

digital

**VAX-11 PL/I**  
**Guide to Program Debugging**

Order No. AA-K221A-TE

**VAX11**

**August 1980**

Describes the operation of the VAX-11 Symbolic Debugger  
with VAX-11 PL/I programs.

**VAX-11 PL/I**  
**Guide to Program Debugging**

Order No. AA-K221A-TE

**SUPERSESSION/UPDATE INFORMATION:** This is a new document for this  
release.

**OPERATING SYSTEM AND VERSION:** VAX/VMS V2.0

**SOFTWARE VERSION:** VAX-11 PL/I V1.0

To order additional copies of this document, contact the Software Distribution  
Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

**digital equipment corporation · maynard, massachusetts**

First Printing, August 1980

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

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

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

Copyright © 1980 by Digital Equipment Corporation

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECTape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	PDT
DATATRIEVE	TRAX	

ZKA73-80

## CONTENTS

	Page
PREFACE	v
CHAPTER 1	INTRODUCTION TO DEBUGGING VAX-11 PL/I PROGRAMS
1.1	VAX-11 SYMBOLIC DEBUGGER FACILITIES . . . . . 1-1
1.2	USING THE VAX-11 DEBUGGER . . . . . 1-2
1.2.1	Beginning and Ending a Debugging Session . . . . . 1-2
1.2.2	The DEBUG Command . . . . . 1-3
1.2.3	Effects of Optimization on Debugging . . . . . 1-3
1.3	DEBUGGER COMMAND SYNTAX AND SUMMARY . . . . . 1-3
1.4	SAMPLE TERMINAL SESSION . . . . . 1-9
1.4.1	Executing the Sample Program . . . . . 1-10
CHAPTER 2	RECOGNITION OF NAMES
2.1	DEBUGGER SYMBOL TABLE . . . . . 2-1
2.1.1	Names Included in the Symbol Table by Default . . . . . 2-1
2.1.2	Adding Names to the Symbol Table . . . . . 2-2
2.1.3	Displaying Names in the Symbol Table . . . . . 2-2
2.2	SPECIFYING REFERENCES AND LOCATIONS . . . . . 2-3
2.2.1	Specifying Internal and External Variables . . . . . 2-3
2.2.2	References to Global Symbols . . . . . 2-4
2.2.3	Specifying Program Locations . . . . . 2-4
2.2.4	Defining Addresses Symbolically . . . . . 2-5
2.2.5	The Debugger's Permanent Symbols . . . . . 2-5
2.3	SCOPE . . . . . 2-5
2.3.1	Specifying Pathnames . . . . . 2-6
2.3.2	Changing the Scope . . . . . 2-7
2.3.3	The Scope of Automatic Variables . . . . . 2-8
2.4	SPECIAL CHARACTERS AND EXPRESSIONS . . . . . 2-8
2.4.1	Characters for Arithmetic Expressions . . . . . 2-8
2.4.2	Characters for the Current, Previous, and Next Locations . . . . . 2-8
CHAPTER 3	EXAMINING AND DEPOSITING DATA
3.1	USING THE EXAMINE AND DEPOSIT COMMANDS . . . . . 3-1
3.1.1	Specifying the Data Type of Data to Deposit . . . . . 3-1
3.1.2	Restrictions on Examining and Depositing Data . . . . . 3-2
3.2	FIXED-POINT BINARY AND FLOATING-POINT VARIABLES . . . . . 3-4
3.3	FIXED-POINT DECIMAL DATA . . . . . 3-4
3.4	CHARACTER-STRING VARIABLES . . . . . 3-4
3.5	BIT-STRING VARIABLES . . . . . 3-5
3.6	STATIC ARRAYS . . . . . 3-5
3.7	AUTOMATIC ARRAYS AND FIXED-POINT DECIMAL ARRAYS . . . . . 3-6

## CONTENTS

	Page
CHAPTER 4	CONTROLLING A PROGRAM'S EXECUTION
4.1	STARTING AND STOPPING EXECUTION . . . . . 4-1
4.2	STEPPING THROUGH A PROGRAM . . . . . 4-2
4.3	BREAKPOINTS . . . . . 4-3
4.4	TRACEPOINTS . . . . . 4-4
4.5	WATCHPOINTS . . . . . 4-5
4.6	ENTERING AND RETURNING FROM SUBROUTINES . . . . . 4-6
4.6.1	Stepping Into and Over Subroutines . . . . . 4-6
4.6.2	Displaying the Calling Sequence . . . . . 4-7
4.6.3	Calling Subroutines . . . . . 4-7
APPENDIX A	VAX-11 PL/I RUN-TIME MODULES AND ENTRY POINTS

INDEX

## FIGURE

Figure 3-1	The Sample Program, MAINP . . . . . 3-3
------------	---

## TABLES

Table 1-1	Summary of Debug Commands . . . . . 1-3
Table 2-1	Arithmetic Operators . . . . . 2-8
Table A-1	VAX-11 PL/I Run-Time Modules . . . . . A-1
A-2	Run-Time Entry Points . . . . . A-3
A-3	Run-Time Library Procedures Called by PL/I . . . . . A-12

## PREFACE

### MANUAL OBJECTIVES

This manual describes the facilities of the VAX-11 Symbolic Debugger for debugging VAX-11 PL/I programs.

### INTENDED AUDIENCE

This manual is intended for programmers using VAX-11 PL/I. To get the most out of this manual, you should have a working knowledge of PL/I program structure and data types, and be familiar with the VAX/VMS operating system. However, while not a tutorial, the manual can be used by relatively inexperienced programmers.

### STRUCTURE OF THIS DOCUMENT

This manual has four chapters and one appendix:

- Chapter 1, "Introduction to Debugging VAX-11 PL/I Programs," provides a functional overview of debugging PL/I programs using the VAX-11 Symbolic Debugger.
- Chapter 2, "Recognition of Names," describes how the debugger recognizes program locations, for example, line numbers and procedure names, that you specify.
- Chapter 3, "Examining and Depositing Data," explains how to examine variables and program locations and to modify their contents while you are debugging a program.
- Chapter 4, "Controlling a Program's Execution," describes how to start, stop, and control a program while you are running it under the control of the debugger.
- Appendix A, "VAX-11 PL/I Run-Time Modules and Entry Points," lists the VAX-11 PL/I run-time modules and entry points.

### ASSOCIATED DOCUMENTS

To obtain supplemental information, the following documents are recommended:

## PREFACE

- VAX-11 Symbolic Debugger Reference Manual, Order Number AA-D026B-TE
- VAX/VMS Command Language User's Guide, Order Number AA-D023B-TE
- VAX-11 PL/I Encyclopedic Reference, Order Number AA-H952A-TE
- VAX-11 PL/I User's Guide, Order Number AA-H951A-TE

## CONVENTIONS USED IN THIS DOCUMENT

EXAMINE reference	Uppercase words and letters, shown in syntax descriptions, indicate that you should type the word or letter exactly as shown.  Lowercase words and letters indicate that you are to substitute a word or value of your choice.
<code>CTRLX</code>	The symbol <code>CTRLX</code> indicates that you press the key "x" while holding down the key labeled CTRL, for example, <code>CTRLC</code> . In examples, this control key sequence is shown as ^x, for example, ^C, because that is how the VAX/VMS system prints control key sequences.
DBG>EXAMINE X ALPHA\X: 2	Command examples show all interactive examples in two colors. Program output and prompting characters that the system prints or displays are shown in black letters. User-entered commands and data are shown in red letters.
option,...	Horizontal ellipses indicate that additional parameters, options, or values can be entered. When a comma precedes the ellipses, it indicates that successive items must be separated by commas.
DBG> EVALUATE X(1):X(10) . .	Vertical ellipses indicate that not all of the text of a program or program output is illustrated. Only relevant material is shown in the example.
quotation mark apostrophe	The term "quotation mark" is used to refer to the quotation mark (") symbol. The term "apostrophe" is used to refer to the single quotation mark (') symbol.
[/qualifier...]	Square brackets indicate that a syntactic element is optional and you need not specify it. Square brackets are not optional, however, when used to delimit a directory name in a VAX/VMS file specification.

## PREFACE

[ INTO ]  
[ OVER ]


Brackets surrounding two or more stacked items indicate a choice of optional data; you may choose one of the two syntactic elements.

{ /ALL  
 module-name }

Braces surrounding two or more stacked items indicate a choice; you must choose one of the two syntactic elements.

DBG>DEPOSIT X = 2

All numeric values in the text of this manual are represented in decimal notation unless otherwise specified.

Unless otherwise specified, you terminate commands by pressing the RETURN key, shown in this document as  .





## CHAPTER 1

### INTRODUCTION TO DEBUGGING VAX-11 PL/I PROGRAMS

One of the most difficult stages in program development is locating and correcting errors. This is "debugging." You need to debug, that is, to correct a program, when any of the following happen:

- The compiler flags syntactic or lexical errors
- Run-time errors occur
- You determine, based on receiving incorrect output during a program's execution, that a logic error exists

The VAX-11 PL/I compiler and run-time system display error and informational messages when errors occur. You can use this information to determine where the error exists in your program and to correct it.

You must detect logic and programming errors yourself. To help you find such errors, VAX/VMS provides a special program: the Symbolic Debugger (or, simply, the debugger). The debugger lets you control the execution of your program so you can monitor specific locations, change the contents of locations, check the sequence of program control, and otherwise locate and correct errors as they occur. After you track down the mistakes, you can edit your source program, recompile, relink, and execute the corrected version.

#### 1.1 VAX-11 SYMBOLIC DEBUGGER FACILITIES

The VAX-11 Symbolic Debugger includes many features to help you, among them the following:

- It is interactive. You control your program and interact with the debugger from your terminal.
- It understands static PL/I variable names and their data types. Thus, when you want to look at the contents of a variable, or change the value of a variable, the debugger will convert your ASCII text input to the data type of the variable.
- It understands other programming languages as well, such as FORTRAN and COBOL. Thus, if your programs consist of procedures written in different languages, you can change from one language to another during the course of a debugging session.

Note that for this version of the VAX-11 PL/I compiler, not all functions of the VAX-11 Symbolic Debugger are completely supported for PL/I program debugging. This manual describes the extent of support as it exists for Version 1.0 of VAX-11 PL/I and Version 2.0 of the VAX-11 Symbolic Debugger.

## 1.2 USING THE VAX-11 DEBUGGER

This section shows brief examples of invoking and using the debugger with a PL/I program.

### 1.2.1 Beginning and Ending a Debugging Session

To execute a PL/I program with the debugger, compile and link the program with the /DEBUG qualifier, as in the following example:

```
$ PLI/DEBUG METRIC
$ LINK/DEBUG METRIC
```

The /DEBUG qualifier on the PLI command requests the compiler to write symbol table records into the object module; these records will permit you to examine and modify variables by name during the debugging session.

The /DEBUG qualifier on the LINK command requests the linker to include the debugger routines, global symbols, and traceback information in the executable image. To include only traceback information, specify /DEBUG=TRACEBACK.

To obtain a program listing of the procedures being debugged, and to have available a storage map listing the variables, you can compile the procedure(s) with the /LIST and /ENABLE=LIST\_MAP qualifiers, in addition to the /DEBUG qualifier. For example:

```
$ PLI/DEBUG/LIST/ENABLE=LIST_MAP METRIC
```

If your program includes files using %INCLUDE statements, you may also want to include these files in the listing to have available the statement line numbers. The /ENABLE qualifier also enables listing INCLUDE files. To list the compiler map and INCLUDE files, specify:

```
$ PLI/DEBUG/LIST/ENABLE=(LIST_MAP, LIST_INCLUDE) METRIC
```

When you execute an image compiled and linked with the debugger, initial control goes to the debugger, which identifies itself as follows:

```
$ RUN METRIC
                                VAX-11 DEBUG Version 2.00

%DEBUG-I-INITIAL, language is BASIC, module set to 'CONVERT'
DBG>
```

For this version of the PL/I debugging support, the language is set to BASIC. The module name displayed in the debugger's message is the name of the outermost procedure in the first object module in the image and is not necessarily the same as the name of the image file. This message indicates that the name of the main procedure in the image file METRIC is CONVERT.

The DBG> prompt indicates that the debugger is now ready to process your commands. You respond to the prompt with one of the commands recognized by the debugger. To terminate the debugging session, use the EXIT command:

```
DBG>EXIT
```

When your program has been thoroughly debugged, you can recompile and relink it without the /DEBUG qualifier. Or, you can run it with the /NODEBUG qualifier. For example:

```
$ RUN/NODEBUG METRIC
```

Note, however, that the modules required by the debugger occupy space within a program image file.

### 1.2.2 The DEBUG Command

When a program that is linked with /DEBUG is executing, you can interrupt it with `CTRL/Y` at any time and invoke the debugger by entering the DEBUG command. For example, if you determine that a program may be looping, or if you see erroneous output, you can interrupt it as follows:

```
$ RUN COMPUTE CTRL/Y
^Y

$ DEBUG
DBG>
```

When you press `CTRL/Y`, the command interpreter displays its dollar sign (\$) prompt, and you can enter the DEBUG command. The DBG> prompt indicates that the debugger is under control.

If the program was compiled with the /DEBUG qualifier, you have access to program variables, line numbers, and entry names.

If the program was not compiled with the /DEBUG qualifier, you can reference program locations and variables using only virtual addresses.

### 1.2.3 Effects of Optimization on Debugging

When you compile a PL/I program, the resulting object code is optimized; that is, the compiler has used some techniques that will make the program run faster. For example, the compiler puts automatic scalar variables in registers, removes invariant expressions within DO-loops so that they are evaluated only once, and so on.

Under normal circumstances, you do not need to disable any compiler optimizations in order to debug a VAX-11 PL/I program. By default, the compiler disables the DISJOINT optimization option when /DEBUG is specified so that automatic variables that are placed in registers will be guaranteed to stay in the same register during the current block activation.

No other optimization options have any effect on debugging.

## 1.3 DEBUGGER COMMAND SYNTAX AND SUMMARY

You enter commands to the debugger in much the same way that you enter DCL commands. You must remember to end each debugging command with a `RET`. The debugger commands have the format:

```
cmd [keyword] [/qualifier] [param ...] !comment
```

#### cmd

Is a command verb (for example, SET, CANCEL) that indicates the general function to be performed.

#### keyword

Gives the specific function to be performed by the command (for example, CANCEL MODULE, SET SCOPE, SHOW LANGUAGE).

INTRODUCTION TO DEBUGGING VAX-11 PL/I PROGRAMS

**/qualifier**

Modifies the effect of the command.

**param**

Qualifies the function in some way, such as specifying a range of locations to be monitored.

**comment**

Is any text message. The debugger ignores all text after the exclamation mark.

You can enter more than one command on a command line by separating the commands with semicolons (;).

You can continue a command on a new line by ending the line with a hyphen (-); the debugger will then prompt for the rest of the command with an underscore (\_).

Table 1-1 summarizes the debugger commands. The boldface letters indicate the minimum abbreviation you must type in order for the debugger to recognize the command name, qualifier, or parameter.

You can obtain information about a debugging command while you are debugging by entering the HELP command to the debugger.

Table 1-1  
Summary of Debug Commands

Command Syntax	Function
@file-spec	Reads debugger commands from the specified command procedure file
CALL entry-name [(argument,...)]	Invokes a specified procedure and optionally passes references to arguments
<b>CANCEL ALL</b>	Cancels all breakpoints, tracepoints, and watchpoints, and restores the mode and scope to their original values
<b>CANCEL BREAK</b> { /ALL %LINE line-number entry-name symbolic-reference nonsymbolic-address }	Cancels a specified breakpoint or all breakpoints
<b>CANCEL EXCEPTION BREAK</b>	Cancels the effect of SET EXCEPTION BREAK and restores the debugger's default method for handling exceptions, which is to let the programs condition handlers, or ON-units, receive control

(Continued on next page)

INTRODUCTION TO DEBUGGING VAX-11 PL/I PROGRAMS

Table 1-1 (Cont.)  
Summary of Debug Commands

Command Syntax	Function
<b>CANCEL MODE</b>	Restores the radix and display modes to their defaults for PL/I debugging, which are decimal and symbolic
<b>CANCEL MODULE</b> { /ALL { module,... } }	Deletes one or more modules from the debugger's symbol table, or deletes all modules from the symbol table
<b>CANCEL SCOPE</b>	Resets the scope to that containing the current program counter
<b>CANCEL TRACE</b> { %LINE line-number entry-name symbolic-reference nonsymbolic-address /ALL /BRANCH /CALL }	Cancels a specified tracepoint or all tracepoints
<b>CANCEL TYPE/OVERRIDE</b>	Restores the debugger's default interpretation of variables, which is to use the variables' declared data types and extents
<b>CANCEL WATCH</b> { /ALL variable-reference symbolic-reference nonsymbolic-address }	Cancels a watchpoint on a specified location or variable or cancels all watchpoints
<b>DEFINE</b> symbol = expression ,...	Creates one or more symbols whose values are equated to program locations or to numeric expressions
<b>DEPOSIT</b> location = data [,data,...]  [ /ASCII:length ] [ /DECIMAL /BYTE          ] [ /HEXADECIMAL /INSTRUCTION ] [ /OCTAL /LONG /WORD          ]	Changes the contents of a specified variable or program location
<b>EVALUATE</b> [/ADDRESS] expression,...  [ /DECIMAL /HEXADECIMAL OCTAL          ]	Evaluates an expression or an address and displays the results in decimal or other specified radix

(Continued on next page)

Table 1-1 (Cont.)  
Summary of Debug Commands

Command Syntax	Function
<p><b>EXAMINE</b> { variable-reference }           { location[:location] }</p> <p>[ /ASCII:length ] [ /DECIMAL ] [ /BYTE ] [ /HEXADECIMAL ] [ /INSTRUCTION ] [ /OCTAL ] [ /LONG ] [ /WORD ]</p> <p>[ /SYMBOLIC ] [ NOSYMBOLIC ]</p> <p><b>EXIT</b></p> <p><b>GO</b> [ %LINE line-number ]       [ entry-name ]       [ symbolic-reference ]       [ nonsymbolic-address ]</p> <p><b>HELP</b></p> <p><b>SET BREAK</b> { %LINE line-number }               { entry-name }               { symbolic-reference }               { nonsymbolic-address }</p> <p>[ DO (cmd [;cmd...]) ]</p> <p>[ /AFTER:n ]</p> <p><b>SET EXCEPTION BREAK</b></p> <p><b>SET LANGUAGE</b> language-name</p>	<p>Displays the current contents of a variable or program location</p> <p>Ends the debugging session and returns control to the command interpreter</p> <p>Starts or continues program execution</p> <p>Displays a description of a debugger command, parameter, or qualifier</p> <p>Sets a breakpoint at a specified statement, procedure, or program address</p> <p>Requests that the debugger treat external exception conditions as if they were breakpoints, and interrupt the program when an exception occurs rather than to allow ON-units to execute</p> <p>Specifies the source language of a module or routine, for language-specific debugging</p>

(Continued on next page)

Table 1-1 (Cont.)  
Summary of Debug Commands

Command Syntax	Function						
<b>SET LOG</b> [file-spec]	Specifies the name of a log file to which the debugger should write program output when the SET OUT LOG command has been entered						
<b>SET MODE</b> { <table style="display: inline-table; vertical-align: middle;"> <tr><td>DECIMAL</td></tr> <tr><td>HEXADECIMAL</td></tr> <tr><td>OCTAL</td></tr> <tr><td>NOSYMBOLIC</td></tr> <tr><td>SYMBOLIC</td></tr> </table> } , ...	DECIMAL	HEXADECIMAL	OCTAL	NOSYMBOLIC	SYMBOLIC	Sets the default mode for entering and displaying program locations that are not declared variables	
DECIMAL							
HEXADECIMAL							
OCTAL							
NOSYMBOLIC							
SYMBOLIC							
<b>SET MODULE</b> { <table style="display: inline-table; vertical-align: middle;"> <tr><td>module-name</td></tr> <tr><td>/ALL</td></tr> </table> } , ...	module-name	/ALL	Adds the symbols from the indicated module(s) to the debugger's symbol table.				
module-name							
/ALL							
<b>SET OUTPUT</b> { <table style="display: inline-table; vertical-align: middle;"> <tr><td>[ LOG ]</td></tr> <tr><td>[ NOLOG ]</td></tr> <tr><td>[ TERMINAL ]</td></tr> <tr><td>[ NOTERMINAL ]</td></tr> <tr><td>[ VERIFY ]</td></tr> <tr><td>[ NOVERIFY ]</td></tr> </table> } , ...	[ LOG ]	[ NOLOG ]	[ TERMINAL ]	[ NOTERMINAL ]	[ VERIFY ]	[ NOVERIFY ]	Controls whether the debugger writes output to a log file or to the terminal, and whether it echoes commands executed from command procedures
[ LOG ]							
[ NOLOG ]							
[ TERMINAL ]							
[ NOTERMINAL ]							
[ VERIFY ]							
[ NOVERIFY ]							
<b>SET SCOPE</b> { <table style="display: inline-table; vertical-align: middle;"> <tr><td>0</td></tr> <tr><td>\</td></tr> <tr><td>scope-number</td></tr> </table> } , ...	0	\	scope-number	Specifies the modules to be searched to find a symbol and the order in which they are to be searched			
0							
\							
scope-number							
<b>SET STEP</b> { <table style="display: inline-table; vertical-align: middle;"> <tr><td>[ OVER ]</td></tr> <tr><td>[ INTO ]</td></tr> <tr><td>[ SYSTEM ]</td></tr> <tr><td>[ NOSYSTEM ]</td></tr> <tr><td>[ INSTRUCTION ]</td></tr> <tr><td>[ LINE ]</td></tr> </table> }	[ OVER ]	[ INTO ]	[ SYSTEM ]	[ NOSYSTEM ]	[ INSTRUCTION ]	[ LINE ]	Specifies how the debugger is to behave when the STEP command is issued
[ OVER ]							
[ INTO ]							
[ SYSTEM ]							
[ NOSYSTEM ]							
[ INSTRUCTION ]							
[ LINE ]							
<b>SET TRACE</b> { <table style="display: inline-table; vertical-align: middle;"> <tr><td>%LINE line-number</td></tr> <tr><td>entry-name</td></tr> <tr><td>symbolic-reference</td></tr> <tr><td>nonsymbolic-address</td></tr> <tr><td>/BRANCH</td></tr> <tr><td>/CALL</td></tr> </table> }	%LINE line-number	entry-name	symbolic-reference	nonsymbolic-address	/BRANCH	/CALL	Establishes a tracepoint at a specified statement, procedure, entry, or program location
%LINE line-number							
entry-name							
symbolic-reference							
nonsymbolic-address							
/BRANCH							
/CALL							

(Continued on next page)



INTRODUCTION TO DEBUGGING VAX-11 PL/I PROGRAMS

Table 1-1 (Cont.)  
Summary of Debug Commands

Command Syntax	Function
<b>SET TYPE</b> { /ASCII:length } { /BYTE } { /INSTRUCTION } { /LONG } { /WORD }  [ /OVERRIDE ]	Sets the default data types for the DEPOSIT and EXAMINE commands for locations that do not have declared data types
<b>SET WATCH</b> variable-reference	Establishes a watchpoint on a specified static variable
<b>SHOW BREAK</b>	Displays current breakpoints
<b>SHOW CALLS</b> [integer]	Displays the current program location and all, or a specified number of, preceding calls
<b>SHOW LANGUAGE</b>	Displays the current debugging language
<b>SHOW LOG</b>	Displays the current status of the log file, if any
<b>SHOW MODE</b>	Displays the current default entry and display modes
<b>SHOW MODULE</b>	Lists the modules in the image being debugged and shows which modules have names in the debugger's symbol table
<b>SHOW OUTPUT</b>	Displays the current status of the debugger's output files
<b>SHOW SCOPE</b>	Displays the current default scopes
<b>SHOW STEP</b>	Displays the current default step conditions
<b>SHOW TRACE</b>	Displays current tracepoints
<b>SHOW TYPE</b> [/OVERRIDE]	Displays current default data type or override type

(Continued on next page)

Table 1-1 (Cont.)  
Summary of Debug Commands

Command Syntax	Function
SHOW WATCH	Displays current watchpoints and the number of bytes being watched
STEP { [ /OVER ] [ /INTO ] [ /SYSTEM ] [ /NOSYSTEM ] [ /INSTRUCTION [ integer ] ] [ /LINE [ integer ] ] }	Executes one or more statements, or into or over subroutines

#### 1.4 SAMPLE TERMINAL SESSION

The sample program REMEMBER is listed below, with the line numbers assigned by the compiler. This program reads a file consisting of names and birthdates, compares each birthday with the current date, and displays a message if any dates match. This program has an obvious bug -- the pointer, P, is not initialized to point to the input record buffer, INREC -- but it will serve to illustrate some simple debugging commands.

```

1      REMEMBER: PROCEDURE;
2      1
3      1  DECLARE P POINTER,
4      1      1 NAME AGE BASED(P),
5      1      2 NAME CHARACTER(40),
6      1      2 BIRTHDAY CHARACTER(6),
7      1      2 REST CHARACTER(34),
8      1      INREC CHARACTER(80) STATIC,
9      1      NAMES FILE RECORD INPUT SEQUENTIAL,
10     1      EOF BIT(1) STATIC INIT('0'B);
11     1
12     1      ON ENDFILE (NAMES) EOF = '1'B;
13     1      OPEN FILE(NAMES);
14     1
15     1      READ FILE (NAMES) INTO(INREC);
16     1      DO WHILE (^EOF);
17     2          IF SUBSTR(AGE(),3,4) = SUBSTR(BIRTHDAY,3,4)
18     2              THEN PUT SKIP EDIT(NAME,'is',
19     2                  BINARY(SUBSTR(AGE(),1,2)) -
20     2                      BINARY(SUBSTR(BIRTHDAY,1,2))),
21     2                  'Today!') (2(A,X),F(2),X,A);
22     2          READ FILE(NAMES) INTO(INREC);
23     2          END;
24     1  END;
```

## 1.4.1 Executing the Sample Program

Assume, for the purposes of this example, that you know that at least one record in the file contains a BIRTHDAY field that matches the current date. You compile, link, and run the program as follows:

```
$ PLI REMEMBER
$ LINK REMEMBER
$ RUN REMEMBER
$
```

The program runs to completion without displaying the message you expected. To debug the program, you must have a listing, and you must compile and link with the debugger, as follows:

```
$ PLI/LIST/DEBUG REMEMBER
$ LINK/DEBUG REMEMBER
$ PRINT REMEMBER
```

The PRINT command prints the listing, which shows the line numbers. You are now ready to begin a debugging session. The notes below are keyed to the terminal session that follows.

1. When you enter the RUN command, the debugger displays its informational message and prompts you with its DBG> prompt.
2. You decide that the problem may be that P has not been initialized. You can test this hypothesis by finding out the address of INREC and putting this value in P. First, you want to get the program to execute up to the first READ statement.

To run a program to a certain point, you can set a breakpoint at a particular line. In this example, you set the breakpoint at line 15.

3. The GO command starts the execution of the program. The debugger tells you where, in the program, you are beginning execution.
4. When line 15 is reached, the debugger interrupts its execution and prompts you to enter a command.
5. At line 15, you examine the contents of the pointer P. The debugger displays the value of P, which does not look like a program address.
6. You use the EVALUATE/ADDRESS command to determine the virtual address of INREC. This would be the equivalent, in PL/I, of using the ADDR built-in function to set a pointer. The debugger displays the address of INREC.
7. You use the DEPOSIT command to give the pointer P the value of the address of INREC.
8. The GO command continues the execution of the program. As you can see, the program outputs its expected result.
9. When the program exits, the debugger displays a message indicating the termination status.
10. The EXIT command terminates the debugging session.

You can now correct the program so that it initializes the pointer P.

# INTRODUCTION TO DEBUGGING VAX-11 PL/I PROGRAMS

\$ RUN REMEMBER ①

VAX-11 DEBUG Version 2.00

%DEBUG-I-INITIAL, language is BASIC, module set to 'REMEMBER'

DBG>SET BREAK %LINE 15 ②

DBG>GO ③

routine start at REMEMBER\REMEMBER

break at REMEMBER\REMEMBER %LINE 15 ④

DBG>EXAMINE P ⑤

REMEMBER\REMEMBER\P: 3

DBG>EVALUATE/ADDRESS INREC ⑥

513

DBG>DEPOSIT P = 513 ⑦

DBG>GO ⑧

start at REMEMBER\REMEMBER %LINE 15

J. RANDOM PROGRAMMER is 19 today!

%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion' ⑨

DBG>EXIT ⑩

\$



## CHAPTER 2

### RECOGNITION OF NAMES

This chapter describes how to specify names to the debugger.

#### 2.1 DEBUGGER SYMBOL TABLE

The debugger maintains a symbol table that lists the symbols you can reference during a debugging session. The debugger symbol table always contains the names of global symbols in the image. The names of local symbols, that is, names of internal variables defined within your program, are available in the image file only if you included the /DEBUG qualifier in the PLI and LINK commands.

The symbol table contains the data type attributes and memory location of each accessible name or variable. The data type attributes includes dimension bound information for arrays, and length information for character data.

##### 2.1.1 Names Included in the Symbol Table by Default

Before you can reference a name, you must ensure that the name is in the debugger symbol table. When a debugging session begins, you have access to global symbols and to automatic variables that are declared within the indicated module name and static variables that are declared within internal blocks, as long as there are no naming conflicts. For example, a PL/I procedure may contain the lines:

```
MAINP: PROCEDURE OPTIONS(MAIN);
DECLARE (X,Y,Z) STATIC FIXED,
        (A,B,C) AUTOMATIC BIT;
      .
PRINTLIST: PROCEDURE;
DECLARE COUNT STATIC FIXED,
        X CHARACTER(10);
```

When this debugging session begins, you can by default access the names X, Y, Z and A, B, and C in MAINP as well as COUNT in PRINTLIST.

When you want to access a variable or location that is not in the default symbol table, you must specify the module containing the variable or location. A module, in PL/I terms, is the name of a level-one procedure, the outermost procedure in the source file (indicated in the source program listing by the number "1" in the left margin).

## RECOGNITION OF NAMES

### 2.1.2 Adding Names to the Symbol Table

The debugger symbol table accommodates approximately 2000 symbols. If you are debugging multiple procedures that define more than 2000 symbols, you can use the SET MODULE command to copy symbols from other modules to the symbol table. For example, a PL/I procedure may declare an external entry as follows:

```
DECLARE PRINT_ARGS EXTERNAL ENTRY;
```

To reference names of static variables declared in PRINT\_ARGS before PRINT\_ARGS is invoked in the debugging session, you can bring these names into the symbol table by entering the command:

```
DBG>SET MODULE PRINT_ARGS
```

This command makes the names of variables in PRINT\_ARGS accessible.

Subsequently, you can use the CANCEL MODULE command to remove from the symbol table symbols you no longer need, and then use the SET MODULE command to insert the symbols you next require.

Note that you cannot access the names of automatic variables until the block that declares these variables is executing, since the variables are not allocated storage until the block is activated.

### 2.1.3 Displaying Names in the Symbol Table

Use the SHOW MODULE command to display the current contents of the symbol table. For example:

```
DBG>SHOW MODULE
module name      symbols  language  size
ARGLIST          yes     BASIC     148
PRINT_ARGS       yes     BASIC     280
PLI$CONDIT       no      MACRO     716
PLI$CONTROL      no      MACRO     336
PLI$PUTFILE      no      MACRO     176
PLI$PUTLISTITEM no      MACRO     176
PLI$PUTBUFFER    no      MACRO     176
PLI$CONVERT      no      MACRO     284
PLI$CLOSE        no      MACRO     228
PLI$CVTPIC       no      MACRO     336
PLI$OPEN         no      MACRO     228
PLI$RECOPT       no      MACRO     176
PLI$BIT          no      MACRO     608
PLI$CHAR         no      MACRO     284
PLI$$BYTESIZE   no      MACRO     228
LIB$LP_LINES     no      BLISS     120
OTSS$CVTDT       no      MACRO     120
OTSS$CVTRT       no      MACRO     176
RMSGBL           no      MACRO     120
total modules: 19.  remaining size: 59304.
```

The modules with names PLI\$, LIB\$, RMS, and OTSS\$ prefixes are run-time modules required for the execution of the PL/I procedures. For a summary of these modules, see Appendix A.

2.2 SPECIFYING REFERENCES AND LOCATIONS

The debugger's symbol table lets you reference names and program locations symbolically. You need concern yourself only with the name, and not the memory location, of the data. This symbolic form of reference applies to program data, such as variables and array elements, and to program addresses, such as program line numbers and procedure names.

You can reference the following kinds of symbols:

- Internal and external variables
- Global symbols
- Program locations
- Symbols you create with the debugger command DEFINE
- Permanent symbols defined by the debugger

Symbols can specify variable references or can contain data values. The debugger interprets data items you specify according to these rules:

1. If a data item begins with an alphabetic letter, the debugger assumes that it is a program variable or a symbolic reference to an address.
2. If a data item begins with a numeric integer (0 through 9), the debugger assumes that the item is a literal numeric constant.
3. If a data item is enclosed in apostrophes or quotation marks, the debugger assumes that the item is a character-string constant.

2.2.1 Specifying Internal and External Variables

You can reference both internal and external variables while debugging PL/I procedures. Internal automatic variables can be referenced only in the block in which they are declared.

There is no up-level addressing, that is, an internal automatic variable in a containing block cannot be examined in a contained block. For example:

```

DECLARE X FIXED;
      .
      .
      INSIDE: PROCEDURE;
    
```

When these PL/I statements are debugged, the variable X cannot be examined or modified within the procedure INSIDE, even though INSIDE may reference X.

You can specify data addresses symbolically for scalar variable names and scalar array elements. For example, a PL/I procedure may contain the following declarations:

```

DECLARE X_MSG CHARACTER(80) STATIC,
      X_LEN(10) FIXED STATIC;
    
```

These variables can be referenced in a debugging session as follows:



## RECOGNITION OF NAMES

```
DBG>DEPOSIT X_MSG = 'This is new string'  
DBG>EXAMINE X_LEN(5)  
XLOOK\XLOOK\X_LEN(5): +14
```

The DEPOSIT command places a new character-string value in the variable X\_MSG. The EXAMINE command displays the current contents of the array element X\_LEN(5).

You can reference array elements using constants and variable expressions. If you reference a variable or array element that is not defined in the symbol table, or if you attempt to reference out of the array bounds defined at compile time, the debugger issues a warning.

### 2.2.2 References to Global Symbols

Global symbols can be referenced from all blocks. In a VAX-11 PL/I procedure, global symbols are those symbols defined with the GLOBALREF or GLOBALDEF attributes, as well as the names of level-one procedures.

### 2.2.3 Specifying Program Locations

You can specify address expressions, that is, program locations by procedure name, line number, or (nonsymbolic) virtual address. To specify a procedure by name, give the command followed by the name of the procedure. For example, the command

```
DBG>SET BREAK LIST_BY_FLOWER
```

sets a breakpoint at the entry to procedure LIST\_BY\_FLOWER.

To specify a line number, use the %LINE specifier, as shown here:

```
DBG>SET BREAK %LINE 6
```

This command sets a breakpoint at line 6, corresponding to the compiler-generated line number shown in the listing.

Note that the debugger does not recognize all line numbers. In particular, it does not recognize those line numbers associated with nonexecutable statements, such as DECLARE and FORMAT statements. If you specify such a line number, the debugger responds with a message indicating that no such line exists.

You can also set a breakpoint as follows:

```
DBG>SET BREAK %LINE LIST_BY_FLOWER\11
```

This command sets a breakpoint at line 11 in LIST\_BY\_FLOWER.

To specify a virtual address, issue the command without a prefix. For example:

```
DBG>SET BREAK 700
```

You can determine the virtual address of a line number or a variable by entering an EVALUATE command as follows:

```
DBG>EVALUATE/ADDRESS %LINE 17  
800
```

The debugger displays the virtual address of the instructions for the statement on line 17.

### 2.2.4 Defining Addresses Symbolically

At times, you may want to assign a symbolic name to a program location. To assign a symbolic name to a location, you must first determine the virtual address of the location and then use the DEFINE command. To determine the virtual address of a location, use the EVALUATE/ADDRESS command. For example:

```
DBG>EVALUATE/ADDRESS %LINE 42
1666
DBG>DEFINE CHK = 1666
```

Subsequent references to line 42 can be made using the defined symbol CHK. For example, the command

```
DBG>SET BREAK CHK
```

sets a breakpoint at line 42. Similarly, the commands

```
DBG>EVALUATE/ADDRESS CARD_COUNTER
6445
DBG>DEFINE CC = 6445
```

define a symbolic name by which the variable CARD\_COUNTER may be referenced.

### 2.2.5 The Debugger's Permanent Symbols

The debugger has the following permanent symbols; you can reference them at any time during the debugging session.

- R0 - R11    General registers 0 through 11
- AP            Argument pointer
- FP            Frame pointer
- SP            Stack pointer
- PC            Program counter
- PSL           Processor status longword

These names cannot be redefined; that is, you cannot use the name R0 to create a symbol definition with the DEFINE command.

## 2.3 SCOPE

If the program you are debugging consists of more than one procedure, you must be sure that your symbolic references are unambiguous. To make a reference unambiguous, you can specify the "scope" of the reference to the debugger: in PL/I terms, the scope of a name is simply the block in which the name is declared.

Most of the time, you can let the debugger determine the scope of a name for you. At certain times, however, you must tell the debugger how to resolve symbolic references. For example, assume that you are debugging two procedures; both procedures use an internal variable I, and both modules are included in the debugger's symbol table. Unless you explicitly specify the scope of I, the debugger may be unable to determine which variable I you want.

## RECOGNITION OF NAMES

You specify scope in one of three ways:

- By using the debugger default scope in effect
- By explicitly specifying the reference's scope with its symbolic name in the command
- By setting a new default scope with the SET SCOPE command

When you begin a debugging session, the debugger automatically defines the first procedure linked (normally the main procedure) as the default scope. However, this default scope is dynamic; that is, as you debug your program, the default scope (also called the PC scope) is always the procedure that is currently executing. When the debugger is resolving a reference, it follows this order in determining the scope:

1. If the specified symbolic name is unique within the debugger symbol table, then the debugger uses that name.
2. If the specified symbol is ambiguous -- that is, it is not unique within the symbol table, but one of its occurrences is within the current PC scope -- then the debugger uses the occurrence in the current scope.
3. If the specified symbol is not defined in the symbol table, or if it is ambiguous with no occurrence defined within the current scope, then the debugger issues an error message indicating that the name is ambiguous.

### 2.3.1 Specifying Pathnames

You can specify the scope of a name explicitly by providing both the name of the symbol and the names of the module and routine in which it is located, separated by a backslash (\) character. This type of specification is called a pathname, since in some cases it may consist of the names of several nested routines. For example, a PL/I procedure may contain the following:

```
MAINP: PROCEDURE OPTIONS(MAIN);
DECLARE X FIXED STATIC;
.
.
INSIDEOUT: PROCEDURE;
DECLARE X BIT;
```

To examine the contents of X within the procedure INSIDEOUT during a debugging session when the current scope is INSIDEOUT, you must specify MAINP as both the routine name and the module name in a pathname. For example:

```
DBG>EXAMINE MAINP\MAINP\X
```

Similarly, to specify an address reference in a routine that is not the current scope, you must give it a pathname, as in this example:

```
DBG>EXAMINE INSIDEOUT\X
```

Note that when you use a %LINE specifier, the specifier must appear before the pathname. For example:

```
DBG>SET BREAK %LINE SUB1\7
```

This command sets a breakpoint at line 7 in the scope of the module SUB1.

## RECOGNITION OF NAMES

Note that if you want to make frequent references to a location with a long pathname, you can define a symbol name for it with the DEFINE command. For example:

```
DBG>SET SCOPE INSIDE
DBG>EVALUATE/ADDRESS CARD_COUNTER
9965
DBG>DEFINE CC = 9965
DBG>SET SCOPE MAINP
.
.
DBG>EXAMINE CC
```

In this example, the SET SCOPE command changes the scope to the module INSIDE, the EVALUATE/ADDRESS command displays the virtual address of the variable CARD\_COUNTER, and the DEFINE command uses this value to define the symbol named CC. Subsequently, the scope is reset to MAINP. During the debugging session, the value of CARD\_COUNTER can be referenced using the symbolic name CC, regardless of the current scope.

### 2.3.2 Changing the Scope

If you want to make a number of symbolic references within the same procedure, you can eliminate the need to specify scope with each symbolic address by using the SET SCOPE command. For example, the following command sets the scope to SUB3:

```
DBG>SET SCOPE SUB3
```

You can also define a scope list to specify the order in which the debugger should search for symbols. For example, the command

```
DBG>SET SCOPE MAR,JAN,FEB
```

instructs the debugger to search for symbols first in procedure MAR. If it cannot find a specified symbol in MAR, then the debugger searches JAN and, if necessary, FEB.

The scope defined in a SET SCOPE command becomes the default scope for all symbolic references until you explicitly change or cancel the scope. You can determine the current scope at any time by entering the SHOW SCOPE command. For example:

```
DBG>SHOW SCOPE
scope: SUB2,SUB1
```

The message shows that the current scope is set first to SUB2, then to SUB1. The SHOW SCOPE command may also respond as follows:

```
DBG>SHOW SCOPE
scope: 0 [= MULT\MULT]
```

The symbol 0 shows that the current scope is the default PC scope. Within brackets, the debugger displays the module and routine name of the default scope: the scope is module MULT, routine MULT.

The CANCEL SCOPE command resets scope to the default PC scope.

Note that when you explicitly SET SCOPE to a procedure (module) name, the debugger implicitly performs a SET MODULE command. Therefore, symbols for the procedure specified in your SET SCOPE command are placed in the symbol table. However, if you use the debugger default scope (PC scope), you must also use SET MODULE to place symbols for the procedure in the symbol table.

2.3.3 The Scope of Automatic Variables

If you reference an automatic variable when the block that defines the variable is not in the current scope, the debugger displays a warning message. For example, this occurs when you try to reference an automatic variable declared in a procedure that has executed a RETURN statement, and control has returned to the debugger:

```
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>EXAMINE X
%DEBUG-I-PCNOTINSCP, PC is not within scope of routine declaring symbol
XLOOK\XLOOK\X: 3
```

This message notifies you that the variable X in the routine XLOOK does not have an address assigned exclusively to it and that its address may have another use in the current section of your program.

2.4 SPECIAL CHARACTERS AND EXPRESSIONS

This section summarizes how the debugger interprets special characters in arithmetic expressions and in address expressions. You can use these operators in references and expressions; the debugger will perform the arithmetic on integers.

2.4.1 Characters for Arithmetic Expressions

Table 2-1 lists special characters used in arithmetic expressions.

Table 2-1  
Arithmetic Operators

Character	Interpretation
+	Arithmetic addition (binary) operator, or unary plus sign
-	Arithmetic subtraction (binary) operator, or unary minus sign
*	Arithmetic multiplication operator
/	Arithmetic division operator
@	Arithmetic shift operator
< >	Precedence operators; do <enclosed> first
^D	Decimal radix operator
^O	Octal radix operator
^X	Hexadecimal radix operator

2.4.2 Characters for the Current, Previous, and Next Locations

The debugger provides a quick method for referencing the relative data addresses or locations in DEPOSIT and EXAMINE commands:

## RECOGNITION OF NAMES

Symbol	Meaning
.	The current location (the location most recently referenced by an EXAMINE or DEPOSIT command). Use this symbol in PL/I to reference a scalar variable, or an element of a static array of scalars.
^	The previous location (the location at the next lower address from the current location). Use this symbol in PL/I to reference the previous element of an array of 32-bit scalar variables.
(RET)	The next location (the location at the next higher address from the current location). Press (RET) in PL/I to reference the next element in an array of scalar variables.

For example, assume the following PL/I variable declaration:

```
DECLARE X_LEN(10) FIXED STATIC;
```

Elements of this array may be accessed as follows:

```
DBG>EXAMINE X_LEN(5)
XLOOK\XLOOK\X_LEN(5): +14
DBG>DEPOSIT . = 100
```

This DEPOSIT command puts a value of 100 in the variable most recently referenced, that is, X\_LEN(5).

To specify the previous location, type an up arrow or a circumflex (^). For example:

```
DBG>EXAMINE ^
XLOOK\XLOOK\X_LEN(4): +19
```

This EXAMINE command displays the contents of the previous location, that is, X\_LEN(4).

To specify the next higher location, simply omit the variable reference. For example:

```
DBG>EXAMINE (RET)
XLOOK\XLOOK\X_LEN(5): 100
```

This EXAMINE command displays the contents of the next element in the array X\_LEN.

The EXAMINE and DEPOSIT commands, and restrictions on the data types that you can examine and deposit, are described in the next chapter.



## CHAPTER 3

### EXAMINING AND DEPOSITING DATA

This chapter describes considerations for displaying, interpreting, and modifying the contents of PL/I variables using the VAX-11 Symbolic Debugger.

#### 3.1 USING THE EXAMINE AND DEPOSIT COMMANDS

The EXAMINE and DEPOSIT commands display and change the contents of variables, respectively. The EXAMINE command displays the contents of selected variables. You can use EXAMINE to display any combination of the following:

- A scalar variable
- Multiple scalar variables
- A range of array elements
- Multiple ranges of array elements

If you specify more than one variable and separate them with commas, the contents of each variable specified are displayed. However, if you use a colon to separate a pair of elements of an array, then all elements within that range are displayed. For example:

```
DBG>EXAMINE STRING(1):STRING(5)
CALC\CALC\STRING(1)(1:10): stringa
CALC\CALC\STRING(2)(1:10): stringb
CALC\CALC\STRING(3)(1:10): stringc
CALC\CALC\STRING(4)(1:10): stringd
CALC\CALC\STRING(5)(1:10): stringe
```

This EXAMINE command displays the elements in the array STRING from element one through element five. When the debugger displays variables declared, it precedes the variable name with the pathname used to locate the variable, if it knows it, and it displays the length of the variable.

In the examples above, the pathname CALC\CALC indicates that the program consists of only one routine: the routine name and the module name are the same.

##### 3.1.1 Specifying the Data Type of Data to Deposit

When you examine a PL/I variable or deposit data into one, you do not need to specify the data type of the variable, unless you want to deposit data of a different type. In the following example, XVALUE is declared with the attributes FLOAT BINARY:



## EXAMINING AND DEPOSITING DATA

```
DBG>EXAMINE XVALUE
MAIN\XVALUE: 14.50000
DBG>EXAMINE/BYTE XVALUE
MAIN\XVALUE: 68
```

The debugger always uses the declared data type (including extent and precision) of a variable, unless you override it. In this example, the /BYTE qualifier tells the debugger to display only the contents of the first byte of the storage occupied by the variable XVALUE.

You can use the SET TYPE/OVERRIDE command to tell the debugger to display all variables using a certain type, for example:

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

After this command is issued, the debugger only displays the first byte of any variable's storage. To restore the normal interpretation of data types, use the CANCEL MODE command.

### 3.1.2 Restrictions on Examining and Depositing Data

For this release of VAX-11 PL/I, there are restrictions on both the data types and storage classes of variables that you can access. You cannot examine or modify:

- Structures
- Arrays with asterisk (\*) or variable extents
- Variables with asterisk (\*) or variable extents
- Label variables
- Pictures
- Parameters
- File data
- Formats
- Area or offset data
- Defined or based variables

In general, you can examine, evaluate, and deposit into a static, scalar variable of any data type. You can also examine static arrays.

Static variables that are not assigned or initialized have initial values of zero. If you display them, numeric values and bit strings are displayed as zeros; character strings are null bytes, which are nonprinting characters and appear blank when displayed. For example:

```
DBG>EXAMINE P
MAINP\MAINP\P(1:10): +0000000000
DBG>EXAMINE A
MAINP\MAINP\ALPHA\A(1:10):
```

Automatic variables may also be examined and deposited into; however, since automatic variables are allocated from stack storage, their contents are not valid until after they have been assigned. For example:

```
DBG>EXAMINE X
MAINP\MAINP\X: 2147287308
```

## EXAMINING AND DEPOSITING DATA

In this example, the contents of variable X are meaningless until after the assignment of a value to the variable X.

There are special considerations for examining automatic arrays, character strings, bit strings, and fixed-point decimal variables. When you examine automatic variables whose storage is more than a longword, you must supply a range of addresses or a length to the debugger. To examine a range, you must change the language to MACRO.

The remainder of this chapter provides notes on examining and depositing into static and automatic variables of different data types.

The program MAINP, shown in Figure 3-1, contains the statements and declarations that are referenced in the examples in the remainder of this chapter.

```
1 | /* Sample Program for Explaining Debugger Rules */
2
3   MAINP: PROCEDURE OPTIONS (MAIN);
4   1
5   1 DECLARE (X, Y, VALUE) FIXED,
6   1         (P, Q, R) FIXED DECIMAL (10,5) STATIC;
7   1
8   1 X = 2;
9   1 Y = 3;
10  1 VALUE = X+Y;
11  1 PUT SKIP LIST(VALUE);
12  1
13  1     P = 123.45;
14  1     Q = 66666.3333;
15  1     R = DIVIDE(Q,P,10,5);
16  1     PUT SKIP LIST(R);
17  1 CALL ALPHA;
18  1
19  1     ALPHA: PROCEDURE; /* Internal procedure */
20  2     DECLARE RESULT FLOAT STATIC,
21  2           A CHARACTER(10) STATIC,
22  2           B BIT(32) ALIGNED STATIC,
23  2           C CHARACTER(10),
24  2           D CHARACTER(60) VARYING;
25  2
26  2 A = 'AAAAA';
27  2 B = '11000'B;
28  2 C = 'CCCCC';
29  2 D = A||B||C;
30  2 PUT SKIP LIST(D);
31  2
32  2
33  2     BETA: BEGIN; /* Begin block */
34  3     DECLARE SQUARE_ROOTS(10) FLOAT STATIC,
35  3           X FIXED;
36  3
37  3 DO X = 1 TO 10;
38  4     SQUARE_ROOTS(X) = SQRT(X);
39  4     PUT SKIP LIST(SQUARE_ROOTS(X));
40  4     END;
41  3     END BETA;
42  2
43  2     END ALPHA;
44  1 END MAINP;
```

Figure 3-1 The Sample Program, MAINP

## 3.2 FIXED-POINT BINARY AND FLOATING-POINT VARIABLES

You can use the EXAMINE and DEPOSIT commands with fixed-point binary and floating-point variables. For example:

```
DBG>EXAMINE Y
MAINP\MAINP\Y: 3
DBG>DEPOSIT Y = 866
DBG>STEP
start at MAINP\MAINP %LINE 10
stepped to MAINP\MAINP %LINE 11
DBG>EXAMINE VALUE
MAINP\MAINP\VALUE: 868
```

Here, the EXAMINE command displays the contents of the fixed-point variable, Y, after its assignment on line 9 in Figure 3-1. Then, a DEPOSIT command changes its contents, a STEP command executes the next statement, and the EXAMINE command displays the resulting contents of VALUE.

## 3.3 FIXED-POINT DECIMAL DATA

You can examine and deposit into static, scalar variables with the fixed-point decimal data type. However, you must infer the position of the decimal point in the value. For example:

```
DBG>EXAMINE R
MAINP\MAINP\R(1:10): +0054002700
```

The precision and scale factor of R are (10,5); thus, this value represents 540.027.

## 3.4 CHARACTER-STRING VARIABLES

The debugger best supports fixed-length static character-string variables. When you examine such a variable, the debugger displays the entire storage of the variable. When you deposit data in it, the debugger by default changes the entire storage of the variable. For example, after the assignment of A on line 26 in Figure 3-1:

```
DBG>EXAMINE A
MAINP\MAINP\ALPHA\A(1:10): AAAAA
```

To examine or change only a portion of a variable, use the /ASCII qualifier to specify the number of characters you want to change, as in this example:

```
DBG>DEPOSIT/ASCII:2 A = 'CC'
```

This command changes only the first two characters of the variable A. Note that you must enclose strings in apostrophes when you specify them to the debugger, as is true in PL/I.

When you examine a fixed-length character-string variable that has the AUTOMATIC attribute, you must specify /ASCII:length on the EXAMINE command to examine the variable. For example:

```
DBG>EXAMINE/ASCII:10 C
2147287779: CCCCC
```

Remember that the value of an automatic variable is not valid until after it has been assigned.

## EXAMINING AND DEPOSITING DATA

For character-string variables with the VARYING attribute, you must change the language to MACRO to determine the current length and display the contents of the variable. The first word of the variable's storage contains its length. For example:

```
DBG>SET LANGUAGE MACRO
DBG>EXAMINE/WORD D
7FFD02DE: 0034
DBG>EXAMINE/ASCII:34 D+2
7FFD02E0: AAAAA 11000000000000000000000000000000CCCC
```

In this example, the length of the variable (after its assignment in statement 29) is 34 hexadecimal. This value is then used as a range in the examination of the contents of the variable, which begins two bytes beyond the beginning of the variable's storage.

Note that when you specify /ASCII, the debugger displays the virtual address of the variable, rather than its identifier.

### 3.5 BIT-STRING VARIABLES

The debugger treats and displays bit strings as if they were longwords. For example:

```
DBG>EXAMINE B
MAINP\MAINP\ALPHA\B: 3
```

Note that bit-string values are stored in reverse order, as in the preceding example. The bit-string constant '11000' is stored as '00011', or 3 (decimal).

The most efficient way to modify a bit-string variable is to use the DEPOSIT command with the /HEXADECIMAL qualifier. For example:

```
DBG>DEPOSIT/HEX B = 0C0CC
DBG>EXAMINE B
MAINP\MAINP\ALPHA\B: 49356
```

Bit strings may be more meaningful if you examine the contents of the variable in hexadecimal. For example:

```
DBG>EXAMINE/HEX B
MAINP\MAINP\ALPHA\B: 0000C0CC
```

### 3.6 STATIC ARRAYS

The debugger can interpret static array references of up to seven dimensions only. You can refer to static arrays of the data types listed below using subscripted references. The valid data types are as follows:

- Fixed-point binary (FIXED BINARY)
- Floating point (FLOAT DECIMAL or FLOAT BINARY)
- Character nonvarying (CHARACTER)
- Aligned bit strings (BIT ALIGNED)

For example, the floating-point array SQUARE\_ROOTS may be examined as follows:

## EXAMINING AND DEPOSITING DATA

```
DBG>EXAMINE SQUARE_ROOTS(2)
MAINP\MAINP\ALPHA\BEGIN%31\SQUARE_ROOTS(2):  2.000000
DBG>EXAMINE SQUARE_ROOTS(7)
MAINP\MAINP\ALPHA\BEGIN%31\SQUARE_ROOTS(7):  2.645751
```

Arrays with bit-string elements are valid only if the array has the ALIGNED attribute, and if the length of the bit-string elements, when rounded to the nearest byte, is 1, 2, or 4.

Under these circumstances, the debugger will recognize the array but treat it as a byte, word, or longword array (that is, an array of fixed binary variables with a precision of 7, 15, or 31). To examine the elements of a such an array, it is convenient to use the /HEXADECIMAL qualifier of the debugger command EXAMINE. For example, a bit-string array may be declared and assigned values as follows:

```
DECLARE BITS(5) BIT (31) ALIGNED STATIC;
DECLARE X FIXED;

DO X = 1 TO 5;
  BITS(X) = BIT(X);
END;
```

During a debugging session, these elements may be examined as follows:

```
DBG>EXAMINE/HEX BITS(1):BITS(5)
ARRAYS\ARRAYS\BITS(1):  40000000
ARRAYS\ARRAYS\BITS(2):  20000000
ARRAYS\ARRAYS\BITS(3):  60000000
ARRAYS\ARRAYS\BITS(4):  10000000
ARRAYS\ARRAYS\BITS(5):  50000000
```

Note again that the values of the bit strings are reversed when they are stored internally. These same values, when output with PUT LIST statements, would appear as follows:

```
'00000000000000000000000000000001'B
'00000000000000000000000000000010'B
'00000000000000000000000000000011'B
'00000000000000000000000000000100'B
'00000000000000000000000000000101'B
```

### 3.7 AUTOMATIC ARRAYS AND FIXED-POINT DECIMAL ARRAYS

To examine and modify elements of automatic arrays and of static arrays of fixed-point decimal variables, you must calculate the address of an element or elements and specify the address range in an expression. To specify an address expression, the language must be set to MACRO.

For example, if the bit-string array in the example in the preceding section were declared without the STATIC attribute, you would have to enter the following commands in order to display the elements:

```
DBG>SET LANGUAGE MACRO
DBG>EXAMINE/HEX BITS:BITS+10
ARRAYS\ARRAYS\BITS:  40000000
ARRAYS\ARRAYS\BITS+04:  20000000
ARRAYS\ARRAYS\BITS+08:  60000000
ARRAYS\ARRAYS\BITS+0C:  10000000
ARRAYS\ARRAYS\BITS+10:  50000000
```

where the hexadecimal value 10 represents the address of the last element of the array. Note that when the language is MACRO, the default radix is set to hexadecimal. In this example, each element

## EXAMINING AND DEPOSITING DATA

occupies a longword, or four bytes. The expression `BITS:BITS+10` displays 20 bytes, the total amount of storage occupied by the array.

Fixed-point decimal arrays (both automatic and static) can also be accessed this way. In a fixed-point decimal value, each digit is stored in a four-bit field; the final field contains a sign digit. For example, the array declared as follows:

```
DECIMALS (5) FIXED DECIMAL (10,5)
```

is stored in consecutive six-byte locations. To examine the third element of this array, you can set the language to MACRO and specify the location of the element as follows:

```
DBG>SET LANGUAGE MACRO
DBG>EXAMINE/BYTE DECIMALS+<2*6>:DECIMALS+<3*6>-1
7FFD035C: 02
7FFD035D: 71
7FFD035E: 68
7FFD035F: 79
7FFD0360: 00
7FFD0361: 0C
```

The expression `<2*6>` represents the offset of two six-byte elements from the beginning of the array's storage. The second expression represents the end of the second element. In the output shown above, each byte contains two digits. The current value of `DECIMALS(3)` is 21786.97000. The C indicates that the value is positive. (A 'D' would indicate a negative value.)

You can similarly calculate the addresses of elements of connected automatic arrays of the following data types:

- Fixed-point binary
- Floating point
- Character nonvarying
- Character varying

All arrays are stored in contiguous storage locations. Note that in character-string arrays with the `VARYING` attribute, each element is preceded by a two-byte length field. You must consider this length when you perform the calculations.



## CHAPTER 4

### CONTROLLING A PROGRAM'S EXECUTION

To see what happens during execution of your program, you must be able to suspend and resume the program at specific points. This chapter describes the following debugging concepts:

- Starting and stopping program execution
- Stepping through a program
- Breakpoints
- Tracepoints
- Watchpoints

This chapter also describes how to invoke subroutines during a debugging session.

#### 4.1 STARTING AND STOPPING EXECUTION

Use the GO command to start program execution. You must use this command when you begin the debugging session, and when you want to continue the program's execution after it has been suspended. For example:

```
$ RUN FLOWERS
```

```
VAX-11 DEBUG Version 2.00
```

```
%DEBUG-I-INITIAL, language is 'BASIC', scope and module set to 'FLOWERS'  
DBG>GO
```

```
·  
·
```

```
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'  
DBG>
```

The EXITSTATUS message indicates that the program has run to completion.

When you are finished with the debugging session, use the EXIT command to leave the debugger. You must not restart a program from the beginning unless you first exit from the debugger. Otherwise, unspecified results occur.

#### NOTE

For this release of the debugger, the debugger sometimes displays erroneous error messages when a procedure with the MAIN option completes. You can ignore these messages.



## CONTROLLING A PROGRAM'S EXECUTION

If your program loops or fails to complete executing, or if you need to interrupt it for any other reason, you can press `CTRL/Y` to return to the DCL command level. For example:

```
DBG>GO CTRL/Y
```

```
^Y  
$
```

The \$ prompt on the terminal indicates that you have returned to the DCL command level. To return to the debugger, type `DEBUG` or `CONTINUE`. If you type `DEBUG`, control returns to the debugger and the debugger prompts you for a command. If you type `CONTINUE`, the debugging session continues from where it was interrupted.

If you do not want to continue the debugging session, you can enter a `STOP` command or another DCL command to stop the debugging session. You can also reissue the `RUN` command for the program you are executing, if you want to rerun it beginning with its starting conditions.

### 4.2 STEPPING THROUGH A PROGRAM

When you want to maintain control of your program, to be able to display and/or modify variables following the execution of single statements, you can use the `STEP` command.

You can use the `STEP` command to execute a program one line at a time or you can specify a number of lines to execute. For example:

```
DBG>STEP 5
```

When this command is executed, the debugger executes the next five statements and then suspends the program.

When you are stepping through a program, the debugger displays only the line numbers of the lines as they are executed; it does not display the statements.

The debugger maintains default modes for stepping commands. You can override the default modes by entering qualifiers on a `STEP` command, or by entering a `SET STEP` command to change the default. For example, the default step for higher-level languages is `STEP/LINE` where step is a line or statement number increment. In assembly language, the default is `STEP/INSTRUCTION`. Thus, if you want to look at the machine instructions that are executed for each PL/I statement line, enter the debugger command `SET STEP INSTRUCTION`, as follows:

```
DBG>SET STEP INSTRUCTION  
DBG>STEP  
start at MAINP\MAINP\ALPHA %LINE 25  
stepped to MAINP\MAINP\ALPHA %LINE 26 :  
      MOVCS #5,W^1536,#32,#10,B^-72(FP)  
DBG>STEP  
start at MAINP\MAINP\ALPHA %LINE 26  
stepped to MAINP\MAINP\ALPHA %LINE 27 : MOVZWL #32,R1  
DBG>STEP  
start at MAINP\MAINP\ALPHA %LINE 27  
stepped to MAINP\MAINP\ALPHA %LINE 27 +3: MOVZWL #32,R3  
DBG>STEP
```

For each PL/I statement, there are one or more machine-language instructions, and you must enter the `STEP` command for each instruction. The debugger displays the machine language instruction.

## CONTROLLING A PROGRAM'S EXECUTION

When you subsequently issue a STEP command without qualifiers, instruction mode remains in effect. You can supersede this default by including the /LINE qualifier in a STEP command. For example:

```
DBG>STEP/LINE 10
```

This command tells the debugger to execute 10 lines, regardless of the current step default.

It is advisable to use STEP to execute only a few instructions at a time. To execute many instructions, and then stop, use a SET BREAK command to set a breakpoint, and then issue a GO command.

### 4.3 BREAKPOINTS

The BREAK commands let you select specified locations for program suspension. Thus, you can let a program run until it reaches a specified statement, and then you can examine and/or modify variables or arrays in the program. The BREAK commands perform the following functions:

- SET BREAK defines a line number, procedure or entry-point name, or an address at which to suspend execution
- SHOW BREAK displays all breakpoints currently set in the program
- CANCEL BREAK removes selected breakpoints or all breakpoints

For example, the command

```
DBG>SET BREAK %LINE 7
```

sets a breakpoint at the statement corresponding to the line numbered 7 in the source program. When the breakpoint at line 7 is reached during the execution of the program, the debugger interrupts the program, as in this example:

```
DBG>SET BREAK %LINE 7
DBG>GO
routine start at MAINP\MAINP
break at MAINP\MAINP %LINE 7
```

After the breakpoint is set, the GO command continues the program execution. When statement 7 is reached, the debugger interrupts the program and displays a message indicating that the breakpoint is reached. At this breakpoint, you can examine or change static variables, begin stepping through the program, and so on.

To set a breakpoint at a procedure entry point, specify it by name. For example:

```
DBG>SET BREAK PRINT_ROUTINE
```

This command sets a breakpoint at the entry to the procedure PRINT\_ROUTINE.

You can use the /AFTER qualifier to control when a breakpoint takes effect. For instance, if you set a breakpoint on a line that is in the range of a DO loop, and you want the breakpoint to be effective the third time through the loop, then specify /AFTER, as shown in the following example:

```
DBG>SET BREAK/AFTER:3 %LINE 20
```

## CONTROLLING A PROGRAM'S EXECUTION

Note that if you use the /AFTER qualifier, the breakpoint is reported not only the nth time it is encountered, but also every time it is encountered thereafter.

The SET BREAK command also lets you specify some action to be taken each time a breakpoint is encountered. For example, to set a breakpoint at a location, examine one or more variables, and continue, you could enter a SET BREAK command as follows:

```
DBG>SET BREAK %LINE 29 DO(EXAMINE TOTAL; EXAMINE AREA; GO)
DBG>GO
```

After this command, the debugger sets a breakpoint at line 29. Each time the statement on this line is executed, the debugger interrupts the program, displays the contents of the variables TOTAL and AREA, and executes the GO command to continue execution.

You can cancel a breakpoint with the CANCEL BREAK command. For example:

```
DBG>CANCEL BREAK %LINE 9
```

This command cancels the breakpoint at line 9. To cancel all breakpoints, enter:

```
DBG>CANCEL BREAK/ALL
```

You can display the current breakpoints in effect with the SHOW BREAK command.

### 4.4 TRACEPOINTS

A tracepoint is similar to a breakpoint in that it suspends program execution and displays the address at the point of suspension. However, in the case of a tracepoint, program execution resumes immediately. Thus, tracepoints let you follow the sequence of program execution to ensure that execution is carried out in the proper order.

Note that if you set a tracepoint at the same location as a current breakpoint, the breakpoint is canceled, and vice versa.

The TRACE commands perform the following functions:

- SET TRACE establishes lines or entry points within the program at which execution is momentarily suspended.
- SHOW TRACE displays the locations in the program at which tracepoints are currently set.
- CANCEL TRACE removes one or more tracepoints currently set in the program.

For example, you can use the SET TRACE if you want to keep track of the number of times a given subroutine is called, as follows:

```
DBG>SET TRACE INSIDEOUT
```

Each time a call is made to INSIDEOUT, the debugger displays a message like the following:

```
routine trace at MAINP\MAINP\INSIDEOUT
```

The message gives the pathname of the symbol.

## CONTROLLING A PROGRAM'S EXECUTION

To set a tracepoint on a given statement, use the %LINE specifier, as in the example below:

```
DBG>SET TRACE %LINE 30
```

While this tracepoint is set, the debugger displays a message each time the statement on line 30 is executed.

### 4.5 WATCHPOINTS

A watchpoint is a location that the debugger monitors so that it can inform you when your program has made an attempt to modify its contents. When you debug a PL/I program, you can set a watchpoint on a variable. When the watched variable is modified, the debugger suspends program execution, displays the address of the instruction, and prompts for a command.

Watchpoints are monitored continuously. You can determine, therefore, whether locations are being modified inadvertently during program execution.

You can use the following commands to control watchpoints:

- SET WATCH defines the location(s) to be monitored.
- SHOW WATCH displays the location(s) currently being monitored.
- CANCEL WATCH disables monitoring of the specified locations.

You can monitor only static scalar variables and array elements. Because automatic variables are allocated storage on the stack, they are protected from access. For example:

```
DBG>SET WATCH AREA
```

Note that you cannot set watchpoints, tracepoints, and breakpoints at the same location; the most recently issued command overrides the other(s).

Note that run-time errors occur if a watchpoint is in effect while I/O is being performed. Thus, to watch a variable, you must be careful not to set the watchpoint until all previous I/O is completed. You can do this by setting a breakpoint following an I/O statement and then setting a watchpoint. For example, if you want to watch a variable R in a procedure that contains a PUT statement on line 12, you could set the watchpoint as follows:

```
DBG>SET BREAK %LINE 13.1 DO (SET WATCH R;GO)
DBG>SET BREAK %LINE 12.1 DO (CANCEL WATCH R;GO)
```

#### NOTE

A bug in the BASIC support for the Release 2 Debugger requires you to use the syntax %line.l when you specify a DO in a SET BREAK command, as in this example.

The SET BREAK commands in the above example ensure that each time the PUT statement is about to execute, the watchpoint at R is canceled. Following the PUT statement, the watchpoint is reestablished.

## CONTROLLING A PROGRAM'S EXECUTION

When a watchpoint is reached, the debugger suspends execution and displays a message similar to the following:

```
write to MAINP\MAINP\R(1:6) at PC MAINP\MAINP %LINE 13 +25
      old value = +0000000000
      new value = +0054002700
DBG>
```

When a watched variable is modified, the debugger displays its former contents, if any, and the modified contents. It then prompts you to enter a command. You must enter GO or STEP to continue the program's execution.

### 4.6 ENTERING AND RETURNING FROM SUBROUTINES

As you debug a program that consists of more than one procedure, you can use the following to control the debugging:

- The STEP command lets you specify whether you want to debug a called subroutine or step over it.
- The SHOW CALLS command displays a traceback showing the calling sequence.
- The CALL command lets you invoke a subroutine and pass it arguments.

#### 4.6.1 Stepping Into and Over Subroutines

When you are stepping through a program, or when you have set a breakpoint at a statement that is a CALL statement, you can decide whether or not to enter the subroutine. To enter the subroutine, enter:

```
DBG>STEP/INTO
```

If the names declared in this module are not already in the debugger's symbol table, you must also enter a SET MODULE command to include the symbols (including line numbers) that you want to reference.

If you do not want to debug the subroutine, enter:

```
DBG>STEP/OVER
```

Then, the debugger continues the program's execution at the subroutine's entry point and returns control to you when the subroutine returns.

The STEP command also lets you decide whether you want to step through system routines, for example, PL/I run-time procedures or system services. If you specify STEP/SYSTEM, then the debugger will step through system routines for you. You cannot, however, set breakpoints or examine data that is being used by system procedures.

You can use the SET STEP command to set a default mode for stepping. For example:

```
DBG>SET STEP INTO
```

After this command, the debugger steps into all subroutines. Note, however, that the debugger steps into the PL/I run-time routines as well as into your subroutines.

#### 4.6.2 Displaying the Calling Sequence

The SHOW CALLS command produces a traceback of calls, and is particularly useful when you have returned to the debugger following a `CTRL/Y` interrupt.

The debugger displays a traceback list that shows you the sequence of calls leading to the current module. If you specify a value, for example

```
DBG>SHOW CALLS 6
```

the six most recent calls are displayed.

#### 4.6.3 Calling Subroutines

You can use the debugger command CALL to invoke an internal or external subroutine or function during the debugging session. You can also specify arguments using variables. For example, assume a program contains the following subroutine:

```
CALC: PROCEDURE (P,Q);
DECLARE (P,Q) FIXED;

      Q = P**P;
      END;
```

If you have variables X and Y declared as FIXED, you can test this subroutine as in the following examples:

```
DBG>DEPOSIT X = 5
DBG>CALL CALC (X,Y)
routine start at MAINP\MAINP\CALC
value returned is 1342195267
DBG>EXAMINE Y
MAINP\MAINP\Y: 3125
DBG>DEPOSIT X = 7
DBG>CALL CALC (X,Y)
routine start at MAINP\MAINP\CALC
value returned is 259017289
DBG>EXAMINE Y
MAINP\MAINP\Y: 823543
```

Note that when you specify arguments with the CALL command, you must use only variable names; the debugger cannot pass constants to PL/I procedures.

The debugger always displays a return value from the procedure that was invoked. Thus, if the procedure is a function, the actual return value will be displayed. However, if the procedure is a subroutine, as in this example, the returned value is meaningless.



APPENDIX A

VAX-11 PL/I RUN-TIME MODULES AND ENTRY POINTS

This appendix summarizes the modules and entry points in the VAX-11 PL/I run-time system. Table A-1 lists the modules in the library and summarizes the function(s) performed by each. Table A-2 lists the entry points, gives the name of the module in which the entry point is defined and summarizes the function performed by that entry. Table A-3 lists the modules from the VAX-11 Run-Time Procedure Library that are called by PL/I run-time modules.

Table A-1  
VAX-11 PL/I Run-Time Modules

Module	Function(s)
LIB\$EMULATE	Emulates G and H floating point
PLI\$\$BYTESIZE	Calculates the size of an item for an I/O operation
PLI\$\$ENVIR	Processes ENVIRONMENT options
PLI\$\$PROTVCHA	Converts system protection bits to character varying strings
PLI\$BIT	Performs bit manipulations
PLI\$CHAR	Performs character manipulations
PLI\$CLOSE	Closes files
PLI\$CONDIT	Performs default condition handling for MAIN procedures
PLI\$CONTROL	Processes main procedure startup and stopping, and performs exit handling
PLI\$CONVERT	Performs data conversions
PLI\$CVTPIC	Performs picture conversions and validation
PLI\$DATA	Contains run-time constants, the collating sequence, and tables
PLI\$DELETE	Performs the DELETE statement
PLI\$DIVIDE_PACKED_LONG	Performs extended precision division for precisions greater than or equal to 30

(Continued on next page)



Table A-1 (Cont)  
VAX-11 PL/I Run-Time Modules

Module	Function(s)
PLI\$DIVIDE_PACKED_SHORT	Performs extended precision division for precisions less than 30
PLI\$ERRORMSG	Constructs and displays error messages
PLI\$FORMAT	Processes format items
PLI\$GETBUFFER	Provides the file system interface for GET FILE statement
PLI\$GETEITEM	Performs GET EDIT operations
PLI\$GETFILE	Provides the program interface for GET FILE operations
PLI\$GETLISTITEM	Performs GET LIST operations
PLI\$HEEP	Obtains dynamic storage
PLI\$MATH	Performs mathematical functions
PLI\$OPEN	Opens files
PLI\$PUTBUFFER	Provides the file system interface for PUT FILE operations
PLI\$PUTEDITITEM	Performs the PUT EDIT statement
PLI\$PUTFILE	Provides the program interface for PUT FILE operations
PLI\$PUTLISTITEM	Performs the PUT LIST statement
PLI\$READ	Performs the READ statement
PLI\$RECOPT	Processes I/O options and keys
PLI\$REWRITE	Performs the REWRITE statement
PLI\$RMSBIS	Performs file-handling built-in functions
PLI\$STRINGIO	Provides the program interface for GET STRING
PLI\$TIME_DATE	Performs the DATE and TIME built-in functions
PLI\$WRITE	Performs the WRITE statement

Table A-2  
Run-Time Entry Points

Entry Point	Module	Performs the PL/I Function
PLI\$\$BYTESIZE	PLI\$\$BYTESIZE	Sizing of I/O item
PLI\$\$CHK_KEYCND	PLI\$RECOPT	Validation of key data type
PLI\$\$CHARBITN_R6	PLI\$GETBUFFER	Conversion of character to bit
PLI\$\$ENVIR	PLI\$\$ENVIR	ENVIRONMENT options
PLI\$\$FXCTLTO_R6	PLI\$RECOPT	FIXED_CONTROL_TO option
PLI\$\$FXDCTLFROM	PLI\$RECOPT	FIXED_CONTROL_FROM option
PLI\$\$GETFMT_R6	PLI\$FORMAT	GET EDIT format
PLI\$\$GETNEDI_R6	PLI\$GETBUFFER	GET EDIT format item
PLI\$\$GETNLIS_R6	PLI\$GETBUFFER	Next list item
PLI\$\$GETSKIP_R2	PLI\$GETBUFFER	SKIP option
PLI\$\$GETSKP1_R2	PLI\$GETBUFFER	SKIP option
PLI\$\$GET_REC	PLI\$GETBUFFER	Stream input
PLI\$\$KEYNUM	PLI\$RECOPT	INDEX_NUMBER option
PLI\$\$KEYTO_R8	PLI\$RECOPT	KEYTO option
PLI\$\$KEY_HND	PLI\$RECOPT	Key conversion errors
PLI\$\$MATCHGEQ	PLI\$RECOPT	MATCH_GREATER_EQUAL option
PLI\$\$MATCHGTR	PLI\$RECOPT	MATCH_GREATER option
PLI\$\$PROTVCHA	PLI\$\$PROTVCHA	Converts system protection bits to character varying strings
PLI\$\$PUTFMT_R6	PLI\$FORMAT	PUT EDIT format items
PLI\$\$PUTNEDI_R6	PLI\$PUTBUFFER	Next output edit item
PLI\$\$PUTNLIS_R6	PLI\$PUTBUFFER	Next output list item
PLI\$\$PUTPAGE_R6	PLI\$PUTBUFFER	PUT PAGE
PLI\$\$PUTSKP1_R2	PLI\$PUTBUFFER	PUT SKIP
PLI\$\$PUT_REC	PLI\$PUTBUFFER	PUT buffer
PLI\$\$READKEY_R6	PLI\$RECOPT	KEY option
PLI\$\$STREAM_HND	PLI\$CONDIT	Condition handling for stream I/O
PLI\$\$TERM_PROG	PLI\$CONTROL	Program termination

(Continued on next page)

VAX-11 PL/I RUN-TIME MODULES AND ENTRY POINTS

Table A-2 (Cont)  
Run-Time Entry Points

Entry Point	Module	Performs the PL/I Function
PLI\$VALRECIDTO	PLI\$RECOPT	RECORD_ID_TO
PLI\$WRITEKEY_R8	PLI\$RECOPT	KEYFROM option
PLI\$ABITABIT_R6	PLI\$CONVERT	Conversion of aligned bit to aligned bit
PLI\$ABITBIT_R6	PLI\$CONVERT	Conversion of aligned bit to unaligned bit
PLI\$ABITCHAR_R6	PLI\$CONVERT	Conversion of aligned bit to character
PLI\$ABITFIXB_R6	PLI\$CONVERT	Conversion of aligned bit to fixed binary
PLI\$ABITFIXD_R6	PLI\$CONVERT	Conversion of aligned bit to fixed decimal
PLI\$ABITFLTB_R6	PLI\$CONVERT	Conversion of aligned bit to floating binary
PLI\$ABITFLTD_R6	PLI\$CONVERT	Conversion of aligned bit to floating decimal
PLI\$ABITPIC_R6	PLI\$CONVERT	Conversion of aligned bit to picture
PLI\$ABITVCHA_R6	PLI\$CONVERT	Conversion of aligned bit to varying character
PLI\$AB_COLAT	PLI\$DATA	Collating table
PLI\$ALOCHEEP	PLI\$HEEP	Memory allocation
PLI\$ANDBIT	PLI\$BIT	AND bit strings
PLI\$BITABIT_R6	PLI\$CONVERT	Conversion of unaligned bit to aligned bit
PLI\$BITBIT_R6	PLI\$CONVERT	Conversion of unaligned bit to unaligned bit
PLI\$BITCHAR_R6	PLI\$CONVERT	Conversion of unaligned bit to character
PLI\$BITFIXB_R6	PLI\$CONVERT	Conversion of unaligned bit to fixed binary
PLI\$BITFIXD_R6	PLI\$CONVERT	Conversion of unaligned bit to fixed decimal
PLI\$BITFLTB_R6	PLI\$CONVERT	Conversion of unaligned bit to floating binary
PLI\$BITFLTD_R6	PLI\$CONVERT	Conversion of unaligned bit to floating decimal
PLI\$BITPIC_R6	PLI\$CONVERT	Conversion of unaligned bit to picture

(Continued on next page)

Table A-2 (Cont)  
Run-Time Entry Points

Entry Point	Module	Performs the PL/I Function
PLI\$BITVCHA_R6	PLI\$CONVERT	Conversion of unaligned bit to varying character
PLI\$BOOLBIT	PLI\$BITVERT	BOOL built-in function
PLI\$BOUND_CHECK	PLI\$CONDIT	Array bound checking
PLI\$B_PAC0	PLI\$DATA	Holds packed decimal constant
PLI\$B_PAC1	PLI\$DATA	Holds packed decimal constant
PLI\$B_PAC5	PLI\$DATA	Holds packed decimal constant
PLI\$B_PACN1	PLI\$DATA	Holds packed decimal constant
PLI\$B_SCAN	PLI\$DATA	Holds scan/span table
PLI\$CATBIT	PLI\$BIT	Bit concatenation
PLI\$CHARABIT_R6	PLI\$CONVERT	Conversion of character to aligned bit
PLI\$CHARBIT_R6	PLI\$CONVERT	Conversion of character to unaligned bit
PLI\$CHARCHAR_R6	PLI\$CONVERT	Conversion of character to character
PLI\$CHARFIXB_R6	PLI\$CONVERT	Conversion of character to fixed binary
PLI\$CHARFIXD_R6	PLI\$CONVERT	Conversion of character to fixed decimal
PLI\$CHARFLTB_R6	PLI\$CONVERT	Conversion of character to floating binary
PLI\$CHARFLTD_R6	PLI\$CONVERT	Conversion of character to floating decimal
PLI\$CHARPIC_R6	PLI\$CONVERT	Conversion of character to picture
PLI\$CHARVCHA_R6	PLI\$CONVERT	Conversion of character to varying character
PLI\$CLOSE	PLI\$CLOSE	CLOSE statement
PLI\$CMPBIT	PLI\$BIT	Bit comparisons
PLI\$CND_HND	PLI\$CONDIT	Condition handling for procedures without MAIN option

(Continued on next page)

Table A-2 (Cont)  
Run-Time Entry Points

Entry Point	Module	Performs the PL/I Function
PLI\$CNVRT_HND	PLI\$CONDIT	Condition handling for conversion errors
PLI\$CVRT_ANY	PLI\$CONVERT	All conversions
PLI\$CVRT_CG_R3	PLI\$CONVERT	All conversions
PLI\$CVT_FR_PIC	PLI\$CVTPIC	Conversions from pictures
PLI\$CVT_TO_PIC	PLI\$CVTPIC	Conversions to pictures
PLI\$DATE	PLI\$TIME_DATE	DATE built-in function
PLI\$DEF_HND	PLI\$CONDIT	Condition handling for MAIN procedures
PLI\$DELETE	PLI\$DELETE	DELETE statement
PLI\$DISPLAY	PLI\$RMSBIS	DISPLAY built-in subroutine
PLI\$DIV_PKSHORT	PLI\$DIVIDE- _PACKED_SHORT	Extended precision division
PLI\$DIV_PK_LONG	PLI\$DIVIDE- _PACKED_LONG	Extended precision division
PLI\$EXIT_HND	PLI\$CONTROL	Exit handling
PLI\$EXTEND	PLI\$RMSBIS	EXTEND built-in subroutine
PLI\$FCB_HEAD	PLI\$CONTROL	List of file headers
PLI\$FIXBABIT_R6	PLI\$CONVERT	Conversion of fixed binary to aligned bit
PLI\$FIXBBIT_R6	PLI\$CONVERT	Conversion of fixed binary to unaligned bit
PLI\$FIXBCHAR_R6	PLI\$CONVERT	Conversion of fixed binary to character
PLI\$FIXBFIxB_R6	PLI\$CONVERT	Conversion of fixed binary to fixed binary
PLI\$FIXBFIxD_R6	PLI\$CONVERT	Conversion of fixed binary to fixed decimal
PLI\$FIXBFLTB_R6	PLI\$CONVERT	Conversion of fixed binary to floating binary
PLI\$FIXBFLTD_R6	PLI\$CONVERT	Conversion of fixed binary to floating decimal
PLI\$FIXBPIC_R6	PLI\$CONVERT	Conversion of fixed binary to picture
PLI\$FIXBVCHA_R6	PLI\$CONVERT	Conversion of fixed binary to varying character
PLI\$FIXDABIT_R6	PLI\$CONVERT	Conversion of fixed decimal to aligned bit

(Continued on next page)

Table A-2 (Cont)  
Run-Time Entry Points

Entry Point	Module	Performs the PL/I Function
PLI\$FIXDBIT_R6	PLI\$CONVERT	Conversion of fixed decimal to unaligned bit
PLI\$FIXDCHAR_R6	PLI\$CONVERT	Conversion of fixed decimal to character
PLI\$FIXDFIXB_R6	PLI\$CONVERT	Conversion of fixed decimal to fixed binary
PLI\$FIXDFIXD_R6	PLI\$CONVERT	Conversion of fixed decimal to fixed decimal
PLI\$FIXDFLTB_R6	PLI\$CONVERT	Conversion of fixed decimal to floating binary
PLI\$FIXDFLTD_R6	PLI\$CONVERT	Conversion of fixed decimal to floating decimal
PLI\$FIXDPIC_R6	PLI\$CONVERT	Conversion of fixed decimal to picture
PLI\$FIXDVCHA_R6	PLI\$CONVERT	Conversion of fixed decimal to varying character
PLI\$FLTBABIT_R6	PLI\$CONVERT	Conversion of floating binary to aligned bit
PLI\$FLTBBIT_R6	PLI\$CONVERT	Conversion of floating binary to unaligned bit
PLI\$FLTBCHAR_R6	PLI\$CONVERT	Conversion of floating binary to character
PLI\$FLTBFIXB_R6	PLI\$CONVERT	Conversion of floating binary to fixed binary
PLI\$FLTBFIXD_R6	PLI\$CONVERT	Conversion of floating binary to fixed decimal
PLI\$FLTBFLT_B6	PLI\$CONVERT	Conversion of floating binary to floating binary
PLI\$FLTBFLTD_R6	PLI\$CONVERT	Conversion of floating binary to floating decimal
PLI\$FLTBPIC_R6	PLI\$CONVERT	Conversion of floating binary to picture
PLI\$FLTBVCHA_R6	PLI\$CONVERT	Conversion of floating binary to varying character
PLI\$FLTDABIT_R6	PLI\$CONVERT	Conversion of floating decimal to aligned bit
PLI\$FLTDBIT_R6	PLI\$CONVERT	Conversion of floating decimal to bit
PLI\$FLTDCHAR_R6	PLI\$CONVERT	Conversion of floating decimal to character

(Continued on next page)

Table A-2 (Cont)  
Run-Time Entry Points

Entry Point	Module	Performs the PL/I Function
PLI\$FLTDFIXB_R6	PLI\$CONVERT	Conversion of floating decimal to fixed binary
PLI\$FLTDFIXD_R6	PLI\$CONVERT	Conversion of floating decimal to fixed decimal
PLI\$FLTDFLTB_R6	PLI\$CONVERT	Conversion of floating decimal to floating binary
PLI\$FLTDFLTD_R6	PLI\$CONVERT	Conversion of floating decimal to floating decimal
PLI\$FLTDPIC_R6	PLI\$CONVERT	Conversion of floating decimal to picture
PLI\$FLTDVCHA_R6	PLI\$CONVERT	Conversion of floating decimal to varying character
PLI\$FLUSH	PLI\$RMSBIS	FLUSH built-in subroutine
PLI\$FREEHEEP	PLI\$HEEP	Virtual memory deallocation
PLI\$GETEABIT_R6	PLI\$GETEITEM	GET aligned bit item to edit
PLI\$GETEBIT_R6	PLI\$GETEITEM	GET EDIT of bit item
PLI\$GETECHAR_R6	PLI\$GETEITEM	GET EDIT of character item
PLI\$GETEFIXB_R6	PLI\$GETEITEM	GET EDIT of fixed binary item
PLI\$GETEFIXD_R6	PLI\$GETEITEM	GET EDIT of fixed decimal item
PLI\$GETEFLTB_R6	PLI\$GETEITEM	GET EDIT of floating binary item
PLI\$GETEFLTD_R6	PLI\$GETEITEM	GET EDIT of floating decimal item
PLI\$GETEPIC_R6	PLI\$GETEITEM	GET EDIT of pictured item
PLI\$GETEVCHA_R6	PLI\$GETEITEM	GET EDIT of varying character item
PLI\$GETFILE	PLI\$GETFILE	GET statement
PLI\$GETLABIT_R6	PLI\$GETLITEM	GET LIST of aligned bit item
PLI\$GETLBIT_R6	PLI\$GETLITEM	GET LIST of bit item
PLI\$GETLCHAR_R6	PLI\$GETLITEM	GET LIST of character item
PLI\$GETLFIXB_R6	PLI\$GETLITEM	GET LIST of fixed binary item
PLI\$GETLFIXD_R6	PLI\$GETLITEM	GET LIST of fixed decimal item
PLI\$GETLFLTB_R6	PLI\$GETLITEM	GET LIST of floating binary item

(Continued on next page)

VAX-11 PL/I RUN-TIME MODULES AND ENTRY POINTS

Table A-2 (Cont)  
Run-Time Entry Points

Entry Point	Module	Performs the PL/I Function
PLI\$GETLFLTD_R6	PLI\$GETLITEM	GET LIST of floating decimal item
PLI\$GETLPIC_R6	PLI\$GETLITEM	GET LIST of pictured item
PLI\$GETLVCHA	PLI\$GETLISTITEM	GET LIST of varying character item
PLI\$GETSTRNG_R6	PLI\$STRINGIO	GET STRING
PLI\$GOTO	PLI\$CONDIT	GOTO
PLI\$INDEXBIT	PLI\$BITDIT	INDEX built-in function for bits
PLI\$IO_ERROR	PLI\$CONDIT	I/O error messages
PLI\$LINK_FCB	PLI\$CONTROL	PLI\$OPEN Linkage of open file headers
PLI\$MOVBIT	PLI\$BIT	Bit copies
PLI\$MOVTRANCHAR	PLI\$CHAR	TRANSLATE built-in function
PLI\$NEXT_VOLUME	PLI\$RMSBIS	NXTVOL built-in subroutine
PLI\$NOLOC_GOTO	PLI\$CONDIT	Nonlocal GOTO
PLI\$NONLOC_RET	PLI\$CONDIT	Nonlocal RETURN
PLI\$NOTBIT	PLI\$BIT	NOT bits
PLI\$ONCNDARG	PLI\$CONDIT	ONARGSLIST built-in function
PLI\$ONCODE	PLI\$CONDIT	ONCODE built-in function
PLI\$ONFILE	PLI\$CONDIT	ONFILE built-in function
PLI\$ONKEY	PLI\$CONDIT	ONKEY built-in function
PLI\$OPEN	PLI\$OPEN	OPEN statement
PLI\$OPTIONSMAIN	PLI\$CONTROL	MAIN procedure initialization
PLI\$OPTMAIN_HND	PLI\$CONDIT	Condition handling for MAIN procedure
PLI\$OPTMAIN_RET	PLI\$CONDIT	RETURN from MAIN procedure
PLI\$ORBIT	PLI\$BIT	OR bits
PLI\$PICABIT_R6	PLI\$CONVERT	Conversion of picture to aligned bit
PLI\$PICBIT_R6	PLI\$CONVERT	Conversion of picture to unaligned bit
PLI\$PICCHAR_R6	PLI\$CONVERT	Conversion of picture to character

(Continued on next page)



Table A-2 (Cont)  
Run-Time Entry Points

Entry Point	Module	Performs the PL/I Function
PLI\$PICFIXB_R6	PLI\$CONVERT	Conversion of picture to fixed binary
PLI\$PICFIXD_R6	PLI\$CONVERT	Conversion of picture to fixed decimal
PLI\$PICFLTB_R6	PLI\$CONVERT	Conversion of picture to floating binary
PLI\$PICFLTD_R6	PLI\$CONVERT	Conversion of picture to floating decimal
PLI\$PICPIC_R6	PLI\$CONVERT	Conversion of picture to picture
PLI\$PICVCHA_R6	PLI\$CONVERT	Conversion of picture to varying character
PLI\$PUTEABIT_R6	PLI\$PUTEDITITEM	PUT EDIT of aligned bit item
PLI\$PUTEBIT_R6	PLI\$PUTEDITITEM	PUT EDIT of unaligned bit item
PLI\$PUTECHAR_R6	PLI\$PUTEDITITEM	PUT EDIT of character item
PLI\$PUTEFIXB_R6	PLI\$PUTEDITITEM	PUT EDIT of fixed binary item
PLI\$PUTEFIXD_R6	PLI\$PUTEDITITEM	PUT EDIT of fixed decimal item
PLI\$PUTEFLTB_R6	PLI\$PUTEDITITEM	PUT EDIT of floating binary item
PLI\$PUTEFLTD_R6	PLI\$PUTEDITITEM	PUT EDIT of floating decimal item
PLI\$PUTEPIC_R6	PLI\$PUTEDITITEM	PUT EDIT of picture item
PLI\$PUTEVCHA_R6	PLI\$PUTEDITITEM	PUT EDIT of varying character item
PLI\$PUTFILE	PLI\$PUTFILE	PUT FILE statement
PLI\$PUTLABIT_R6	PLI\$PUTLISTITEM	PUT LIST of aligned bit item
PLI\$PUTLBIT_R6	PLI\$PUTLISTITEM	PUT LIST of unaligned bit item
PLI\$PUTLCHAR_R6	PLI\$PUTLISTITEM	PUT LIST of character item
PLI\$PUTLFIXB_R6	PLI\$PUTLISTITEM	PUT LIST of fixed binary item
PLI\$PUTLFIXD_R6	PLI\$PUTLISTITEM	PUT LIST of fixed decimal item
PLI\$PUTLFLTB_R6	PLI\$PUTLISTITEM	PUT LIST of floating binary item
PLI\$PUTLFLTD_R6	PLI\$PUTLISTITEM	PUT LIST of floating decimal item
PLI\$PUTLPIC_R6	PLI\$PUTLISTITEM	PUT LIST of pictured item
PLI\$PUTLVCHA_R6	PLI\$PUTLISTITEM	PUT LIST of varying character item

(Continued on next page)

Table A-2 (Cont)  
Run-Time Entry Points

Entry Point	Module	Performs the PL/I Function
PLI\$PUTSTRNG_R6	PLI\$STRINGIO	PUT STRING statement
PLI\$PUT_END_R6	PLI\$PUTBUFFER	Flushing of PUT buffers
PLI\$READ	PLI\$READUFFER	READ statement
PLI\$RESIGNAL	PLI\$CONDIT	RESIGNAL built-in subroutine
PLI\$RETURN	PLI\$CONDIT	RETURN statement
PLI\$REWIND	PLI\$RMSBIS	REWIND built-in subroutine
PLI\$REWRITE	PLI\$REWRITE	REWRITE statement
PLI\$RT_SUBSCRIP	PLI\$CONDIT	Signaling of subscript range errors for uninitialized label arrays
PLI\$RVRT_CND	PLI\$CONDIT	REVERT statement
PLI\$SPACEBLOCK	PLI\$RMSBIS	SPACEBLOCK built-in subroutine
PLI\$STOP_PROG	PLI\$CONTROL	STOP statement
PLI\$TIME	PLI\$TIME_DATE	TIME built-in function
PLI\$VALID_PIC	PLI\$CVTPIC	VALID built-in function and picture validation
PLI\$VCHAABIT_R6	PLI\$CONVERT	Conversion of varying character to aligned bit
PLI\$VCHABIT_R6	PLI\$CONVERT	Conversion of varying character to unaligned bit
PLI\$VCHACHAR_R6	PLI\$CONVERT	Conversion of varying character to character
PLI\$VCHAFIXB_R6	PLI\$CONVERT	Conversion of varying character to fixed binary
PLI\$VCHAFIXD_R6	PLI\$CONVERT	Conversion of varying character to fixed decimal
PLI\$VCHAFLTB_R6	PLI\$CONVERT	Conversion of varying character to floating binary
PLI\$VCHAFLTD_R6	PLI\$CONVERT	Conversion of varying character to floating decimal
PLI\$VCHAPIC_R6	PLI\$CONVERT	Conversion of varying character to picture
PLI\$VCHAVCHA_R6	PLI\$CONVERT	Conversion of varying character to varying character
PLI\$VERIFY	PLI\$CHAR	VERIFY built-in function
PLI\$WRITE	PLI\$WRITE	WRITE statement

VAX-11 PL/I RUN-TIME MODULES AND ENTRY POINTS

Table A-3  
Run-Time Library Procedures Called by PL/I

Procedure	Function
LIB\$EMULATE	G and H floating-point emulation
LIB\$FREE_VM	Virtual memory deallocation
LIB\$GET_VM	Virtual memory allocation
LIB\$LP_LINES	Determine system default lines/page
LIB\$SIGNAL	Condition signaling

The VAX-11 PL/I mathematical built-in functions are performed by the VAX-11 run-time procedures listed below. These routines are all called by PLI\$MATH:

MTH\$\$JACKETHND	MTH\$DLOGR8	MTH\$HATAN2
MTH\$\$SIGNAL	MTH\$DSIND	MTH\$HATAND
MTH\$ALOG2	MTH\$DSINR7	MTH\$HATAND2
MTH\$ALOGR5	MTH\$DTAND	MTH\$HATANH
MTH\$ATAN2	MTH\$DTANR7	MTH\$HATANR8
MTH\$ATAND	MTH\$GATAN2	MTH\$HCOSD
MTH\$ATAND2	MTH\$GATAN2	MTH\$HCOSR5
MTH\$ATANH	MTH\$GATAND2	MTH\$HLOG2
MTH\$ATANR4	MTH\$GATANH	MTH\$HLOGR8
MTH\$COSD	MTH\$GATANR7	MTH\$HSIND
MTH\$COSR4	MTH\$GCOSD	MTH\$HSINR5
MTH\$DATAN2	MTH\$GCOSR7	MTH\$HTAND
MTH\$DATAND	MTH\$GLOG2	MTH\$HTANR5
MTH\$DATAND2	MTH\$GLOGR8	MTH\$KINVARGMAT
MTH\$DATANH	MTH\$GSIND	MTH\$\$SIND
MTH\$DATANR7	MTH\$GSINR7	MTH\$\$SINR4
MTH\$DCOSD	MTH\$GTAND	MTH\$TAND
MTH\$DCOSR7	MTH\$GTANR7	MTH\$TANR4
MTH\$DLOG2		

VAX-11 PL/I also calls run-time library modules that perform data conversion. The following modules are called by PLI\$CONVERT:

OTS\$\$CVT\_D\_T\_R8 OTS\$CVT\_T\_D  
 OTS\$\$CVT\_G\_T\_R8 OTS\$CVT\_T\_G  
 OTS\$\$CVT\_H\_T\_R8 OTS\$CVT\_T\_H  
 OTS\$CHARSTAR\_R6

The following routines are called by PLI\$FORMAT:

FOR\$CVT\_D\_TE  
 FOR\$CVT\_G\_TE  
 FOR\$CVT\_H\_TE

## INDEX

%INCLUDE statement, 1-2  
%LINE  
    set tracepoint, 4-5  
    specify breakpoint, 4-3  
    specify pathname, 2-6  
/ASCII qualifier, 3-4  
/DEBUG qualifier, 1-2  
/NODEBUG qualifier, 1-2  
@ command, 1-4

### A

Address expressions  
    how to specify, 2-4  
Addresses  
    determine virtual, 2-4  
Areas, 3-2  
Arguments  
    specify on CALL command, 4-7  
Arrays  
    automatic, 3-6  
    bit strings, 3-6  
    examine range of elements, 3-1  
    fixed-point decimal, 3-6  
    static, 3-5  
    variable extents, 3-2  
Automatic variables  
    examine and deposit, 3-3  
    in registers, 1-3  
    scope, 2-8

### B

Based variables, 3-2  
BASIC message, 1-2  
Bit-string variables, 3-5  
    arrays, 3-6  
Breakpoints, 4-3  
    at procedure entry points, 4-3  
    at statements, 4-3  
    continue execution, 4-3  
    restriction on setting, 4-5  
    set, 2-5  
    specify pathname, 2-6

### C

CALL command, 1-4, 4-6 to 4-7  
CANCEL ALL command, 1-4  
CANCEL BREAK command, 1-4, 4-3  
    example, 4-4  
CANCEL EXCEPTION BREAK command,  
    1-4  
CANCEL MODE command, 1-5

CANCEL MODULE command, 1-5, 2-2  
CANCEL SCOPE command, 1-5, 2-7  
CANCEL TRACE command, 1-5  
CANCEL TYPE/OVERRIDE command,  
    1-5  
CANCEL WATCH command, 1-5, 4-5  
Character strings  
    specify, 3-4  
    specify to the debugger, 2-3  
Character-string variables, 3-4  
Characters  
    recognized by debugger, 2-8  
Commands, debugger  
    summary, 1-4 to 1-9  
    syntax, 1-3  
CONTINUE command, 4-2  
CTRL/Y  
    interrupt program, 1-3  
    return to command level, 4-2  
Current location symbol, 2-9

### D

Data types  
    override declared, 3-1  
    restrictions, 3-2  
DEBUG command, 1-3, 4-2  
Debugger  
    compile and link with, 1-2  
    restart restriction, 4-1  
    stop, 4-2  
    summary of features, 1-1  
    symbol table, 2-1  
Debugger command summary, 1-4  
Default scope, 2-6 to 2-7  
DEFINE command, 1-5, 2-5, 2-7  
Defined variables, 3-2  
DEPOSIT command, 1-5, 3-1  
    specify current location, 2-9  
Disjoint registers, 1-3

### E

Entry names  
    specify to the debugger, 2-4  
Entry points  
    PL/I run-time, Appendix A  
    set breakpoints, 4-3  
    set tracepoints, 4-4  
EVALUATE command, 1-5  
    determine virtual address, 2-4  
EXAMINE command, 1-6, 3-1  
    examine previous location, 2-9  
    specify data type, 3-1  
    specify pathname, 2-6

## INDEX

EXIT command, 1-6, 4-1  
External variables  
  references, 2-3

### F

File data, 3-2  
Fixed-point binary variables,  
  3-4  
Fixed-point decimal arrays, 3-6  
Fixed-point decimal variables,  
  3-4  
Floating-point variables, 3-4  
Formats, 3-2  
Functions  
  invoke, 4-7

### G

Global symbols, 2-4  
GO command, 1-6, 4-1  
  after breakpoint, 4-3

### H

HELP command, 1-6

### I

INCLUDE files  
  print in listing, 1-2  
Internal variables  
  references, 2-3

### L

Labels, 3-2  
Level-one procedure, 2-1  
Line numbers  
  specify breakpoints, 4-3  
  specify to the debugger, 2-4  
  stepping, 4-2  
LINK command  
  link with debugger, 1-2  
Listing (compiler), 1-2

### M

Modes  
  stepping, 4-2  
Module name  
  displayed by debugger, 1-2

Modules  
  in debugger symbol table, 2-1  
  in image file, 2-1  
  in symbol table  
    list, 2-2  
  PL/I run-time, Appendix A

### N

Names  
  add to symbol table, 2-2  
  how to specify, 2-1  
  scope, 2-5  
Numeric constants  
  specify to debugger, 2-3

### O

Optimization  
  effect on debugging, 1-3  
Override  
  declared data types, 3-1

### P

Parameters, 3-2  
Pathnames, 2-6  
  specify %LINE, 2-6  
PC scope, 2-6 to 2-7  
Permanent symbols, 2-5  
Pictures, 3-2  
PLI command  
  compile with debugger, 1-2  
Previous location symbol, 2-9  
Procedures  
  invoke, 4-7  
  specify arguments, 4-7  
  specify to the debugger, 2-4  
Program locations  
  how to specify, 2-4

### R

References  
  ambiguous, 2-5  
  external variables, 2-3  
  internal variables, 2-3  
Registers  
  automatic variables in, 1-3  
  reference, 2-5  
Resolution of references, 2-5  
Restart a program, 4-1  
Restrictions  
  data that cannot be examined, 3-2

## INDEX

RUN command, 1-2, 4-2  
Run-time modules, Appendix A

### S

Sample terminal session, 1-9  
Scope, 2-5, 2-8  
    automatic variables, 2-8  
    changing, 2-7  
SET BREAK command, 1-6, 4-3  
    /AFTER, 4-3  
    examples, 2-4 to 2-5, 4-5  
    with DO specification, 4-4  
SET EXCEPTION BREAK command, 1-6  
SET LANGUAGE command, 1-6  
SET LOG command, 1-7  
SET MODE command, 1-7  
SET MODULE command, 1-7, 2-2  
    performed by SET SCOPE, 2-7  
SET OUTPUT command, 1-7  
SET SCOPE command, 1-7, 2-7  
    effect on symbol table, 2-7  
SET STEP command, 1-7, 4-2  
SET TRACE command, 1-7, 4-4  
SET TYPE command, 1-8  
SET TYPE/OVERRIDE command, 3-1  
SET WATCH command, 1-8, 4-5  
SHOW BREAK command, 1-8, 4-3  
SHOW CALLS command, 1-8, 4-7  
    display calls, 4-6  
SHOW LANGUAGE command, 1-8  
SHOW LOG command, 1-8  
SHOW MODE command, 1-8  
SHOW MODULE command, 1-8, 2-2  
SHOW OUTPUT command, 1-8  
SHOW SCOPE command, 1-8, 2-7  
SHOW STEP command, 1-8  
SHOW TRACE command, 1-8, 4-4  
SHOW TYPE command, 1-8  
SHOW WATCH command, 1-9, 4-5  
Statements  
    execute singly, 4-2  
    set tracepoints, 4-4  
    suspend program execution at,  
        4-3  
STEP command, 1-9, 4-2  
    SET STEP INSTRUCTION, 4-2  
    step into a subroutine, 4-6  
    STEP/INTO, 4-6  
    STEP/LINE, 4-3  
    STEP/OVER, 4-6  
Stepping, 4-2  
    modes, 4-2  
Storage classes  
    restrictions, 3-2

Storage map, 1-2  
Structures, 3-2  
Subroutines  
    invoking, 4-6  
Symbol table  
    add names, 2-2  
    debugger, 2-1  
    display modules in, 2-2  
    effect of SET SCOPE command,  
        2-7  
    names included in, 2-1  
Symbolic references  
    define names for addresses,  
        2-5  
Symbols  
    accessible, 2-1  
    debugger permanent, 2-5

### T

Traceback  
    of active calls, 4-7  
Tracepoints, 4-4  
    at procedure entry points, 4-4  
    restriction on setting, 4-5

### V

Variables  
    arrays  
        automatic, 3-6  
        static, 3-5 to 3-6  
    bit-string, 3-5  
    character strings, 3-4  
    display  
        at breakpoint, 4-4  
    display contents, 3-1  
    examine and deposit, 3-1  
    fixed-point binary, 3-4  
    fixed-point decimal, 3-4  
    floating-point, 3-4  
    in storage map, 1-2  
    modify contents, 3-1  
    variable extents, 3-2  
Virtual address  
    determine, 2-4  
Virtual addresses  
    specify to the debugger, 2-4

### W

Watchpoints, 4-5  
    restriction on setting, 4-5



READER'S COMMENTS

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

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

---

---

---

---

---

---

---

---

---

---

---

---

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

---

---

---

---

---

---

---

---

---

---

---

---

Please indicate the type of 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

BSSG PUBLICATIONS ZK1-3/J3-5  
DIGITAL EQUIPMENT CORPORATION  
110 SPIT BROOK ROAD  
NASHUA, NEW HAMPSHIRE 03062



Do Not Tear - Fold Here