINT *,MEDIAN(VECTOR,ICOUNT)

C  COMPUTE THE RANGE
      PRINT *,VECTOR(ICOUNT)-VECTOR(1)

      STOP
      END
```

7.1 Debugging the Function AVERGE

The first routine to be debugged is a FUNCTION in the program that computes the average of a number of values. A listing of the source code follows:

```
        C  COMPUTE THE AVERAGE OF THE VALUES IN AN ARRAY

0001             FUNCTION AVERGE(VALUES,MAXDIM)
0002             DIMENSION VALUES (MAXDIM)

0003             AVG = 0.0

0004             DO 10 INDEX = 1, MAXDIM
0005    10             AVG = AVG + VALUES(INDX)

0006             AVERGE = AVG/REAL(MAXDIM)

0007             RETURN
0008             END
```

The value of the variable AVG is not correct after AVERGE is executed. The following is a debugging session that finds the error in AVERGE. The comments given beside each debugger command briefly explain the function of the command. These comments are not generated by the PDP-11 Symbolic Debugger; they have been added to clarify the example. The callout numbers beside each comment correspond to the numbered explanations that follow.

```
DBG>SET BREAK AVERGE\%LINE 3          !Set to break before DO loop        ❶
DBG>GO                                !Begin execution at strat of program ❷
%DEBUG-I-START, routine start at MAIN
%DEBUG-I-BREAKPOINT, breakpoint at AVERGE\%LINE 3
DBG>EXAMINE/INSTRUCTION %LINE 4:%LINE 6 !Look for branching instruction  ❸
AVERGE\%LINE 4: MOV     @4(%R5),1512)
AVERGE\%LINE 4 +6:      MOV     #1,AVERGE\INDEX
AVERGE\%LINE 4 +12:     CMP     AVERGE\INDEX,1512
AVERGE\%LINE 4 +18:     BGT     AVERGE\%LINE 6
AVERGE\%LINE 5: MOV     #65531,1612
AVERGE\%LINE 5 +6:      MOV     AVERGE\INDX,%RO
AVERGE\%LINE 5 +10:     ASL     %RO
AVERGE\%LINE 5 +12:     ASL     %RO
AVERGE\%LINE 5 +14:     ADD     1460,%RO
AVERGE\%LINE 5 +18:     SETF
AVERGE\%LINE 5 +20:     LDF     AVERGE\AVG,%FO
AVERGE\%LINE 5 +24:     ADDF    (%RO),%FO
AVERGE\%LINE 5 +26:     STF     %FO,AVERGE\AVG
AVERGE\%LINE 5 +30:     INC     AVERGE\INDEX
AVERGE\%LINE 5 +34:     CMP     AVERGE\INDEX,1512
AVERGE\%LINE 5 +40:     BLE     AVERGE\%LINE 5
AVERGE\%LINE 6: MOV     #65530,1612
DBG>SET BREAK %LINE 5+40              !Set to break at branching instruction❹
DBG>GO                                !Resume execution                  ❺
%DEBUG-I-START, start at AVERGE\%LINE 3
%DEBUG-I-BREAKPOINT, breakpoint at AVERGE\%LINE 5+40
DBG>EXAMINE AVG                       !Does AVG contain the 1st array value (10)?❻
AVERGE\AVG:     0.0000000
DBG>SET SCOPE MAIN                    !Change scope to routine containing array VECTOR❼
DBG>EXAMINE VECTOR(1):VECTOR(3)       !Verify initialization of VECTOR   ❽
MAIN\VECTOR(1): 10.00000
MAIN\VECTOR(2): 12.00000
MAIN\VECTOR(3): 8.000000
DBG>SET SCOPE 0                       !Return scope to active routine    ❾
DBG>EXAMINE/INSTRUCTION %LINE 5:%LINE 6 !Because VECTOR is correct, error must be in assignment
                                        !of AVG.  Look at range of intructions to find error.❿
AVERGE\%LINE 5: MOV      #65531,1612
AVERGE\%LINE 5 +6:       MOV     AVERGE\INDX,%RO  !INDEX is misspelled.       ⓫
AVERGE\%LINE 5 +10:      ASL     %RO
AVERGE\%LINE 5 +12:      ASL     %RO
AVERGE\%LINE 5 +14:      ADD     1460,%RO
AVERGE\%LINE 5 +18:      SETF
AVERGE\%LINE 5 +20:      LDF     AVERGE\AVG,%FO
AVERGE\%LINE 5 +24:      ADDF    (%RO),%FO
AVERGE\%LINE 5 +26:      STF     %FO,AVERGE\AVG
AVERGE\LINE 5 +30:       INC     AVERGE\INDEX
AVERGE\%LINE 5 +34:      CMP     AVERGE\INDEX,1512
AVERGE\%LINE 5 +40:      BLE     AVERGE\%LINE 5
AVERGE\%LINE 6: MOV      #65530,1612

DBG>DEPOSIT INDX = 1                   !Assign beginning value to INDX        ⓬
DBG>GO                                 !Execute loop                          ⓭
%DEBUG-I-START, start at AVERGE\%LINE 5+40
%DEBUG-I-BREAKPOINT, breakpoint at AVERGE\%LINE 5+40
DBG>EXAMINE AVG                        !Does AVG contain the 1st array value?⓮
AVERGE\AVG:    10.00000
DBG>DEPOSIT INDEX=2                    !Reset loop controller                 ⓯
DBG>DEPOSIT INDX=2                     !Increment INDX                        ⓰
DBG>SET TRACE %LINE 5+30 DO (DEPOSIT INDX=INDX+1) !Set tracepoint to increment INDX.⓱
DBG>CANCEL BREAK AVERGE\%LINE 5+40     !Cancel breakpoint in loop             ⓲
DBG>SET BREAK MAIN\%line 7             !Set to break after routine is finished⓳
DBG>GO                                 !Complete routine execution            ⓴
%DEBUG-I-START, start at AVERGE\%LINE 5 +40
%DEBUG-I-TRACEPOINT, tracepoint at AVERGE\%LINE 5+30
%DEBUG-I-TRACEPOINT, tracepoint at AVERGE\%LINE 5+30
%DEBUG-I-TRACEPOINT, tracepoint at AVERGE\%LINE 5+30
%DEBUG-I-TRACEPOINT, tracepoint at AVERGE\%LINE 5+30
%DEBUG-I-TRACEPOINT, tracepoint at AVERGE\%LINE 5+30
%DEBUG-I-TRACEPOINT, tracepoint at AVERGE\%LINE 5+30
%DEBUG-I-TRACEPOINT, tracepoint at AVERGE\%LINE 5+30
%DEBUG-I-TRACEPOINT, tracepoint at AVERGE\%LINE 5+30
%DEBUG-I-TRACEPOINT, tracepoint at AVERGE\%LINE 5+30
11.00000
%DEBUG-I-BREAKPOINT, breakpoint at MAIN\%LINE 7
```

❶ The first command sets a breakpoint at the instruction before the loop to be debugged.

❷ The GO command begins program execution, which continues until the breakpoint is reached.

❸ Once this instruction is reached, a range of instructions is examined to determine the location of the branching instruction that controls the loop.

❹ Now a breakpoint is set at the address of the instruction that controls the looping process. The debugger stops the program immediately before this instruction is executed.

❺ The GO command resumes program execution.

❻ When program execution is halted, the value of AVG is examined to determine if its value is being assigned correctly.

❼ Because the value of AVG is not correct, the array that contains the values to be assigned to AVG may have been initialized incorrectly. The scope is set to the main routine so that the array can be displayed.

❽ The EXAMINE command verifies the initialization of the array VECTOR. Note that here a range of values is examined.

❾ Because this initialization is correct, the scope is set to the currently active program unit (AVERGE) with the command SET SCOPE 0 and debugging continues in this function.

❿ The error must be in the instruction:

```
AVG = AVG + VALUES(INDEX)
```

Therefore, the PDP–11 machine instructions to which this source code line translates are displayed.

⓫ The error is found in AVERGE\%LINE 5 + 6. The loop control variable, INDEX, is misspelled as INDX.

⓬ To ensure that this error is the one that is causing the wrong values to be assigned to the variable AVG, the value 1 is deposited in INDX. When the loop is executed, this assignment causes it to behave as if it is executing for the first time.

⓭ The loop is executed again with the GO command.

⓮ The variable AVG is examined and found to contain the correct value 10.00000.

⓯ Therefore, to make the loop work correctly, the variable INDEX is given the value 2. This assignment allows the loop to be executed the correct number of times.

⓰ The misspelled variable INDX is also given the value 2.

⓱ This misspelled variable INDX is incremented during loop execution by setting a tracepoint at the PDP–11 machine instruction that increments the loop controller INDEX. The DO portion of this tracepoint increments INDX thus forcing the correct values to be assigned to the variable AVG.

⓲ Because the error has been found in this loop, the breakpoint is canceled.

⑲ A breakpoint is set at the line in the main routine that calls the next routine. This breakpoint halts execution immediately before the calling instruction is executed.

⑳ The GO command resumes program execution. Note that after the debugger issues 9 informational messages about the activation of the tracepoint in the function, the value of the function is printed on the terminal screen by the program. This value is not displayed by the debugger; it is displayed by the user program. The value the program displays is correct, so this routine is known to be operating correctly.

7.2 Debugging the Subroutine SORT

The second routine to be debugged is a subroutine in the program that sorts the numbers in the array VECTOR into ascending order. A listing of the source code follows:

```
C  SORT THE ELEMENTS OF THE ARRAY 'VALUES' INTO ASCENDING ORDER

0001           SUBROUTINE SORT (VALUES,MAXDIM)
0002           DIMENSION VALUES(MAXDIM)
0003           LOGICAL DONE

0004   10      DONE = .TRUE.
0005           DO 20 INDEX = 1, MAXDIM-1
0006               IF (VALUES(INDEX) .GT. VALUES(INDEX+1)) THEN
0007                   VALUES(INDEX) = VALUES(INDEX+1)
0008                   SWITCH = VALUES(INDEX)
0009                   VALUES(INDEX+1) = SWITCH
0010                   DONE = .FALSE.
0011               ENDIF
0012   20      CONTINUE

0013           IF (.NOT. DONE) GOTO 10

0014           RETURN
0015           END
```

The value of the array VECTOR is not correct after this routine is executed. The following example debugging session finds the error that causes the incorrect results. The comments given beside each debugger command briefly explain the function of the command. These comments are not generated by the PDP–11 Symbolic Debugger; they have been added to clarify the example. The callout numbers beside each comment correspond to the numbered explanations that follow.

```
DBG>SET BREAK/AFTER:3 SORT\%LINE 6        !Set to break on 3rd encounter of Line 6❶
DBG>GO                                    !Resume execution                      ❷
%DEBUG-I-START, routine start at MAIN
%DEBUG-I-BREAKPOINT, breakpoint at SORT\%LINE 6
DBG>EXAMINE INDEX                         !Check loop control variable           ❸
DBG>SORT\INDEX:    3
DBG>EXAMINE SWITCH                        !Check temporary storage variable      ❹
DBG>SORT\SWITCH:        8.000000
DBG>CANCEL BREAK SORT\%LINE 6             !Source code lines in wrong order
                                          !Cancel breakpoint to continue program execution❺
DBG>SET BREAK MAIN\%line 8                !Set to break before call to next function❻
DBG>GO                                    !Resume execution                      ❼
%DEBUG-I-START, routine start at SORT\%LINE 6
%DEBUG-I-BREAKPOINT, breakpoint at MAIN\%LINE 8
```

As mentioned previously, the subroutine SORT arranges the values in the array VECTOR in ascending order. The values that SORT is working on, in order, follow:

```
10.,12.,8.,14.,6.,16.,4.,18.,2.,20
```

The following list explains the debugger commands used to find the error that causes incorrect results to be returned from SORT.

❶ The first command sets a breakpoint at Line 6 in the subroutine. The /AFTER:3 qualifier is used because the first time the variables in the subroutine contain meaningful information is when this instruction is encountered the third time.

❷ The GO command begins execution of the subroutine.

❸ The loop control variable, INDEX, is examined to determine if this variable is being incremented correctly. Because the variable contains the number 3, it is known to be working correctly.

❹ The variable SWITCH is then examined. Note that at this point in the program, the variable contains the value that it was assigned during the previous execution of the loop. Because SWITCH is used to store the higher of two values that must be reversed in the array VECTOR, the variable should contain the number 12. This is determined by checking the contents of VECTOR and noting that the numbers 12 and 8 must be reversed during the second execution of the loop. This variable contains the incorrect value.

❺ When it is determined that SWITCH is being assigned the wrong value, the source code is examined to see how a value is stored in SWITCH. It is noted that two of the source code lines, Lines 7 and 8, are in the wrong order.

Because there is not a way to reverse source code lines using the debugger, this routine cannot be fixed for purposes of testing to see if the error found is the only one that exsits, so the breakpoint in the routine is canceled to allow it to complete execution.

❻ Another breakpoint is set in the main routine that stops program execution before the call to the next functions.

❼ The GO command resumes program execution.

7.3 Testing the Function MEDIAN

Because the rest of this program works on a sorted array and because the routine in this program that sorts an array is incorrect, it must be determined if the incorrect values the rest of the program generates are caused by incorrect sorting or incorrect coding. Therefore, the function MEDIAN is tested.

The following is a listing of the source code in this function:

```
C  THIS FUNCTION COMPUTES THE MEDIAN OF THE ARRAY 'VALUES'

0001          REAL FUNCTION MEDIAN(VALUES,MAXDIM)
0002          DIMENSION VALUES(MAXDIM)

C  COMPUTE THE MEDAN
0003          IF (MOD(MAXDIM,2) .EQ. 0) THEN
0004             MEDIAN=(VALUES(MAXDIM/2+1)+VALUES(MAXDIM/2))/2
0005          ELSE
0006             MEDIAN=VALUES(MAXDIM/2)
0007          ENDIF

0008          RETURN
0009          END
```

The following is a list of the commands and debugger responses issued during the testing of the function MEDIAN. The comments given beside each debugger command briefly explain the function of the command. These comments are not generated by the PDP–11 Symbolic Debugger; they have been added to clarify the example. The callout numbers beside each comment correspond to the numbered explanations that follow.

```
DBG>DEPOSIT VECTOR(1)=2.,4.,6.,8.,10.,12.,14.,16.,18.,20.  !Deposit sorted values in VECTOR❶
DBG>EXAMINE VECTOR(1):VECTOR(10)                           !Verify deposit     ❷
MAIN\VECTOR(1): 2.000000
MAIN\VECTOR(2): 4.000000
MAIN\VECTOR(3): 6.000000
MAIN\VECTOR(4): 8.000000
MAIN\VECTOR(5): 10.00000
MAIN\VECTOR(6): 12.00000
MAIN\VECTOR(7): 14.00000
MAIN\VECTOR(8): 16.00000
MAIN\VECTOR(9): 18.00000
MAIN\VECTOR(10):        20.00000
DBG>SET STEP OVER                                   !Set to execute routine with
                                                    !one step command        ❸
DBG>STEP                                            !Step to next line in main routine❹
%DEBUG-I-START, start at MAIN\%LINE 8
11.00000
%DEBUG-I-START, stepped to MAIN\%LINE 9
DBG>STEP                                            !Test computation of range❺
%DEBUG-I-START, start at MAIN\%LINE 9
18.00000
%DEBUG-I-START, stepped to MAIN\%LINE 10
DBG>EXIT                                            !Leave the debugger      ❻
```

❶ Because the values in the array VECTOR are incorrect, the correct values are assigned to this array using the DEPOSIT command.

❷ The results of the DEPOSIT command are verified with the EXAMINE command.

❸ Because it has not been determined if the function MEDIAN works correctly and the function is not debugged, but only tested. Therefore, the step conditions are set to OVER, so the routine can be executed using the STEP command.

❹ The STEP command executes MEDIAN, and steps to the next line in the main routine. The value the program displays on the terminal (11.0000) is correct for the value of the median of this array, so this function is known to be working correctly.

❺ The STEP command is issued again, so the last command in the program, which computes the range of the array, can be tested. The program displays the correct value (18.00000).

❻ Now that the program has been fully debugged and tested, the debugger is exited using the command EXIT.

INDEX

PDP–11 Symbolic Debugger
FORTRAN–77 User's Guide
AA–FA64A–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