digital

VAX–11 PASCAL
User's Guide

Order No. AA-H485B-TE

VAX11

March 1981

This document describes how to compile, link, execute, and debug VAX-11 PASCAL programs on the VAX/VMS operating system. It also contains information useful to VAX-11 PASCAL programmers, dealing with input and output, procedure calling, error processing, and storage allocation.

VAX-11 PASCAL
User's Guide

Order No. AA-H485B-TE

SUPERSESSION/UPDATE INFORMATION: This revised document supersedes the VAX-11 PASCAL User's Guide (Order No. AA-H485A-TE)

SOFTWARE VERSION: VAX-11 PASCAL V1.2

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, November 1979
Revised, March 1981

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:

DEC	DECsystem-10	PDT
DECUS	DECSYSTEM-20	RSTS
DIGITAL	DECwriter	RSX
PDP	DIBOL	VMS
UNIBUS	Edusystem	VT
VAX	IAS	digital
DECnet	MASSBUS	

ZKA25-81

3/81-15

CONTENTS

CONTENTS

CONTENTS

CONTENTS

FIGURES

CONTENTS

TABLES

PREFACE

MANUAL OBJECTIVES

The VAX-11 PASCAL User's Guide is intended for use in developing and debugging new PASCAL programs, and in compiling and executing existing PASCAL programs on VAX/VMS systems. PASCAL language elements supported on VAX/VMS are described in the VAX-11 PASCAL Language Reference Manual.

INTENDED AUDIENCE

This manual is designed for programmers who have a working knowledge of PASCAL. Detailed knowledge of VAX/VMS is helpful but not essential; familiarity with the VAX/VMS Primer is recommended. Some sections of this book, however, (condition handling, for instance) require more extensive understanding of the operating system. In such sections, you are directed to the appropriate manual(s) for the required additional information.

STRUCTURE OF THIS DOCUMENT

This manual is organized as follows:

- Chapter 1 provides an overview of the steps you must follow to create, compile, link, and execute a VAX-11 PASCAL program.
- Chapter 2 describes how to compile your program and explains the options available at compile time.
- Chapter 3 supplies information on the VAX-11 Linker and its options, as they apply to PASCAL programs.
- Chapter 4 describes the commands used to run a program.
- Chapter 5 describes how to use the VAX-11 Symbolic Debugger.
- Chapter 6 provides information about PASCAL input/output, including details on the use of logical names, file conventions, and record structure.
- Chapter 7 discusses the conventions followed in calling procedures, especially the conventions for passing parameters.
- Chapter 8 describes error processing, in particular, how to use the condition handling facility. This chapter is intended for users with in-depth knowledge of VAX/VMS.

- Chapter 9 describes the relationship between VAX-11 PASCAL and the VAX/VMS operating system, with particular emphasis on program section usage, storage allocation, and data representation.
- Appendix A summarizes diagnostic messages.
- Appendix B illustrates the contents of the run-time stack during procedure calls.
- Appendix C summarizes in alphabetical order all the debugger commands.
- Appendix D summarizes in alphabetical order all the debugger and PASCAL specific error messages.

ASSOCIATED DOCUMENTS

The following documents are relevant to VAX-11 PASCAL programming:

- VAX/VMS Primer
- VAX-11 PASCAL Primer
- VAX-11 PASCAL Language Reference Manual
- VAX/VMS Command Language User's Guide
- VAX-11 Run-Time Library Reference Manual
- VAX-11 Linker Reference Manual
- VAX/VMS System Services Reference Manual
- VAX-11 Architecture Handbook
- VAX-11 PASCAL Installation Guide/Release Notes

For a complete list of VAX-11 software documents, see the VAX-11 Information Directory and Index.

CONVENTIONS USED IN THIS DOCUMENT

This document uses the following conventions.

Convention	Meaning
{ }	Braces enclose lists from which you must choose one item, for example: {expr {statement}
...	A horizontal ellipsis means that the preceding item can be repeated as indicated, for example: filename, ...

Convention	Meaning
. . .	A vertical ellipsis means that not all of the statements in a figure or example are shown.
[[ ]]	Double brackets in statement format descriptions enclose items that are optional, for example: [[PACKED]] Double brackets in statement and declaration format description enclose items that are optional, for example: WRITE ([[OUTPUT,]] print list)
[ ]	Square brackets mean that the statement syntax requires the square bracket characters. This notation is used with arrays and sets, for example: ARRAY [subscriptl]
items in uppercase letters and special symbols	Uppercase letters and special symbols in format descriptions indicate PASCAL reserved words that you must not abbreviate, for example: BEGIN END
items in lowercase letters	Lowercase letters represent elements that you must replace according to the description in the text.
$ PASCAL $_File:	In examples of commands you enter and system responses, all output lines and prompting characters that the system prints or displays are shown in black letters. All the lines you type are shown in red letters.
RET	A symbol with a 1- to 3-character abbreviation indicates that you press a key on the terminal, for example, RET.

SUMMARY OF TECHNICAL CHANGES

This section summarizes the technical changes made to the VAX-11 PASCAL compiler for Version 1.2.

- Debugger records are now generated to allow for full use of the VAX-11 Symbolic Debugger.
- An enhanced machine-code listing is now generated.
- Nonabortive run-time errors are signaled when issued.

CHAPTER 1

USING VAX-11 PASCAL

VAX-11 PASCAL is an extended implementation of the PASCAL language. The VAX-11 PASCAL compiler executes in native mode under the VAX/VMS operating system.

This manual describes how you interact with the VAX/VMS operating system using VAX-11 PASCAL. It contains instructions for compiling, linking, executing, and debugging a PASCAL program, and provides information on the following topics:

- Performing input and output operations
- Calling VAX/VMS system services and Run-Time Library routines
- Using the VAX/VMS condition handling facility
- Writing efficient VAX-11 PASCAL programs

This chapter provides an overview of the steps in creating and executing a VAX-11 PASCAL program. It also describes the standard VAX/VMS file specification and defaults.

1.1 CREATING AND EXECUTING A PROGRAM

Figure 1-1 illustrates the program development process, from inception to execution. You specify the steps shown in Figure 1-1 by entering commands strings to the VAX/VMS operating system.

```
$ EDIT file-spec (RET)
$ PASCAL file-spec (RET)
$ LINK file-spec (RET)
$ RUN file-spec (RET)
```

With each command, you include information that further defines what you want the system to do. Of prime importance is the file specification, indicating the file to be processed. You can also specify qualifiers that modify the processing performed by the system. Each command string is terminated by pressing the RETURN key.

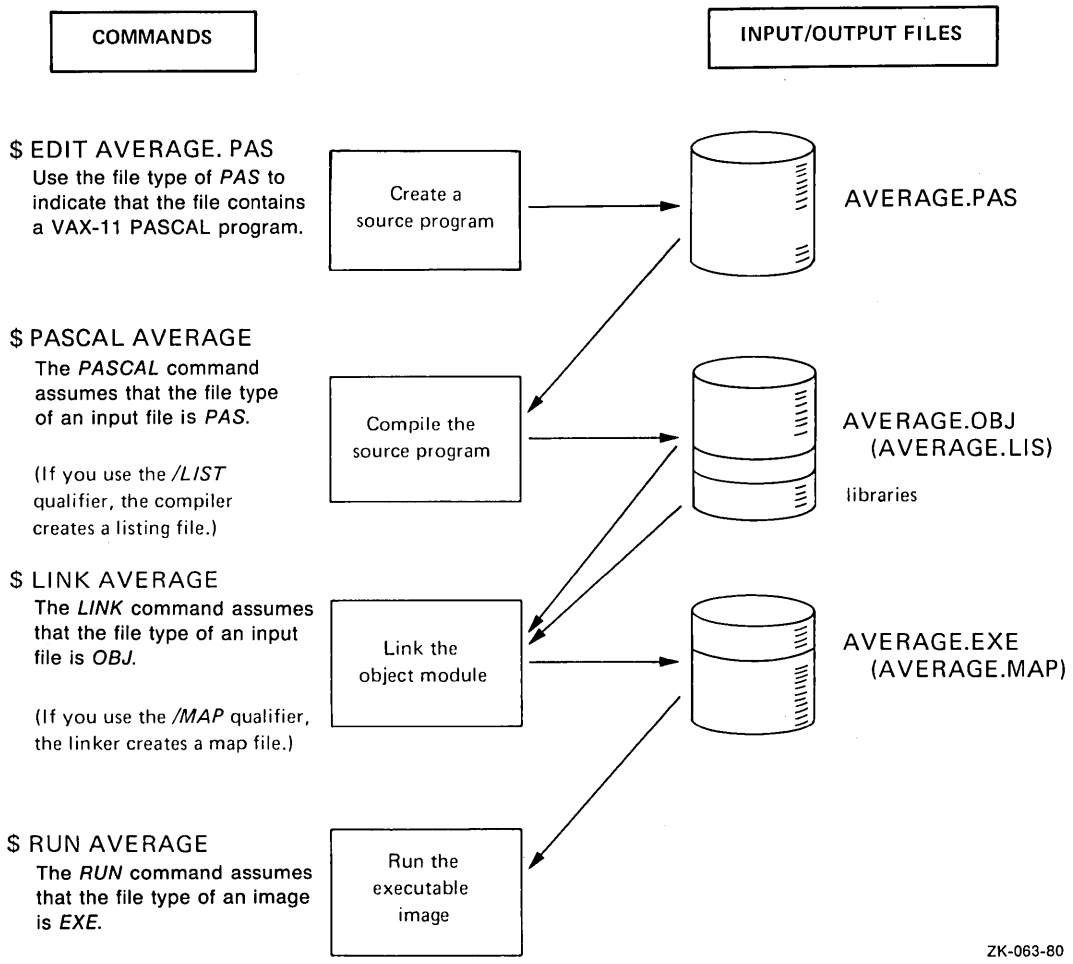


Figure 1-1 Program Development Process

1.2 VAX/VMS FILE SPECIFICATIONS AND DEFAULTS

A VAX/VMS file specification indicates the input file to be processed or the output file to be produced. File specifications have the following form:

```
node::device:[directory]filename.type;version
```

The punctuation marks (colons, brackets, period, semicolon) are required syntax that separate the various components of the file specification.

node

Specifies a network node name. This is applicable only to systems that support DECnet-VAX.

device

Identifies the device on which the file is stored or is to be written.

directory

Identifies the name of the directory under which the file is cataloged, on the device specified. You can delimit the directory name with square brackets, as shown, or with angle brackets (< >).

filename

Identifies the file by its name; filename can be up to 9 alphanumeric characters long.

type

Describes the kind of data in the file; type can be up to 3 alphanumeric characters long.

version

Specifies which version of the file is desired. Versions are identified by a decimal number, which is incremented by 1 each time a new version of a file is created. Either a semicolon or a period can be used to separate type and version.

You need not explicitly state all elements of a file specification each time you compile, link, or execute a program. The only part of the file specification that is usually required is the file name. If you omit any other part of the file specification, a default value is used. Table 1-1 summarizes the default values.

Table 1-1
File Specification Defaults

Optional Element	Default Value
node	Local network node
device	User's current default device
directory	User's current default directory
filetype	Depends on usage: Input to PASCAL compiler PAS Output from PASCAL compiler OBJ Input to linker OBJ Output from linker EXE Input to RUN command EXE Compiler source listing LIS Linker map listing MAP Input to executing program DAT Output from executing program DAT
version	Input: highest existing version Output: highest existing version plus 1

If you request compilation of a PASCAL program and you specify only a file name, the compiler can process the source program if it finds a file with the specified file name that:

- Is stored on the default device
- Is cataloged under the default directory name
- Has a file type of PAS

If more than one file meets these conditions, the compiler chooses the one with the highest version number.

For example, assume that your default device is DBA0, your default directory is SMITH, and you supply the following file specification to the compiler:

```
$ PASCAL (RET)
$_File: Circle (RET)
```

The compiler will search device DBA0 in directory SMITH, seeking the highest verion of Circle.PAS. If you do not explicitly specify an object file, the compiler will generate the file Circle.OBJ, store it on device DBA0 in directory SMITH, and assign it a version number 1 higher than any other version of Circle.OBJ currently cataloged in directory SMITH on DBA0.

CHAPTER 2

COMPILING A PROGRAM

After creating a VAX-11 PASCAL source program, you compile it. At compile time, you specify the name of the file(s) containing the source code and indicate which qualifiers you want to use.

At your option, the compiler produces one or more object files, which can be input to the linker (see Chapter 3), and one or more listing files. The listing files contain source and object code listings, information about compilation errors, and optional items such as cross-reference listings.

2.1 THE PASCAL COMMAND

To compile a source program, use the PASCAL command in the following form:

PASCAL [/command-qualifier(s)] file-spec-list [/file-qualifier(s)]

/command-qualifier(s)

Indicates special processing is to be performed by the compiler on all files being compiled (see Section 2.2).

file-spec-list

Specifies the source file(s) containing the program or module to be compiled. You can specify more than one source file. If source file specifications are separated by commas, the programs are compiled separately. If source file specifications are separated by plus signs, the files are concatenated and compiled as one program.

/file-qualifier(s)

Indicates special processing is to be performed by the compiler on the file(s) in the file-spec-list.

In interactive mode, you can also enter the file specification on a separate line by pressing the RETURN key after you type PASCAL. The system responds with a prompt for the file specification:

```
$ PASCAL RET
$_File:
```

Type the file specification and any file qualifiers immediately after the $_File: prompt.

2.2 PASCAL COMPILER QUALIFIERS

In many cases, the simplest form of the PASCAL command is sufficient for compilation. Sometimes, however, you will need to use qualifiers to specify special processing. Table 2-1 lists the qualifiers you can use with the VAX-11 PASCAL compiler. You specify the qualifiers on the command line. Some qualifiers may be specified in source code comments. This section describes the effect of each qualifier on a PASCAL program. Section 2.2.1 describes how to specify command line qualifiers. Section 2.2.2 deals with specifying qualifiers in source code comments.

Table 2-1
PASCAL Compiler Qualifiers

Qualifier	Purpose	Can Be Specified in Source Code?
CHECK	Generates code to perform run-time checks	Yes
CROSS_REFERENCE	Produces a cross-reference listing of identifiers	Yes
DEBUG	Generates records for VAX-11 Symbolic Debugger	Yes
ERROR_LIMIT	Terminates compilation after 30 errors	No
LIST	Produces source listing file	Yes
MACHINE_CODE	Includes representation of machine code in the source listing file	Yes
OBJECT	Specifies name of object file	No
STANDARD	Prints messages indicating use of PASCAL extensions	Yes
WARNINGS	Prints diagnostics for warning-level errors	Yes

CHECK

The CHECK qualifier directs the compiler to generate code to perform run-time checks. This code checks for invalid assignments to sets and subranges, and out-of-range array bounds and case labels. The system issues an error message and normally terminates execution if any of these conditions occur.

When this qualifier is disabled, the compiler generates no check code. By default, CHECK is disabled.

CROSS_REFERENCE

The CROSS_REFERENCE qualifier produces a cross-reference listing of all identifiers. The compiler generates separate cross-references for each procedure and function. To get complete cross-reference listings for a program, the qualifier must be in effect for all modules of the program. This qualifier is ignored if no listing file is being generated.

By default, CROSS_REFERENCE is disabled.

You can specify this qualifier in the source code, as described in Section 2.2.2. Note, however, that the cross-reference listing for a portion of a procedure or function may be incomplete.

DEBUG

The DEBUG qualifier specifies that the compiler is to generate information for use by the VAX-11 Symbolic Debugger and the run-time error traceback mechanism. When you enable the option, the compiler generates the debugger and Traceback records for each procedure or program for which the qualifier is in effect.

When this qualifier is disabled, the compiler generates only Traceback records. By default, the debugger is disabled.

Refer to Chapter 5 for more information on the debugger.

ERROR_LIMIT

The ERROR_LIMIT qualifier terminates compilation after 30 errors, excluding warning-level errors. If this qualifier is disabled, compilation continues through the entire unit. You cannot specify this qualifier in the source code.

By default, ERROR_LIMIT is enabled.

Note that after it finds 20 errors (including warning messages) on any one source line, the compiler generates the error code 255 -- too many errors on this source line. Compilation of the line continues, but no further error messages are printed for that line.

LIST

The LIST qualifier produces a source listing file. It has the form:

```
/LIST [=file-spec]
```

If you omit the file specification, the listing file defaults to the name of the first source file, your default directory, and a file type of LIS.

The compiler does not produce a listing file in interactive mode unless you specify the LIST qualifier. In batch mode, the compiler produces a listing file by default. In either case, the listing file is not automatically printed. You must use the DIGITAL Command Language (DCL) PRINT command to obtain a line printer copy of the listing file. A sample listing is explained in Section 2.4.

MACHINE_CODE

The MACHINE_CODE qualifier places in the listing file a representation of the object code generated by the compiler.

The compiler ignores this qualifier if the LIST qualifier is not enabled.

By default, MACHINE_CODE is disabled.

OBJECT

The OBJECT qualifier can be used when you want to specify the name of the object file. It has the form:

```
/OBJECT [=file-spec]
```

If you omit the file specification, the object file defaults to the name of the first source file, the default directory, and a file type of OBJ. You cannot specify this qualifier in the source code.

You can disable this qualifier to suppress object code; for example, when you only want to test the source program for compilation errors.

By default, OBJECT is enabled.

STANDARD

The STANDARD qualifier tells the compiler to print warning-level messages wherever the program uses "nonstandard" PASCAL features.

Nonstandard PASCAL features are the extensions to the PASCAL language that are incorporated in VAX-11 PASCAL. Nonstandard features include VALUE declarations and the exponentiation operator. Refer to the VAX-11 PASCAL Language Reference Manual for a list of all the extensions.

By default, STANDARD is enabled.

WARNINGS

The WARNINGS qualifier directs the compiler to generate diagnostic messages in response to warning-level (W) errors.

By default, WARNINGS is enabled. A warning diagnostic message indicates that the compiler has detected acceptable but unorthodox syntax or has performed some corrective action; in either case, unexpected results may occur. To suppress warning diagnostic messages, disable this qualifier. Note that messages generated when the STANDARD qualifier is enabled appear even if WARNINGS is disabled.

Appendix A lists the compiler diagnostic messages.

2.2.1 Specifying Qualifiers with the PASCAL Command

A PASCAL command qualifier has the form:

```
/qualifier [=file-spec]
```

Table 2-2 lists the qualifiers, and their negative form that you can use on the PASCAL command line. The optional file specification indicates the name of an output file for the /OBJECT and /LIST

qualifiers only. To enable the qualifier, specify its name. To disable the qualifier, specify the negative form.

You can abbreviate all command line qualifiers by truncating them on the right. All qualifiers are unique when truncated to their first four characters, not including the NO of the negative form. You can truncate further as long as the resulting qualifier is unique. For example, you can truncate /CROSS_REFERENCE to /CR, /CHECK to /CH, and /DEBUG to /D.

It is recommended that you use the full qualifier names in command procedure files, to ensure readability. To guarantee compatibility with future releases of the system, you should not abbreviate qualifiers in command procedures to fewer than four characters.

Table 2-2
PASCAL Command Qualifiers

Qualifier	Negative Form	Default
/CHECK	/NOCHECK	/NOCHECK
/CROSS_REFERENCE	/NOCROSS_REFERENCE	/NOCROSS_REFERENCE
/DEBUG	/NODEBUG	/NODEBUG
/ERROR_LIMIT	/NOERROR_LIMIT	/ERROR_LIMIT
/LIST [=file-spec]	/NOLIST	/NOLIST (interactive) /LIST (batch)
/MACHINE_CODE	/NOMACHINE_CODE	/NOMACHINE_CODE
/OBJECT [=file-spec]	/NOOBJECT	/OBJECT
/STANDARD	/NOSTANDARD	/STANDARD
/WARNINGS	/NOWARNINGS	/WARNINGS

Examples

1. $ PASCAL Calc

   The source file Calc.PAS is compiled. By default, the /OBJECT, /STANDARD, /WARNING, and /ERROR_LIMIT qualifiers are enabled.

2. $ PASCAL/CHECK/NOSTANDARD Calc

   The source file Calc.PAS is compiled, and check code is generated. The compiler does not issue warnings for the use of language extensions.

3. $ PASCAL/LIST/CR Calc

   The source file Calc.PAS is compiled and the listing file Calc.LIS is generated. The listing file includes a cross-reference listing.

2.2.2 Specifying Qualifiers in the Source Code

You can use qualifiers in the source code to enable and disable special processing during compilation. When specified in the source code, qualifiers have the form:

(*$qualifier {+|-} [[,qualifier {+|-} ,...]] ;comment*)

The first character after the comment delimiter must be a dollar sign ($); the dollar sign cannot be preceded by a space. Table 2-3 lists the qualifiers you can specify in your source program. Note that you can optionally use a 1-character abbreviation for each qualifier. The abbreviation is simply the first character of the qualifier name, except for CROSS_REFERENCE, whose abbreviation is X.

Table 2-3
Source Code Qualifiers

Qualifier	Abbreviation
CHECK	C
CROSS_REFERENCE	X
DEBUG	D
LIST	L
MACHINE_CODE	M
STANDARD	S
WARNINGS	W

To enable a qualifier, specify a plus sign (+) after its name or abbreviation. To disable a qualifier, specify a minus sign (-) after its name or abbreviation. You can specify any number of qualifiers. You can also include a text comment after the qualifiers, separated from the list of qualifiers by a semicolon.

When specified in the source code, the LIST qualifier cannot contain a file specification. The listing file will have the default specification as described in Section 2.2 above.

For example, to generate check code for only one procedure in a program, enable the CHECK qualifier before the procedure declaration and disable it at the end of the procedure, as follows:

```
(*$C+; enable CHECK for Test1 only*)
PROCEDURE Test1;
.
.
.
END;
(*$C-;disable CHECK*)
```

Command line qualifiers override source code qualifiers. If, for example, the source code specifies DEBUG+, but you enter PASCAL/NODEBUG, the DEBUG option will not be in effect.

2.3 SPECIFYING OUTPUT FILES

The compiler produces object files and listing files. You can control the production of these files by using the /LIST and /OBJECT qualifiers with the PASCAL command. Unless you specify otherwise, the compiler generates an object file. In interactive mode, the compiler, by default, does not generate a listing file; you must use the /LIST qualifier to explicitly specify a listing file. In batch mode, however, the opposite is true: by default, the compiler produces a listing file. To suppress the listing file, you specify the /NOLIST qualifier.

During the early stages of program development, you may find it helpful to suppress the production of object files until your source program compiles without errors. To do so, specify the NOOBJECT qualifier on the PASCAL command line. If you do not specify /NOOBJECT, the compiler generates object files as follows:

- If you specify one source file, one object file is generated.
- If you specify multiple source files, separated by plus signs, the source files are concatenated and compiled, and one object file is generated.
- If you specify multiple source files, separated by commas, each source file is compiled separately, and an object file is generated for each source file.
- You can use both plus signs and commas in the same command line to produce different combinations of concatenated and separate object files (see Example 4 below).

To produce an object file with an explicit file specification, you must specify /OBJECT on the PASCAL command line (see Section 2.2). Otherwise, the object file will have the name of its corresponding source file and a file type of OBJ. By default, the object file produced from concatenated source files has the name of the first source file. All other file specification attributes (node, device, directory, and version) assume the default attributes.

Examples

1.
```
$ PASCAL/LIST A, B, C
```

   Source files A.PAS, B.PAS, and C.PAS are compiled as separate files, producing object files named A.OBJ, B.OBJ, and C.OBJ; and listing files named A.LIS, B.LIS, and C.LIS.

2.
```
$ PASCAL X + Y + Z
```

   Source files X.PAS, Y.PAS, and Z.PAS are concatenated and compiled as one file, producing an object file named X.OBJ. In batch mode, this command also produces the listing file X.LIS.

3.
```
$ PASCAL/OBJECT=Square <RET>
$_File: Circle
```

   The system issues the $_File: prompt because the PASCAL command does not specify a source file. The file Circle.PAS is compiled, producing an object file named Square.OBJ, but no listing file. (This example applies to interactive mode only.)

4. $ PASCAL A + B/LIST,C

   Two object files are produced: A.PAS and B.PAS are concatenated to become A.OBJ and C.PAS becomes C.OBJ. In interactive mode, this command produces the listing file A.LIS. In batch mode, it produces two listing files: A.LIS and C.LIS.

5. $ PASCAL A + Circ/NOOBJECT + X

   When you include a qualifier in a list of files that are to be concatenated, the qualifier affects all files in the list. Thus, the command shown above completely suppresses the object file. That is, source files A.PAS, Circ.PAS, and X.PAS are concatenated and compiled, but no object file is produced.

6. $ PASCAL/LIST [DIR]M

   The source file M.PAS in directory [DIR] is compiled, producing an object file named M.OBJ and a listing file named M.LIS. The compiler places the object and listing files in the default directory.

2.4 COMPILER LISTING FORMAT

This section explains the format of the compiler listing.

Each section of a compiler listing is shown and described in detail. A complete compiler listing is illustrated in Section 2.4.4.

The compiler listing contains the following three sections:

- Source code listing -- When you specify the /LIST qualifier, the source code is listed by default.
- Cross-reference listing -- To generate cross references for all identifiers used in the program, you must specify the /CROSS_REFERENCE and /LIST qualifiers.
- Machine code listing -- To generate the machine code listing, you must specify the /MACHINE_CODE and /LIST qualifiers.

The numbers in the section below are keyed to the circled numbers in the examples that appear above each discussion and to the complete listing in Section 2.4.4.

Title line -- Each page of the compiler listing contains a title line.


```
EXAMPLE 15-AUG-1980 13:47:41    VAX-11 PASCAL V1.2-80                Page 1
```

The title line consists of four parts:

❶ The module name: EXAMPLE

❷ The date (day, month, year) and time (hour, minute, second) of compilation: 15-AUG-1980 13:26:37

❸ The PASCAL compiler name and version number: VAX-11 PASCAL V1.2

❹ The page number: Page 1

2.4.1 Source Code Listing

Each page of the source code listing contains a line under the title line.

```
❺      ❻                        ❼                    ❽
01 SOURCE LISTING     19-JUN-1980 14:10:08     _DBA1:[SMITH]EXAMPLE.PAS;2(1)
```

❺ The Module Identifier: 01

❻ The subtitle describing the listing:

SOURCE LISTING or (CROSS REFERENCE or GENERATED CODE)

❼ The date (day, month, year) and time (hour, minute, second) of source file creation: 19-JUNE-1980 14:10:08

❽ The VAX/VMS file specification of the source file:

_DBA1:[SMITH]EXAMPLE.PAS;2 (1)

Source Code Listing

```
  LINE          LEVEL
 NUMBERS       PROC STMT     STATEMENT.          .          .          .
  ❾      ❿      ⓫   ⓬
  100      1     1    0     PROGRAM EXAMPLE(INPUT, OUTPUT) ;
  200      2     1    0
  300      3     1    0     LABEL 10 ;
  400      4     1    0
  500      5     1    0     VAR
  600      6     1    0        A, B, C, : REAL ;
  700      7     1    0
  800      8     1    0     BEGIN
  900      9     1    0
 1000     10     1    0        REPEAT
 1100     11     1    2           WRITELN('Enter triangle sides') ;
 1200     12     1    2           IF EOLN(INPUT) THEN
 1300     13     1    2              GOTO 10 ;
 1400     14     1    2           READLN(A, B) ;
 1500     15     1    2           C := ( SQR(A) + SQR(B) ) ** 0.5 ;
```

The lines of the source code are printed in the source code listing.

❾ SOS line numbers -- If you created or edited the source lines in a PASCAL module with the SOS editor, SOS line numbers appear in the leftmost column of the source code listing. SOS line numbers are irrelevant to the PASCAL compiler.

❿ Line numbers -- The compiler assigns unique line numbers to the source lines in a PASCAL module. The symbolic traceback that is printed if your program encounters an exception at run time refers to these line numbers. These line numbers are used when executing a program under the VAX-11 Symbolic Debugger.

⓫ Procedure level -- Each line that contains a declaration lists the procedure level of that declaration. Procedure level 1 indicates declarations in the outermost block. The procedure level number increases by one for each nesting level of functions or procedures.

⓬ Statement level -- The listing specifies a statement level for each line of source code after the first BEGIN delimiter. The statement level starts at 0 and increases by 1 for each nesting

level of PASCAL structured statements. The statement level of a comment is the same level as that of the statement that follows it.

Error and Warnings --

```
 2000     20                                  WRITELN( 'Done' ;
  %PAS-F-DIAGN                                        ^20,*,4    *** 20==>15
             *** ERROR   4:  ")" expected
             *** ERROR  20:  "," expected (15)   (13) (14) (16) (17)  (18)  (19)
 2100     21           1    1
 2200     22           1    1   END.
          23           1    0
```

The source code listing includes information on any errors or warnings detected by the compiler. A line beneath the source code line in which the error is detected specifies whether the diagnostic is a warning or an error.

(13) A circumflex (^) that points to the character position in the line where the error was detected.

(14) A numeric code, following the circumflex, that specifies the particular error.

(15) On the following lines of the source listing, the compiler prints the text that corresponds to each numeric error code.

(16) An asterisk (*) shows where the compiler resumed translation after the error.

(17) Note that one source program error often causes the PASCAL compiler to detect more than one error.

The following line numbers are indicated:

(18) The line number in which the error was detected

(19) The line number in the last previous line containing an error diagnostic

You can use these error numbers to trace the error diagnostics backwards through the source listing.

Summary --

```
 2 Errors 1 Nonstandard feature (20)
Last error (warning) on line (21)

Active options at end of compilation: (22)
NODEBUG,STANDARD,LIST,NOCHECK,WARNINGS,CROSS_REFERENCE,
MACHINE_CODE,OBJECT,ERROR_LIMIT = 30
```

If your program generated warning or error messages, the compiler prints a summary of:

(20) All the errors

(21) The source line number of the last message

(22) The compiler then lists the status of all the compilation options.

Compilation Statistics --

```
COMPILATION STATISTICS

  Total Space Allocated:  874 bytes, 363 Code + 511 Data (23)
  Stack Frame Size Total: 64 bytes (24)
  Run Time:               .50 seconds.          (1452 lines/minutes) (25)
  Page Faults:            990 (26)
```

The source code listing contains the following statistics for the current compilation.

(23) Total Space Allocated -- The number of bytes of static storage the program occupies. The total number of bytes is listed first followed by the amount of space required by the code, and data, respectively.

(24) Stack Frame Size Total -- This number is the total size of all the stack frames for all procedures and functions. This number is only an estimate of the maximum stack size since the number does not take into account recursion or conformant array parameters.

(25) Run Time -- This time reflects the CPU time used during the compilation.

(26) Page Faults -- This number reflects the number of page faults that occurred during compilation.

2.4.2 Cross-Reference Listing

```
 A (27)                                  6     14     15
 B                                       6     14     15
 C                                       6     15     16
 INPUT                                   1     12
 OUTPUT (29)                      (30)   1

GLOBALLY DEFINED IDENTIFIERS:

 EOLN                                    0     12
 FALSE (28)                              0     17
 READLN                                  0     14
 REAL                                    0      6
 SQR                                     0     15     15
 WRITELN                                 0     11     16     20
```

The cross-reference listing (if requested with the /CROSS_REFERENCE qualifier) appears before the machine code listing. It contains two sections.

(27) User-specified identifiers -- This section lists all the identifiers you declared.

(28) Globally defined identifiers -- This section lists the PASCAL predefined identifiers that the program uses.

Each line of the cross-reference listing contains:

(29) An identifier

(30) A list of the source line numbers where the identifier is used.

The first line number indicates where the identifier is declared. Predefined identifiers are listed as if they were declared on line 0. The cross-reference listing does not specify pointer type identifiers that are used before they are declared.

2.4.3 Machine-Code Listing

```
                                (31)
                        .TITLE  EXAMPLE
                        .IDENT  \01\
                                (32)
                        .EXTRN  PAS$CLOSEINOUT
                        .EXTRN  OTS$POWRR
                        .EXTRN  PAS$EOLN
                        .EXTRN  PAS$READLN
                        .EXTRN  PAS$READREAL
                        .EXTRN  PAS$WRITEREALE
                        .EXTRN  PAS$WRITELN
                        .EXTRN  PAS$WRITESTR
                        .EXTRN  PAS$OUTPUT
                        .EXTRN  PAS$INPUT

                ; PREDEFINED SYMBOLS
00000001        TRUE = 1
00000000        FALSE = 0
7FFFFFFF        MAXINT = 2147483647 (33)
00000000        NIL = 0
```

The machine-code listing (if requested with the MACHINE_CODE and LIST qualifiers) follows the cross-reference listing.

(31) The object module title and ident fields are listed first.

(32) All external routines called by the program are listed.

(33) Definitions of the PASCAL predeclared constants TRUE, FALSE, MAXINT, and NIL. The hexadecimal equivalent of each constant appears on the left side of the listing.

```
                ; SYMBOLS FOR EXAMPLE
000001D0        A = 464                                         ; 0006
000001D4        B = 468                                         ; 0006
000001D8        C = 472                                      (35) ; 0006
00000008        INPUT = 8    (34)                               ; 0001
000000EC        OUTPUT = 236                                    ; 0001
```

(34) Definitions of user-declared identifiers -- These identifiers are used in the listing of generated code as either register offsets or constant values. The hexadecimal equivalent of each name appears on the left side of the listing.

(35) The comment column, on the right side contains the source line number where each identifier was declared.

```
                                                                    00000
                                        (37)                         (36)
65 64 69 73 20 65 6C 67 6E 61 69 72 74 20 72 65 74 6E 45    00000
                                                          73    00013
            20 3A 73 69 20 65 73 75 6E 65 74 6F 70 79 48    00014
```

(36) The current location counter value appears in the center of the listing and divides the listing into 2 parts.

(37) The hexadecimal representation of the code appears on the left side of the counter and is read right to left.

```
00000           .PSECT  (38) $PDATA, PIC,REL,SHR,NOEXE RD,NOWRT
 (36)     (40)                (39)
00000  C.AAA:  .ASCII      \Enter triangle side\              ; 0011 (41)
00013          .ASCII      \s\
00014  C.AAB:  .ASCII      \Hypotenuse is: \                  ; 0016
```

(38) The symbolic representation of the code appears on the right side of the counter and is read left to right.

Each program section in the object module has a corresponding location counter.

(39) The $PDATA program section -- This section contains any constants longer than 4 bytes.

(40) The compiler generates names for literal (anonymous) constants of the form C.AAA, C.AAB, etc.

(41) The source line numbers where the constants were first used or declared appear as comments on the right.

```
                        .PSECT (42) $CODE, PIC,REL,SHR,EXE,RD NOWRT,2

4FAC  00000          (43) .ENTRY  EXAMPLE, -
      00002                    (44) ^M<R2,R3,R5,R7,R8,R9,R10, 11,IV>
```

(42) THE $CODE program section -- This program section contains the machine instructions generated by the compiler.

Each procedure and function starts with:

(43) The entry point

(44) The register save mask definition

```
                                                  (45)
      08   AE            14  D0 00049        MOVL    #20, 8(SP)
00000000'  EF            05  FB 0004D        CALLS   #5, PAS$WRITESTR
           5A   00EC     CB  9E 00054        MOVAB   OUTPUT(R11), R10
                         5A  DD 00059        PUSHL   R10
00000000'  EF            01  FB 0005B        CALLS   #1, PAS$WRITELN
           5A     08     AB  9E 00062        MOVAB   INPUT(R11), R10   : 0012
                         5A  DD 00066        PUSHL   R10
00000000'  EF            01  FB 00068        CALLS   #1, PAS$EOLN
           5A            50  D0 0006F        MOVL    R0, R10
           03            5A  E9 00072        BLBC    R10, 2$
                      00A8  31 00075        BRW     .10               ; 0013
                            00078 2$:
```

(45) The listing formats of the generated machine instructions are similar to the VAX-11 MACRO listings.

```
(46)      (47)
MOVL      #20, 8(SP)
CALLS     #5, PAS$WRITESTR
MOVAB     OUTPUT(R11), R10
PUSHL     R10                                   (48)
CALLS     #1, PAS$WRITELN
MOVAB     INPUT(R11), R10                      ; 0012
```

The right side of the listing contains:

(46) The symbolic opcode

(47) A set of symbolic operands

(48) A comment section that can follow the symbolic operand and contains a source line number if the corresponding instruction is the first one generated for that source line.

```
            (52)         (51)          (49)
             08    AE              14   D0 00049
      00000000'    EF              05   FB 0004D
                   5A    00EC      CB   9E 00054
                                   5A   DD 00059
 (53) 00000000'    EF              01   FB 0005B
                   5A      08      AB   9E 00062
                          (50)
```

The left side of the listing contains the following:

(49) The hexadecimal opcode

(50) The hexadecimal operands

Each operand consists of:

(51) A register/mode part

(52) An offset or literal value

(53) Operands whose values are supplied by the linker are flagged with a single quote.

```
  50  D0 00006F          MOVL    R0, R10
  5A  E9 00072           BLBC    R10, 2$ (55)
00A8  31 00075           BRW     .10
         00078 2$: (54)
```

(54) Compiler generated labels are designated by a $, and are the targets of the branch instructions.

(55) Branch instructions point to the compiler-generated labels.

```
  50   D0 0006F     MOVL     R0, R10
  5A   E9 00072     BLBC     R10, 2$
00A8   31 00075     BRW      .10 (56)
          00078 2$:
  .
  .
  .
  CB   9E 0010C     MOVAB    OUTPUT(R11), R8
  58   DD 00111     PUSHL    R8
  01   FB 00113     CALLS    #1, PAS$WRITELN
  00   E8 0011A     BLBS     #FALSE, .+3
FF10   31 0011D     BRW      1$
          00120 .10: (57)
```

(56) Branch instructions generated for GOTO statements use the user-defined labels as targets.

(57) User-defined labels are represented by a period followed by the label value.

(58)
Routine Size: 363 bytes, Routine Base: $CODE + 00000, (59)

Stack Frame Size (exclusive of conformant array): 64 bytes. (60)

A summary line is printed for each procedure and function. The line contains:

(58) The routine size in bytes

(59) The routine base in terms of an offset from the start of the $CODE program section.

(60) The stack frame size in bytes

The stack frame size includes any saved registers, but does not include copies of conformant arrays or any transient information such as parameter values pushed on the stack prior to a CALLS instruction.

(61)
```
.PSECT  $GLBL, PIC,OVL,REL,GBL,NO HR,NOEXE,RD,WRT
.BLKB   476
```

(61) The definition of the $GLBL program section -- This program section contains the storage for all program level variables. Conceptually, it is the stack frame for the main program, exclusive of saved registers. Register 11 always contains the base address of this program section.

2.4.4 A Compiler Listing Example

Figure 2-1 illustrates a complete compiler listing. The examples described in Sections 2.4.1, 2.4.2, and 2.4.3 are segments of this listing.

```
EXAMPLE                                    15-AUG-1980 13:47:41    VAX-11 PASCAL V1.2-80              Page 1
01                          SOURCE LISTING 19-JUN-1980 14:10:08    _DBA1:[SMITH]EXAMPLE.PAS;2 (1)

     LINE        LEVEL
     NUMBERS     PROC STMT  STATEMENT.        .         .         .         .         .         .         .         .

     100      1      1    0   PROGRAM EXAMPLE( INPUT, OUTPUT ) ;
     200      2      1    0
     300      3      1    0   LABEL 10 ;
     400      4      1    0
     500      5      1    0   VAR
     600      6      1    0      A, B, C : REAL ;
     700      7      1    0
     800      8      1    0   BEGIN
     900      9      1    0
    1000     10      1    0      REPEAT
    1100     11      1    2         WRITELN( 'Enter triangle sides' ) ;
    1200     12      1    2         IF EOLN( INPUT ) THEN
    1300     13      1    2            GOTO 10 ;
    1400     14      1    2         READLN( A, B ) ;
    1500     15      1    2         C := ( SQR(A) + SQR(B) ) ** 0.5 ;
  %PAS-W-DIAGN                                                ^450                  ***     15 ==>      0
                  *** WARNING 450:  Nonstandard Pascal: Exponentiation
    1600     16      1    2         WRITELN( 'Hypotenuse is: ', C ) ;
    1700     17      1    2      UNTIL FALSE ;
    1800     18      1    1
    1900     19      1    1   10:
    2000     20      1    1      WRITELN( 'Done' ;
  %PAS-F-DIAGN                                                    ^20,*,4           ***     20 ==>     15
                  *** ERROR    4:  ")" expected
                  *** ERROR   20:  "," expected
    2100     21      1    1
    2200     22      1    1   END.
             23      1    0

 2 Errors  1 Nonstandard feature
Last error (warning) on line     20.

Active options at end of compilation:
NODEBUG,STANDARD,LIST,NOCHECK,WARNINGS,CROSS_REFERENCE,
MACHINE_CODE,OBJECT,ERROR_LIMIT = 30

COMPILATION STATISTICS

  Total Space Allocated:  874 bytes, 363 Code + 511 Data
  Stack Frame Size Total: 64 bytes
  Run Time:               .50 seconds.        (   1452 lines/minute)
  Page Faults:            990
```

ZK-064-80

Figure 2-1 VAX-11 PASCAL Compiler Listing

```
EXAMPLE                                          15-AUG-1980 13:47:41    VAX-11 PASCAL V1.2-80              Page 2
01                          CROSS REFERENCE      19-JUN-1980 14:10:08    _DBA1:[SMITH]EXAMPLE.PAS;2 (1)
```

(27)

```
A                                6      14      15
B                                6      14      15
C                                6      15      16
INPUT                            1      12
OUTPUT (29)                   (30) 1

GLOBALLY DEFINED IDENTIFIERS:

EOLN (28)                        0      12
FALSE                            0      17
READLN                           0      14
REAL                             0       6
SQR                              0      15      15
WRITELN                          0      11      16      20
```

```
EXAMPLE                                          15-AUG-1980 13:47:41    VAX-11 PASCAL V1.2-80              Page 3
01                          GENERATED CODE       19-JUN-1980 14:10:08    _DBA1:[SMITH]EXAMPLE.PAS;2 (1)
```

(31)

```
                                .TITLE  EXAMPLE
                                .IDENT  \01\
```

(32)

```
                                .EXTRN  PAS$CLOSEINOUT
                                .EXTRN  OTS$POWRR
                                .EXTRN  PAS$EOLN
                                .EXTRN  PAS$READLN
                                .EXTRN  PAS$READREAL
                                .EXTRN  PAS$WRITEREALE
                                .EXTRN  PAS$WRITELN
                                .EXTRN  PAS$WRITESTR
                                .EXTRN  PAS$OUTPUT
                                .EXTRN  PAS$INPUT

                            ; PREDEFINED SYMBOLS
00000001                    TRUE = 1                (33)
00000000                    FALSE = 0
7FFFFFFF                    MAXINT = 2147483647
00000000                    NIL=0

                            ; SYMBOLS FOR EXAMPLE
000001D0                    A = 464                                                    ; 0006
000001D4                    B = 468                                                    ; 0006
000001D8                    C = 472          (34)                                   (35) ; 0006
00000008                    INPUT = 8                                                  ; 0001
000000EC                    OUTPUT = 236                                               ; 0001
```

Figure 2-1 (Cont.) VAX-11 PASCAL Compiler Listing

```
                                                          (36)                              (38)
                                                          00000         .PSECT  $PDATA, PIC,REL,SHR,NOEXE,RD,NOWRT
                    (37)                                             (40)  (39)
65 64 69 73 20 65 6C 67 6E 61 69 72 74 20 72 65 74 6E 45  00000 C.AAA:  .ASCII  \Enter triangle side\                ; 0011
                                                       73  00013         .ASCII  \s\                                       (41)
         20 3A 73 69 20 65 73 75 6E 65 74 6F 70 79 48  00014 C.AAB:  .ASCII  \Hypotenuse is: \                  ; 0016
                                                                          (42)
                                                          00000         .PSECT  $CODE, PIC,REL,SHR,EXE,RD,NOWRT,2

                                              4FAC        00000     (43) .ENTRY  EXAMPLE, -
                                                          00002         (44) ^M<R2,R3,R5,R7,R8,R9,R10,R11,IV>
                              5B 00000000'   EF    9E  00002         MOVAB   $GLBL, R11
                 00000004'    EF              5D    D0  00009         MOVL    FP, $GLBL+4
                                              7E    D4  00010         CLRL    -(SP)
                                              7E    7C  00012         CLRD    -(SP)
                                        08    AB    DF  00014         PUSHAL  INPUT(R11)
                 00000000'    EF              01    FB  00017         CALLS   1, PAS$INPUT
                                        08    AB    DF  0001E         PUSHAL  INPUT(R11)
                                      00EC    CB    DF  00021         PUSHAL  OUTPUT(R11)
                 00000000'    EF              02    FB  00025         CALLS   2, PAS$OUTPUT
                        F4    AD              5E    D0  0002C         MOVL    SP, -12(FP)
                                                        00030 1$:
                              5A      00EC    CB    9E  00030         MOVAB   OUTPUT(R11), R10                ; 0011
                              5E              10    C2  00035         SUBL2   16, SP
                                              5A    DD  00038         PUSHL   R10
                             59 00000000'    EF   9E  0003A        MOVAB   C.AAA, R9
                        04    AE              59    D0  00041         MOVL    R9, 4(SP)
                        0C    AE              14    D0  00045         MOVL    20, 12(SP)
```

```
EXAMPLE                                          15-AUG-1980 13:47:41     VAX-11 PASCAL V1.2-59              Page 4
01                  GENERATED CODE               19-JUN-1980 14:10:08     _DBA1:[SMITH]EXAMPLE.PAS;2 (1)
                        (52)        (51)          (49)          (45)      (46)      (47)
                        08    AE              14    D0  00049         MOVL     20, 8(SP)
                 00000000'    EF              05    FB  0004D         CALLS    5, PAS$WRITESTR
                              5A      00EC    CB    9E  00054         MOVAB   OUTPUT(R11), R10
                                              5A    DD  00059         PUSHL   R10
            (53) 00000000'    EF              01    FB  0005B         CALLS    1, PAS$WRITELN                 (48)
                              5A        08    AB    9E  00062         MOVAB   INPUT(R11), R10                 ; 0012
                                              5A    DD  00066         PUSHL   R10
                 00000000'    EF              01    FB  00068         CALLS    1, PAS$EOLN
                              5A              50    D0  0006F         MOVL    R0, R10
                              03              5A    E9  00072         BLBC    R10, 2$ (55)
                                            00A8    31  00075         BRW     .10 (56)                        ; 0013
                                   (50)                 00078 2$: (54)
                              5A        08    AB    9E  00078         MOVAB   INPUT(R11), R10                 ; 0014
                              59      01D0    CB    9E  0007C         MOVAB   A(R11), R9
                                              59    DD  00081         PUSHL   R9
                                              5A    DD  00083         PUSHL   R10
                 00000000'    EF              02    FB  00085         CALLS    2, PAS$READREAL
                              59      01D4    CB    9E  0008C         MOVAB   B(R11), R9
                                              59    DD  00091         PUSHL   R9
                                              5A    DD  00093         PUSHL   R10
```

Figure 2-1 (Cont.) VAX-11 PASCAL Compiler Listing

```
                00000000'  EF              02  FB 00095          CALLS    2, PAS$READREAL
                           5A        08    AB  9E 0009C          MOVAB    INPUT(R11), R10
                                           5A  DD 000A0          PUSHL    R10
                00000000'  EF              01  FB 000A2          CALLS    1, PAS$READLN
                           5A      01D0    CB  50 000A9          MOVF     A(R11), R10                    ; 0015
52                         5A              5A  45 000AE          MULF3    R10, R10, R2
                           53      01D4    CB  50 000B2          MOVF     B(R11), R3
58                         53              53  45 000B7          MULF3    R3, R3, R8
                           58              52  40 000BB          ADDF2    R2, R8
                           57              00  50 000BE          MOVF     ^F0.5, R7
                                           57  DD 000C1          PUSHL    R7
                                           58  DD 000C3          PUSHL    R8
                00000000'  EF              02  FB 000C5          CALLS    2, OTS$POWRR
                           52              50  50 000CC          MOVF     R0, R2
                     01D8  CB              52  50 000CF          MOVF     R2, C(R11)
                           58      00EC    CB  9E 000D4          MOVAB    OUTPUT(R11), R8                ; 0016
                           5E              10  C2 000D9          SUBL2    16, SP
                                           58  DD 000DC          PUSHL    R8
                           55 00000014'    EF  9E 000DE          MOVAB    C.AAB, R5
                       04  AE              55  D0 000E5          MOVL     R5, 4(SP)
                       0C  AE              0F  D0 000E9          MOVL     15, 12(SP)
                       08  AE              0F  D0 000ED          MOVL     15, 8(SP)
                00000000'  EF              05  FB 000F1          CALLS    5, PAS$WRITESTR
                           5E              10  C2 000F8          SUBL2    16, SP
                                           58  DD 000FB          PUSHL    R8
                       04  AE              52  D0 000FD          MOVL     R2, 4(SP)
                       08  AE              10  D0 00101          MOVL     16, 8(SP)
                00000000'  EF              05  FB 00105          CALLS    5, PAS$WRITEREALE
                           58      00EC    CB  9E 0010C          MOVAB    OUTPUT(R11), R8
                                           58  DD 00111          PUSHL    R8
                00000000'  EF              01  FB 00113          CALLS    1, PAS$WRITELN
                           03              00  E8 0011A          BLBS     FALSE, .+3                     ; 0017
                                         FF10  31 0011D          BRW      1$
                                               00120 .10: (57)
```

Figure 2-1 (Cont.) VAX-11 PASCAL Compiler Listing

```
EXAMPLE                                          15-AUG-1980 13:47:41    VAX-11 PASCAL V1.2-80              Page 5
01                      GENERATED CODE           19-JUN-1980 14:10:08    _DBA1:[SMITH]EXAMPLE.PAS;2 (1)

                          5A      00EC    CB  9E 00120         MOVAB   OUTPUT(R11), R10                    ; 0020
                          5E              10  C2 00125         SUBL2    16, SP
                                          5A  DD 00128         PUSHL   R10
                          59 656E6F44     8F  D0 0012A         MOVL     ^A\Done\, R9
                    04    AE              59  D0 00131         MOVL    R9, 4(SP)
                    0C    AE              04  D0 00135         MOVL     4, 12(SP)
                    08    AE              04  D0 00139         MOVL     4, 8(SP)
               00000000'  EF              05  FB 0013D         CALLS    5, PAS$WRITESTR
                          5A      00EC    CB  9E 00144         MOVAB   OUTPUT(R11), R10
                                          5A  DD 00149         PUSHL   R10
               00000000'  EF              01  FB 0014B         CALLS    1, PAS$WRITELN
                             08           AB  DF 00152         PUSHAL  INPUT(R11)                         ; 0022
               00000000'  EF              01  FB 00155         CALLS    1, PAS$CLOSEINOUT
                          00EC            CB  DF 0015C         PUSHAL  OUTPUT(R11)
               00000000'  EF              01  FB 00160         CALLS    1, PAS$CLOSEINOUT
                          50              01  D0 00167         MOVL     SS$_NORMAL, R0
                                              04 0016A         RET

 Routine Size:  363 bytes,    Routine Base:  $CODE + 00000,  Stack Frame Size (exclusive of conformant arrays):  64 bytes.

                                          ; R11-BASED PROGRAM LEVEL STATIC STORAGE
                                    00000       .PSECT  $GLBL, PIC,OVL,REL,GBL,NOSHR,NOEXE,RD,WRT
                                000001DC  00000 .BLKB   476

                                    001DC       .END
```

Figure 2-1 (Cont.) VAX-11 PASCAL Compiler Listing

CHAPTER 3

LINKING A PROGRAM

After a VAX-11 PASCAL program is compiled, you link the object module(s) to produce an executable image file. Linking resolves all references in the object code and establishes absolute addresses for symbolic locations.

3.1 THE LINK COMMAND FORMAT

To link an object module, the LINK command is issued in the following general form:

LINK/[command-qualifiers(s)] file-spec-list [/file-qualifier(s)]

/command-qualifier(s)

Specify output file options. Table 3-1 lists the command qualifier options in their positive and negative forms, and the default options.

file-spec-list

Specifies the input object file to be linked.

/file-qualifier(s)

Specify input file options. Table 3-2 lists the file qualifier options.

Table 3-1
Command Qualifiers

Command Qualifier	Negative Form	Default
/BRIEF	None	Not applicable
/CROSS_REFERENCE	/NOCROSS_REFERENCE	/NOCROSS_REFERENCE
/DEBUG	/NODEBUG	/NODEBUG
/EXECUTABLE [=file-spec]	/NOEXECUTABLE	/EXECUTABLE

(continued on next page)

Table 3-1(cont.)
Command Qualifiers

Command Qualifier	Negative Form	Default
/FULL	None	Not applicable
/MAP =file-spec	/NOMAP	/NOMAP (interactive) /MAP (batch)
/SHAREABLE [=file-spec]	/NOSHAREABLE	/NOSHAREABLE
/TRACEBACK	/NOTRACEBACK	/TRACEBACK

Table 3-2
File Qualifiers

File Qualifier	Negative Form	Default
/INCLUDE=module-name(s)	None	Not applicable
/LIBRARY	None	Not applicable

In interactive mode, you can issue the LINK command with no accompanying file specification. The system responds with the prompt:

```
$_File:
```

The file specification must be typed on the same line as the prompt. If the file specification does not fit on one line, type a hyphen (-) as the last character of the line and continue on the next line.

Multiple file specifications are entered separated by commas or plus signs. When used with the LINK command, the comma has the same effect as the plus sign: the linker creates a single executable image from the several input files. If no output file is specified, the linker produces an executable image with the same name as the first object module and a file type of EXE.

The following sections describe in detail the LINK command options.

3.2 COMMAND QUALIFIERS

The command qualifiers, in the LINK command, modify the output of the linker and specify the debugging or traceback facility. Linker output consists of an image file and, optionally, a map file. The following qualifiers, control the image file generated by the linker:

```
/EXECUTABLE [=file-spec]
/NOEXECUTABLE
/SHAREABLE [=file-spec]
```

These qualifiers are referred to as image-file qualifiers and are described in Section 3.2.1.

The following qualifiers control the map file generated by the linker:

```
/BRIEF
/CROSS_REFERENCE
/FULL
/MAP  [=file-spec]
```

These qualifiers are referred to as map-file qualifiers and are described in Section 3.2.2.

The following qualifiers specify the debugging or traceback facility.

```
/DEBUG
/TRACEBACK
```

These qualifiers are described in Section 3.2.3.

3.2.1 **Image-File Qualifiers**

The image-file qualifiers are:

```
/EXECUTABLE and /NOEXECUTABLE
/SHAREABLE
```

To produce an executable image, you specify the /EXECUTABLE qualifier. To suppress production of an image file, you specify the /NOEXECUTABLE qualifier. If no image-file qualifier is specified, the default is /EXECUTABLE.

For example:

```
$ LINK/NOEXECUTABLE Circle
```

The file Circle.OBJ is linked, but no image is generated. The /NOEXECUTABLE qualifier is useful to verify the results of linking an object file without actually producing the image.

To designate a file specification for an executable image, use /EXECUTABLE in the form:

```
/EXECUTABLE=file-spec
```

For example:

```
$ LINK/EXECUTABLE=Test Circle
```

The file Circle.OBJ is linked, and the executable image generated is named Test.EXE.

A shareable image is one that can be used in a number of different applications; as a private image for your own applications, or installed for use by all users in the system by the system manager. To create a shareable image, specify /SHAREABLE. For example:

```
$ LINK/SHAREABLE Circle
```

To include a shareable image as input to the linker, use an options file and specify the /OPTIONS file qualifier in the LINK command. Refer to the VAX-11 Linker Reference Manual for details.

If /NOSHAREABLE is specified, the linker generates an executable image.

3.2.2 Map-File Qualifiers

The map-file qualifiers control the generation of a map file and the contents of the map file.

The map-file qualifiers are:

/MAP [=file-spec] [/FULL] [/BRIEF] [/CROSS_REFERENCE]

Note that the /MAP option must be used if /BRIEF, /FULL, or /CROSS_REFERENCE is specified.

The linker uses defaults to generate or suppress a map file. In interactive mode, the default is to suppress the map; in batch mode, the default is to generate the map.

If no file specification is included with /MAP qualifier, the map file has the name of the first input file and a file type of MAP. The map file is stored on the default device, in the default directory.

The optional qualifiers /BRIEF and /FULL define the type of information included in the map file, as follows:

- /BRIEF produces a summary of the image's characteristics and a list of contributing modules.
- /FULL produces (1) a summary of the image's characteristics and a list of contributing modules (as produced by /BRIEF), (2) listings of global symbols by name and by value, and (3) a summary of characteristics of image sections in the linked image.

By default, if neither /BRIEF nor /FULL is specified, the map file contains a summary of the image's characteristics and a list of contributing modules (as produced by /BRIEF), plus a list of global symbols and values, in symbol name order. For a complete description of the map file's contents, refer to the VAX-11 Linker Reference Manual.

The /CROSS_REFERENCE qualifier can be used with either the default or /FULL map qualifier to request cross-reference information for global symbols. This cross-reference information indicates the object modules that define and/or refer to global symbols encountered during linking. The default is /NOCROSS_REFERENCE.

3.2.3 Debugging and Traceback Qualifiers

The /DEBUG qualifier indicates that the VAX-11 Symbolic Debugger is to be included in the executable image and a symbol table is to be generated. If /DEBUG is specified at link time, the program executes under the control of the debugger, unless /NODEBUG is specified with the RUN command. The default at link time is /NODEBUG.

The /TRACEBACK qualifier causes error messages accompanied by symbolic traceback information showing the sequence of calls that transferred control to the program unit in which the error occurred to be generated. /NOTRACEBACK specifies that no traceback information is to be produced. The default is /TRACEBACK. If you specify both /DEBUG and /NOTRACEBACK, the traceback capability is automatically included, and /NOTRACEBACK has no effect.

3.3 FILE QUALIFIERS

The file qualifiers /LIBRARY and /INCLUDE are used as modifiers on the input file specification. Input files can be object files, shareable images specified in an options file, or library files.

3.3.1 /LIBRARY Qualifier

The /LIBRARY qualifier has the form:

```
/LIBRARY
```

The /LIBRARY qualifier specifies that the input file is an object-module library that the linker must search to resolve undefined symbols referenced in other input modules. The default file type is OLB.

3.3.2 /INCLUDE Qualifier

The /INCLUDE qualifier has the form:

```
/INCLUDE=module-name(s)
```

The /INCLUDE qualifier specifies that the input file is an object-module library, and that the modules named are the only modules in that library that are to be explicitly included as input. At least one module name is required. To specify more than one, enclose the module names in parentheses, and separate them with commas. The /LIBRARY qualifier can be used with the /INCLUDE qualifier to modify a single input file specification. If /INCLUDE and /LIBRARY are specified for the same input file, the specified library is also searched for unresolved references.

CHAPTER 4

EXECUTING A PROGRAM

After you have compiled and linked your program, the system can execute it. The RUN command initiates execution. It has the form:

```
RUN /NODEBUG file-spec
```

You must specify the file name; default values are applied if you omit optional elements of the file specification. The default file type is EXE.

The DEBUG qualifier allows you to use the VAX-11 Symbolic Debugger, even if you omitted this qualifier from the PASCAL and LINK commands (see Sections 2.2 and 3.2). If you specify /NODEBUG, the program executes without debugger intervention. This qualifier allows you to override a /DEBUG qualifier specified at link time.

4.1 FINDING AND CORRECTING ERRORS

Both the compiler and the Run-Time Library include facilities for detecting and reporting errors. VAX/VMS also provides the debugger to help you locate and correct errors. In addition to the debugger, you can use a traceback facility to track down errors that occur during program execution.

4.1.1 Error-Related Command Qualifiers

At each step in compiling, linking, and executing your program, you can specify command qualifiers that affect how errors are reported. At compile time, you can use the /DEBUG qualifier to ensure that symbolic information is preserved for use by the debugger. At link time, you can also specify the /DEBUG qualifier to make the symbolic information available to the debugger. The same qualifier can be specified with the RUN command to invoke the debugger at run time.

Table 4-1 summarizes the /DEBUG and /TRACEBACK qualifiers.

If you use qualifiers at any point in the compile-link-execute sequence, and an execution error occurs, you receive a traceback list by default.

Table 4-1
/DEBUG and /TRACEBACK Qualifiers

Qualifier	Command	Effect	Default
/DEBUG	PASCAL	The PASCAL compiler creates symbolic data needed by the debugger.	/NODEBUG
/DEBUG	LINK	Symbolic data created by the compiler is passed to the debugger. Traceback list is also produced.	/NODEBUG
/TRACEBACK	LINK	Traceback information is passed to the debugger. Traceback list is produced.	/TRACEBACK
/DEBUG	RUN	Invokes the debugger. The DBG> prompt is displayed. Not needed if $ LINK/DEBUG was specified.	None
/NODEBUG	RUN	If /DEBUG was specified in the LINK command, RUN/NODEBUG suppresses the DBG> prompt.	None

4.1.2 Specifying Command Qualifiers

To perform symbolic debugging, you must specify the /DEBUG qualifier with both the PASCAL command and the LINK command. It then is unnecessary to specify /DEBUG with the RUN command. If you omit /DEBUG from either the PASCAL command or the LINK command, you can use it with the RUN command to invoke the debugger. However, the executable image does not contain debugger records or symbol tables used in debugging. You must express addresses as absolute values, rather than symbolically.

If you specify LINK/NOTRACEBACK combination, a traceback list is not produced in event of an error. Figure 4-1 shows an example of a source program listing and a traceback list.

The traceback list is interpreted as follows.

When the error is detected, you receive the appropriate message, followed by the traceback information. In this example, a message is displayed by the system, indicating the nature of the error, the address at which the error occurred (the Program Counter, PC), and the contents of the processor status longword (PSL). This message is followed by the traceback information.

The traceback information is presented in inverse order to the routine or subprogram calls. Of particular interest are the values listed under routine name and line, the first of which shows which routine or subprogram generated the error. The value given for line corresponds to a compiler-generated line number in the source program listing (not to be confused with editor-generated line numbers). The line number indicates the nearest previous line on which a statement begins. Using this information, you can usually isolate the error in a short time.

If you specify either LINK/DEBUG or RUN/DEBUG, the debugger assumes control of execution. If an error occurs, control reverts to the debugger; the traceback list is not automatically printed. For more information on using the debugger, refer to Chapter 5.

```
TRACETEST                              20-AUG-1980 13:32:07   VAX-11 PASCAL V1.2-59              Page   1
01                SOURCE LISTING       20-AUG-1980 13:31:50   _DBA1:[SMITH]TRACE.PAS;1 (1)

    LINE         LEVEL
   NUMBERS     PROC STMT  STATEMENT.       .       .       .       .       .       .       .       .

         1      1    0   PROGRAM TRACETEST;
         2      1    0
         3      2    0   PROCEDURE P1 (VAR X : REAL);
         4      2    0   BEGIN
         5      2    0       X := 1.0/X;
         6      2    1   END;
         7      2    0
         8      2    0   PROCEDURE P2 (Y : REAL);
         9      2    0   BEGIN
        10      2    0       P1(Y);
        11      2    1   END;
        12      2    0
        13      1    0   BEGIN
        14      1    0       P2(0.0);
        15      1    1   END.
        16      1    0

Active options at end of compilation:
NODEBUG,STANDARD,LIST,NOCHECK,WARNINGS,NOCROSS_REFERENCE,
NOMACHINE_CODE,OBJECT,ERROR_LIMIT = 30

COMPILATION STATISTICS

  Total Space Allocated:  140 bytes, 128 Code + 12 Data
  Stack Frame Size Total: 144 bytes
  Run Time:               0.13 seconds.          (   451 lines/minute)
  Page Faults:            983


  %SYSTEM-F-FLTDIV, arithmetic trap, floating/decimal divide by zero at PC=00000429, PSL=03C0002A
  %TRACE-F-TRACEBACK, symbolic stack dump follows

  module name     routine name      line    relative PC  absolute PC

  TRACETEST       P1                   5      00000029     00000429
  TRACETEST       P2                  10      00000030     0000045D
  TRACETEST       TRACETEST           14      0000004E     000004AC
```

ZK-065-80

Figure 4-1 Source Program Listing and Traceback List

4.2 SAMPLE TERMINAL SESSION

A simple dialog between you and the system might appear as follows:

```
(RET)
Username: SMITH (RET)
Password:  (RET)   (Your password is not displayed)

     WELCOME TO VAX/VMS VERSION 1.2

$ EDIT CIRCLE.PAS (RET)
Input:DBA2:[SMITH]CIRCLE.PAS
00100
     (enter source program)
*E  (RET)   (terminate edit session and write file to disk)
[DBA2:[SMITH]CIRCLE.PAS;1]
$ PASCAL/LIST Circle
$ LINK Circle
$ RUN Circle


$ ENTER VALUE:
```

CHAPTER 5

DEBUGGING PASCAL PROGRAMS

This chapter describes how to debug VAX-11 PASCAL programs interactively using the VAX-11 Symbolic Debugger.

Section 5.1 describes some features of the VAX-11 Symbolic Debugger.

Section 5.2 describes the debugging of a simple PASCAL program and gives a functional description of the debugger commands.

Section 5.3 describes in detail each debugger command in alphabetical order.

Section 5.4 describes the debugging of a complex PASCAL program.

For a detailed description of the debugger, refer to the VAX-11 Symbolic Debugger Reference Manual.

5.1 VAX-11 SYMBOLIC DEBUGGER

The VAX-11 Symbolic Debugger provides the following:

- Access to the symbol table generated by the PASCAL compiler
- A set of debugger commands and qualifiers that allow for specific control of an executing program

5.1.1 Debugger Symbol Table

The debugger maintains a table of symbols that can be referenced by a program during a debugging session. The symbol table provides information on all scalar (real, subrange, integer, character, Boolean, double, enumerated) and structured (array, record, file, set, pointer) variables, user-defined types, labels, procedures, functions, main programs, and modules.

To place the symbols from a program into the symbol table you must first specify the /DEBUG qualifier in the PASCAL and LINK commands. This makes the symbols available in the executable program image file. To copy a module's symbols from the image file to the symbol table you use a SET MODULE command. The debugger will place the symbols from the main program into the symbol table at the beginning of a debugging session. (Refer to Section 5.3.22 for more information on the SET MODULE command.)

5.1.2 VAX-11 Symbolic Debugger Command Syntax

You control the execution of a program by using VAX-11 Symbolic Debugger commands.

Section 5.3 describes the debugger commands in detail. In addition, Appendix C contains a list of all debugger commands, with abbreviations and syntax, for quick reference.

The debugger commands resemble other DIGITAL Command Language (DCL) commands and observe the same language conventions.

Format

COMMAND [[/qualifier(s)]] [[parameters]] [[DO(command-string;...)]] [[!comment]]

COMMAND

Specifies the command name.

/qualifier(s)

Modifies some debugging commands.

Qualifiers change the defaults the debugger uses to process commands. For example, when you deposit a value, the debugger uses decimal radix by default; you can override the default by specifying /HEXADECIMAL. Table 5-1 summarizes the command qualifiers of particular significance in PASCAL debugging.

Refer to the VAX-11 Symbolic Debugger Reference Manual for more information on qualifiers.

parameter

Specifies the object of the command. The parameter can be an address, constant, name of a variable, or an expression.

DO (command-string;...)

Tells the debugger to perform the specified command. (The command-string can be a series of debugger commands.)

!comment

Contains user remarks about the intent of the command.

Separate the command and parameter fields by one or more spaces.

Table 5-1
Debugger Command Qualifiers

Qualifier	Commands	Function
/ADDRESS	EVALUATE	Indicates that an address value is desired
/HEXADECIMAL /OCTAL	EVALUATE EXAMINE DEPOSIT	Overrides the default (decimal)
/BYTE /WORD /LONG /ASCII	EXAMINE EVALUATE DEPOSIT	Specifies a type

5.1.3 Debugger Operations

The EVALUATE command evaluates expressions and performs operations on integer, real, double-precision, character, Boolean, enumerated scalar, record, file, array, and set types. The PASCAL operators used in your programs that are supported by the debugger are listed below in decreasing order of precedence:

```
NOT
**
*,/, DIV, MOD, AND
+,-, OR
=,<>, <, <=, >, >=, IN
```

The debugger evaluates expressions in the same way as PASCAL. Spaces are used to separate elements of the debugger commands, and are significant to the debugger; therefore, variable names and multicharacter operators must contain no embedded spaces.

The debugger accepts constants in PASCAL syntax, with the following exception: constructors cannot be deposited into arrays or records nor can they be evaluated. For more information on constructors, refer to the VAX-11 PASCAL Language Reference Manual.

5.1.4 Specifying Addresses

The debugger allows you to specify addresses as symbolic names in the debugger commands. For example, to examine a variable you need only refer to it by its name; you need not know its actual memory location. This form of symbolic expression applies to data addresses (such as variables, array elements, or record fields), to program addresses (such as program line numbers, or labels) and to program unit names.

When you are debugging more than one program unit, you should be aware of the concept of scope (see Section 5.2.2.4), since it affects how the debugger interprets symbols. The following sections describe how to specify data and program addresses using the debugger commands.

5.1.4.1 **Specifying Data Addresses** - You can specify data addresses symbolically as variables names. For example:

```
DBG> DEPOSIT ISSUE = 100

DBG> EXAMINE PURCH[I,J+1]
```

The first command deposits the value 100 in the integer identifier ISSUE, and the second command examines the contents of an element of the array PURCH.

You can reference array elements with subscripts that are constants or expressions. If you reference a variable or array element that is not in the symbol table, or if you attempt to reference an element that is out of the array bounds, the debugger issues a warning.

5.1.4.2 **Specifying Current, Previous, and Next Locations** - The debugger provides a quick method for referencing any of three relative data addresses, or locations:

- A period (.) references the current location, that is, the location most recently referenced by an EXAMINE or DEPOSIT command.
- A circumflex (^) references the logical predecessor of the current location.
- A carriage return (RET) references the logical successor.

For example:

```
DBG> DEPOSIT .=100
```

This command puts a value of 100 in the current location. The current location is the last address used by an EXAMINE or DEPOSIT command.

To specify the previous location, type:

```
DBG> EXAMINE ^
```

This command displays the the contents of the previous location.

To specify the next location, type:

```
DBG> EXAMINE RET
```

This command displays the contents of the next location.

5.1.4.3 **Specifying Program Addresses** - You can specify program addresses by program unit name, routine name, line number, statement label, or (nonsymbolic) virtual address. To specify a procedure by name, give the command followed by the name of the procedure. For example:

```
DBG> SET BREAK proced1
```

This command sets a breakpoint at the entry to proced1.

To specify an address using a compiler-generated line number, use the %LINE prefix, as follows:

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

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

You can also set a breakpoint at a line number within a particular program unit. For example, to stop execution at line 11 in the module MOD2, you could set a breakpoint as follows:

```
DBG> SET BREAK MOD2\%LINE 11
```

To specify a GOTO statement label, use the %LABEL prefix. For example:

```
DBG> SET BREAK %LABEL 7
```

This command sets a breakpoint at statement label 7.

To specify a virtual address, issue the command without a modifier (for example, %LABEL). This causes the address you specify to be interpreted as the Program Counter (PC) value at which to perform the debugging function. For example:

```
DBG> SET BREAK %X700
```

This command sets a breakpoint at the hexadecimal address 700.

5.1.5 Special Symbols

The debugger accepts the special symbols listed in Table 5-2. These symbols are used in the debugger commands in place of an address or value.

Table 5-2
Special Symbols

Symbol	Definition	Example
%X	A hexadecimal number	DEPOSIT TOT = %XFF
%O	An octal number	EVALUATE %077
%B	A binary number	DEPOSIT TOT = %B101
\	The last value displayed by EXAMINE or EVALUATE	EXAMINE \
.	Current location	EVALUATE .
^	Logical predecessor	EXAMINE ^
RET	Logical successor	EXAMINE RET

(continued on next page)

Table 5-2 (Cont.)
Special Symbols

Symbol	Definition	Example
%R0-%R11	General registers 0-11	EXAMINE %R5
%AP	Argument Pointer	EXAMINE %AP
%FP	Frame Pointer	EXAMINE %FP
%SP	Stack Pointer	EXAMINE %SP
%PC	Program Counter	EXAMINE %PC
%PSL	Processor Status Longword	EXAMINE %PSL

5.2 USING THE DEBUGGER

The following sections illustrate how to debug a simple PASCAL program and describe in general terms the use of the debugger.

5.2.1 Debugging a PASCAL Program

Figure 5-1 is a program that requires debugging. The program was compiled and linked without diagnostic messages from either the compiler or the linker. However, the program produces erroneous results because of the missing asterisk in the exponentiation operator (Radius*2 should be Radius**2).

```
1     PROGRAM Circle(INPUT,OUTPUT);
2             CONST Pi = 3.1415927;
3             VAR Radius, Area : REAL;
4             LABEL1;
5             BEGIN
6               WRITE ('ENTER THE RADIUS VALUE: ');
7               WHILE NOT EOF
8               DO
9               BEGIN
10                WHILE NOT EOLN
11                DO
12                BEGIN
13                  READLN (Radius);
14                  1:  Area := Pi * Radius * 2;
15                  WRITELN ('AREA OF CIRCLE EQUALS ', Area : 4);
16                  WRITELN ('ENTER RADIUS VALUE OR CTRL/Z : ');
17                END;
18              END
19            END.
```

Figure 5-1 Sample PASCAL Program

The key to debugging is to find out what happens at critical points in your program. To do this, you need a way to stop execution at these points and look at the contents of program variables to determine whether they contain the correct values. Points at which execution is stopped are called breakpoints. The SET BREAK command lets you specify where you want to stop the program.

To look at the contents of a location, use the EXAMINE command. To resume execution, use either the GO or STEP command.

The debugger commands relevant to PASCAL are described in Section 5.3.

Figure 5-2 is an example of a simple terminal dialog for a debugging session. The circled numbers are keyed to notes that follow the figure and explain the dialog.

```
$ PASCAL/LIST/DEBUG CIRCLE (1)
$ LINK/DEBUG CIRCLE (2)
$ RUN CIRCLE (3)

         VAX-11 DEBUG VERSION 2.3

%DEBUG-I-INITIAL, language is PASCAL, module set to 'CIRCLE'

DBG> SET BREAK %LINE 14 (4)

DBG> GO (5)
routine start at CIRCLE
ENTER RADIUS VALUE : 24
break at CIRCLE\CIRCLE\%LINE 14 (6)

DBG> STEP (7)
start at CIRCLE\%LINE 14
stepped to CIRCLE\%LINE 15

DBG> EXAMINE Area (8)
CIRCLE\CIRCLE\AREA: 1.507964478E+02

DBG> EXAMINE Radius (9)
CIRCLE\CIRCLE\RADIUS: 2.400000000E+01

DBG> GO (10)
start at CIRCLE\%LINE 15
AREA OF CIRCLE EQUALS  1.4E+02 (11)
ENTER RADIUS VALUE OR CTRL/Z :

^Z
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion' (12)

DBG> EXIT (13)

$
```

Figure 5-2 Sample Debugging Session

❶ Invoke the PASCAL compiler, specifying the /LIST and /DEBUG qualifiers.

❷ Link the program using the /DEBUG qualifier to include a symbol table for the debugger.

❸ Issue the RUN command. In response, the debugger displays its identification, indicating that your program will be executed under the debugger's control. Following the identification message, the debugger displays an initial message, identifying the language and module settings it has assumed. The debugger derives these settings from the first main program specified in the LINK command.

❹ Set a breakpoint at an appropriate point in the program. This point should be one at which you are able to examine key variables. Note: Breakpoints suspend execution just before the point specified.

❺ Begin program execution. The debugger displays the point at which execution starts.

❻ The program request for input for the variable Radius is displayed. The debugger displays that program execution was suspended at the specified breakpoint.

❼ The STEP command causes the equation to be executed.

❽ Examine the variable Area. The debugger displays the value of Area.

❾ Examine the variable Radius.

❿ Resume execution. The debugger displays a message indicating the point at which program execution resumed.

⓫ The Area is calculated and displayed. A request for input is displayed, and a CTRL/Z is entered to end the program.

⓬ Successful completion of the program is indicated by this message. However, as you can see, the result is incorrect.

⓭ Exit from the debugger.

By examining the variables Radius and Area as the program is executing, you can determine that the values being entered or calculated are being stored correctly. It follows, then, that the error is probably in the expression of the formula for computing the area. To correct the problem, you must edit, recompile, and relink the source program, with the exponentiation operator properly specified in the formula expression.

5.2.2 General Description of the Debugger Commands

The debugger's commands can be broken down into four areas of function:

- The commands that allow you to control the environment (the language, where the output is to go) in which the debugger is to operate
- The commands that allow you to control the execution of your program
- The commands that allow you to examine and modify locations in your program
- The commands that allow you to specify scope

Each of these areas and the commands that perform these functions are discussed in the following sections.

5.2.2.1 **Preparing to Debug a Program** - The commands used to establish the proper environment for debugging PASCAL programs are:

```
SET LANGUAGE
SHOW LANGUAGE

SET LOG
SHOW LOG

SET MODE
SHOW MODE
CANCEL MODE

SET MODULE
SHOW MODULE
CANCEL MODULE

SET OUTPUT
SHOW OUTPUT

SET TYPE
SHOW TYPE
CANCEL TYPE/OVERRIDE
```

The LANGUAGE commands let you establish or determine the programming language.

- SET LANGUAGE indicates to the debugger that the debugging session should be conducted according to the conventions of the specified language.
- SHOW LANGUAGE displays the current language.

The LOG commands let you establish or determine the log file's name and whether it is being written to.

- SET LOG specifies the name of the log file.
- SHOW LOG displays the log file name and whether the log file is active.

The MODE commands let you establish, display, and cancel default entry and display modes.

- SET MODE establishes entry and display modes.
- SHOW MODE displays the current entry and display modes.
- CANCEL MODE cancels all modes and sets them to the default values for the current language.

The MODULE commands let you control the contents of the symbol table when the program you want to debug consists of multiple program units (a program and module(s)).

- SET MODULE places the symbols defined in the specified program unit or units in the debugger's internal symbol table.
- SHOW MODULE displays the names of all program units whose symbols are available to the debugger.
- CANCEL MODULE removes the specified program unit's symbols from the symbol table.

The OUTPUT commands control where the output from the debugger is written.

- SET OUTPUT specifies whether the debugger output is to be written to the terminal or to the log file.
- SHOW OUTPUT displays the output configuration.

The TYPE commands let you establish or determine the default data types for the DEPOSIT and EXAMINE commands.

- SET TYPE specifies the default data type as ASCII, BYTE, INSTRUCTION, LONG, or WORD.
- SHOW TYPE displays the current default data type.
- CANCEL TYPE/OVERRIDE cancels the current default data type.

5.2.2.2 **Controlling Program Execution** - To control the execution of your program, you must be able to suspend and resume execution at specific points. The following commands are available for these purposes:

```
SET BREAK
SET EXCEPTION BREAK
SHOW BREAK
CANCEL BREAK
CANCEL EXCEPTION BREAK

SET TRACE
SHOW TRACE
CANCEL TRACE

SET WATCH
SHOW WATCH
CANCEL WATCH
```

```
SHOW CALLS

GO

SET STEP
SHOW STEP
STEP

CTRL/Y

EXIT
```

The BREAK commands let you select specified locations for program suspension, so you can examine and/or modify structured or scalar variables in the program.

- SET BREAK defines a procedure's line number, statement label, or address at which to suspend execution.
- SET EXCEPTION BREAK defines exceptions (conditions that interrupt execution of your program) as breakpoints.
- SHOW BREAK displays all breakpoints currently set in the program.
- CANCEL BREAK removes selected breakpoints.
- CANCEL EXCEPTION BREAK removes the exception breakpoint.

The TRACE commands let you set, examine, and remove tracepoints in your program. 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 after a message is displayed. 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 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.

The WATCH commands let you monitor specified locations to determine when attempts are made to modify their contents, so you can take the appropriate action. These locations are called watchpoints. When an attempt is made to change the value of a watchpoint, the debugger suspends program execution, displays the old and new contents of the location, and prompts for a command. Watchpoints are monitored continuously. Thus, you can determine whether locations are being modified inadvertently during program execution.

The WATCH commands perform the following functions:

- 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.

The SHOW CALLS command is used to produce a traceback of procedure and function calls, and is particularly useful when you have returned control to the debugger following a CTRL/Y command.

The GO command lets you resume program execution.

- GO initiates execution at the current location and continues to the conclusion of the program or to the next breakpoint or watchpoint.

The STEP commands give you specific control over how much of a program is to be executed at one time. A program can, for example, be executed line by line or instruction by instruction.

The STEP commands perform the following functions:

- SET STEP establishes the current step conditions (LINE, INSTRUCTION, and so on).
- SHOW STEP displays the current step conditions.
- STEP initiates execution from the current location and continues for a specified number of lines.

The CTRL/Y command returns control to DCL command level but does not terminate the debugging session.

The EXIT command lets you exit from the debugger by terminating the debugging session.

5.2.2.3 **Examining and Modifying Locations** - Once you set breakpoints and begin program execution, the next step is to determine whether correct values are assigned to the identifiers and, possibly, to change the contents of locations as execution proceeds. You may also want to calculate the value of an expression that appears in your program. You can use the following commands:

DEPOSIT

EVALUATE

EXAMINE

The DEPOSIT command lets you place a new value into a variable during the debugging session.

The EVALUATE command lets you determine the value of an expression.

The EXAMINE command lets you display the current contents of a specific location.

Refer to Section 5.1.5 for information on special symbols that can be used with these commands.

5.2.2.4 **Specifying Scope** - If the program you are debugging consists of more than one program unit, your symbolic references should be unambiguous. Usually you can let the debugger specify scope, that is, the module in which a symbol is unique. Sometimes, however, you must tell the debugger how to resolve symbolic references. For example, suppose you are debugging two program units and both units use the variable I, and both variables are defined in the symbol table. Unless you explicitly specify scope, the debugger may not be able to determine which variable I you want to use. You can make a symbol unique by specifying scope in one of three ways:

- By using the debugger default scope
- By explicitly specifying the desired scope and the symbolic name in the command
- By explicitly specifying the desired scope in the SET SCOPE command (once scope has been set, it is in effect for all subsequent commands until you issue another SET SCOPE command)

When you begin a debugging session, the debugger automatically defines the main program unit as the default scope. However, this default scope is dynamic; that is, as you debug your program, the default scope is always the routine you are currently executing. The debugger's default scope rules are:

- If the specified symbol name is unique within the debugger symbol table, the debugger uses that name.
- If the specified symbol is ambiguous (not unique within the symbol table), but one of its declarations is within the current scope, the debugger uses that particular declaration.
- If the specified symbol is not defined in the symbol table, or if it is ambiguous with no declarations defined within the current scope, the debugger issues an error message.

You can specify scope explicitly by providing the name of the symbol and the name of the routine in which it is accessible, separated by a backslash (\) character, as shown in these examples:

```
DBG> SET WATCH MOD1\ARRMN[12]

DBG> SET BREAK MOD1\%LINE10
```

In addition, if you want to make a number of symbolic references within the same program unit, you can eliminate the need to specify scope with each symbolic address by using the SET SCOPE command.

You specify the scope of a name explicitly by providing both the symbol name 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, because it consists of a path of names for several nested routines. For example:

```
PROGRAM Main(INPUT, OUTPUT);
     .
     .
     .
PROCEDURE Check_Table (X,Y : INTEGER);
          VAR Z : CHAR;
          .
          .
          .
     PROCEDURE Put_Table (R : REAL );
               VAR Z : CHAR;
               .
               .
               .
```

To change the value of Z, which is the variable defined in the procedure Put_Table, the following debugger command is given:

```
DBG> DEPOSIT MAIN \Check_Table \Put_Table \Z = 'a'
```

By specifying this pathname you change only the value of Z in the procedure Put_Table. The variable Z in the procedure Check_Table remains unchanged.

Another way of altering the variable Z in the routine Put_Table follows:

```
DBG> SET SCOPE Put_Table
DBG> DEPOSIT Z = 'a'
```

This method does not use a pathname, but defines the scope as Put_Table and then deposits a value directly into the variable Z.

You can also use a pathname to specify an address reference that is in a routine not under the current scope setting. This is done by giving the address a pathname qualifier:

```
DBG> SET BREAK MOD1\%LINE 7
```

This command sets a breakpoint at line 7 of the module MOD1.

The following commands can be used to specify scope:

```
SET SCOPE
SHOW SCOPE
CANCEL SCOPE
```

These commands let you explicitly control how the debugger resolves symbolic references.

- SET SCOPE explicitly establishes the specified unit or units as the scope to be used for the translation of symbols.
- SHOW SCOPE displays the current setting of scope.
- CANCEL SCOPE revokes the program unit previously established in a SET SCOPE command and establishes as the default scope the currently active program unit.

5.3 VAX-11 SYMBOLIC DEBUGGER COMMANDS

The following sections describe the VAX-11 Symbolic Debugger commands that can be used to debug a PASCAL program. The commands are listed in alphabetical order for ease of reference.

CANCEL ALL

5.3.1 CANCEL ALL Command

The CANCEL ALL command cancels all breakpoints, tracepoints, watchpoints, and restores scope and user-set entry/display modes to their default values.

CANCEL ALL does not affect the modules included in the debugger symbol table or the current language.

Format

CANCEL ALL

Command Parameters

None.

Command Qualifiers

None.

Examples

DBG> CANCEL ALL

All breakpoints, tracepoints, watchpoints, are canceled. Scope and modes are set to their default values.

CANCEL BREAK

5.3.2 CANCEL BREAK Command

The CANCEL BREAK command cancels a specific breakpoint or all breakpoints.

Format

CANCEL BREAK [/qualifier] [address-expression]

Command Qualifiers

/ALL

Command Parameters

address-expression

Specifies the address at which a breakpoint is to be canceled.

Command Qualifiers

/ALL

Cancels all breakpoints in the program.

Examples

```
DBG> CANCEL BREAK %LINE 110
```

The break located at line 110 is canceled.

CANCEL EXCEPTION BREAK

5.3.3 CANCEL EXCEPTION BREAK Command

The CANCEL EXCEPTION BREAK command cancels the exception breakpoint.

Format

CANCEL EXCEPTION BREAK

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG> CANCEL EXCEPTION BREAK
All exception breakpoints are canceled.
```

CANCEL MODE

5.3.4 CANCEL MODE Command

The CANCEL MODE command cancels all modes and types and sets them to the default values.

The CANCEL MODE command sets the default address display mode to the PASCAL default of symbolic and the radix mode to the default radix of decimal.

Format

CANCEL MODE

Command Parameters

None.

Command Qualifiers

None.

Examples

DBG> CANCEL MODE

The current modes are canceled and are reset to the PASCAL default values.

CANCEL MODULE

5.3.5 CANCEL MODULE Command

The CANCEL MODULE command removes the specified program unit's symbols from the debugger's internal symbol table.

Format

CANCEL MODULE [[/qualifier]] [[module [[,module ...]]]]

Command Qualifiers

/ALL

Command Parameters

module

Specifies the name of the program unit for which symbols are to be removed from the active symbol table.

Command Qualifiers

/ALL

Specifies that all information is to be deleted from the active symbol table.

Examples

1. DBG> CANCEL MODULE MOD1

   The symbols from the program unit MOD1 are removed from the active symbol table.

2. DBG> CANCEL MODULE/ALL

   All the symbols in the active symbol table are removed.

CANCEL SCOPE

5.3.6 CANCEL SCOPE Command

The CANCEL SCOPE command cancels all previous settings for scope and resets the scope to the current scope, the scope of the program unit within which the PC is currently located.

The current scope changes as different sections of the program are executed. CANCEL SCOPE is the same as setting the scope equal to 0 (refer to Section 5.3.24).

Format

CANCEL SCOPE

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG> CANCEL SCOPE

The scope(s) currently set are canceled.
```

CANCEL TRACE

5.3.7 CANCEL TRACE Command

The CANCEL TRACE command specifies an address at which the tracepoint is to be removed.

Format

CANCEL TRACE [[/qualifier]] [[address]]

Command Qualifiers

/ALL

Command Parameters

address

Specifies the address from which a tracepoint is to be removed.

Command Qualifiers

/ALL

Removes all tracepoints in the program.

Examples

1. DBG> CANCEL TRACE %LABEL 140

   The tracepoint at label 140 is canceled.

2. DBG> CANCEL TRACE/ALL

   All existing tracepoints are canceled.

CANCEL TYPE/OVERRIDE

5.3.8 CANCEL TYPE/OVERRIDE Command

The CANCEL TYPE/OVERRIDE command cancels the current override type and resets the override type to none.

Format

CANCEL TYPE/OVERRIDE

Command Parameters

None.

Command Qualifiers

/OVERRIDE

The /OVERRIDE qualifier must be specified.

Examples

```
DBG> CANCEL TYPE/OVERRIDE
```

The current override type is canceled and the override type is reset to none.

CANCEL WATCH

5.3.9 CANCEL WATCH Command

The CANCEL WATCH command cancels a watchpoint during a debugging session.

Format

CANCEL WATCH [[/qualifier]] [[var]]

Command Qualifiers

/ALL

Command Parameters

var

Specifies the location at which monitoring is to be disabled.

Command Qualifiers

/ALL

Removes all watchpoints from the program.

Examples

1. DBG> CANCEL WATCH AREA

   This example cancels the monitoring of location AREA.

2. DBG> CANCEL WATCH/ALL

   All existing watchpoints are canceled.

CTRL/Y

5.3.10 CTRL/Y Command

You can use the CTRL/Y command at any time to return to the system command level. To issue this command, press the Y key while holding down the CTRL key. The dollar sign ($) prompt is displayed on the terminal. To return control to the debugger, type DEBUG. This brings you back to the debugger in your old program with the same set of command defaults. You can use the CTRL/Y command if your program loops or otherwise fails to stop at a breakpoint. To determine the point at which CTRL/Y was executed, use the SHOW CALLS command (Section 5.3.30) after you return to the debugger.

Format

CTRL/Y

Command Parameters

None.

Command Qualfiers

None.

Examples

```
DBG> GO CTRL/Y
^Y
$DEBUG

     VAX-11 DEBUG VERSION 2.3

DBG>
```

DEPOSIT

5.3.11 DEPOSIT Command

The DEPOSIT command places the contents of the value field into the specified variable:

- If the value is the same size as the specified variable, the value is deposited in the variable with no conversion.
- If the value is larger than the specified variable, the value is truncated on the left and deposited in the variable.
- If the value is smaller than the specified variable, the value is zero-filled on the left and deposited in the variable.

If you assign a value to a variable that does not conform to the rules of assignment compatibility, an informational message is displayed. (For more information on assignment compatibility, refer to the VAX-11 PASCAL Language Reference Manual.)

Format

DEPOSIT [[/qualifier]] address-expression = value

Command Qualifiers

/ASCII:length
/BYTE
/DECIMAL
/HEXADECIMAL
/INSTRUCTION
/LONG
/OCTAL
/WORD

Command Parameters

address-expression

Specifies the variable reference into which the value is to be deposited.

value

Specifies the value to be deposited.

Command Qualifiers

/ASCII:length

Specifies that the data value is to be interpreted as an ASCII string. The length specifies the number of bytes of ASCII data to be deposited for each data item. It is interpreted in the radix specified by a command qualifier that precedes the /ASCII qualifier or in the current radix mode. If length is omitted, the debugger assumes a length of 4.

/BYTE

Specifies that the data value is to be interpreted as a byte integer. For each data item specified, the debugger deposits 1 byte of data. The data is interpreted in the radix mode specified by a command qualifier or in the current radix mode. Any value that cannot be stored in a byte (greater than 255, unsigned) is left-truncated (that is, the high-order bits are dropped).

/DECIMAL

Specifies that all numbers following the qualifier in the command are to be interpreted in decimal radix.

/HEXADECIMAL

Specifies that all numbers following the qualifier in the command are to be interpreted in hexadecimal radix.

/INSTRUCTION

Specifies that the data is instructions enclosed in quotation marks or apostrophes. The debugger deposits the binary opcode and operands. The length of the instruction deposited depends on the opcode and the addressing modes used.

/LONG

Specifies that the data value is to be interpreted as a longword integer. For each data item specified, the debugger deposits 4 bytes of data. The data is interpreted in the radix mode specified by a command qualifier or in the current radix mode. Any value that cannot be stored in a longword will be left-truncated.

/OCTAL

Specifies that all numbers following the qualifier in the command are interpreted in octal radix.

/WORD

Specifies that the data value is to be interpreted as a word integer. For each data item specified, the debugger deposits 2 bytes of data. The data is interpreted in the radix mode specified by a command qualifier or in the current radix mode. Any value that cannot be stored in a word (greater than 65,535, unsigned) is left-truncated.

Examples

1. DBG> DEPOSIT Hair_Color = Brown

   DBG> DEPOSIT Married = True

   These examples place the values Brown and True into the variables Hair_Color and Married.

2. DBG> DEPOSIT A^.Name = 'SMITH'

   The name 'SMITH' is deposited into the name field of the record A.

EVALUATE

5.3.12 EVALUATE Command

The EVALUATE command functions as a calculator to determine the value of an expression.

Format

EVALUATE [[/qualifier]] expression

Command Qualifiers

/ADDRESS
/DECIMAL
/HEXADECIMAL
/OCTAL

Command Parameters

expression

Specifies the expression whose value is to be determined. A backslash (\) can also be used in an expression to reference the last expression evaluated. A function reference is the only PASCAL expression that cannot be evaluated. An expression must contain only variables or constants in the active symbol table to be evaluated.

Command Qualifiers

/ADDRESS

Indicates that an address value is desired.

/DECIMAL

Specifies that decimal is the default radix for numbers entered in the command and for the display of values.

/HEXADECIMAL

Specifies that hexadecimal is the default radix for numbers entered in the command and for the display of values.

/OCTAL

Specifies that octal is the default radix for numbers entered in the command and for the display of values.

Examples

1. DBG> EVALUATE Pi*Radius
   6.28

   The value of the expression Pi*Radius is displayed.

2. DBG> EVALUATE/ADDRESS /HEX I
   007F3AA4

   This command calculates and displays the hexadecimal address of the variable I.

3. DBG> EVALUATE/ADDRESS A[J]
   1600

   This command calculates the decimal address of the Jth element of array A and displays it.

4. You can also use EVALUATE to perform address arithmetic, such as computing an offset or array element address, as in this following example.

   DBG> EVALUATE/ADDRESS I+4
   3000

   This command calculates the decimal address of the data stored at a location 4 bytes after the address of I.

5. DBG> EVALUATE \
   3000

   The last scalar value resulting from an EVALUATE or DEPOSIT command is displayed.

EXAMINE

5.3.13 EXAMINE Command

The EXAMINE command displays the contents of specified locations.

Note you cannot examine variables of an array, record, or file type, except when the variable is a packed or unpacked array [1..n] of CHAR, where n is less than 65535.

The following predefined locations can be examined:

- %R0 - %R11 General registers 0-11
- \ Last value
- . Current location
- ^ Logical predecessor
- (RET) Logical successor
- %AP Argument Pointer
- %FP Frame Pointer
- %SP Stack Pointer
- %PC Program Counter
- %PSL Processor Status Longword

Format

EXAMINE [/qualifier] [address-expression]

Command Qualifiers

/ASCII:length
/BYTE
/DECIMAL
/HEXADECIMAL
/INSTRUCTION
/LONG
/NOSYMBOLIC
/OCTAL
/SYMBOLIC
/WORD

Command Parameters

address-expression

Specifies the address whose contents are to be examined. The address is usually given symbolically as a scalar variable name or component of a structured variable type. You can examine array elements or record fields but not whole arrays or records; except for variables of the type PACKED ARRAY [1..n] OF CHAR, which are displayed as ASCII strings.

Command Qualifiers

/ASCII:length

Specifies that the data value is to be interpreted as an ASCII string. The length specifies the number of bytes of memory to be examined and the number of characters to be displayed. It is interpreted in the radix specified by a command qualifier that precedes the /ASCII qualifier or in the current radix. If length is omitted, the debugger assumes a length of 4. The number of characters actually displayed is limited by the maximum size of the debugger's output line, 132 characters.

/BYTE

Specifies that the data value is to be interpreted as a byte integer. The contents of each byte of the value examined is displayed in the radix mode specified by a command qualifier or in the current radix mode.

/DECIMAL

Specifies that the radix is decimal. The debugger displays all numbers and interprets all numbers in the command in decimal radix.

/HEXADECIMAL

Specifies that the radix is hexadecimal. The debugger displays all numbers and interprets all numbers in the command in hexadecimal radix.

/INSTRUCTION

Specifies that the contents of memory should be displayed as VAX-11 MACRO instructions. The debugger attempts to decode the address specified as an instruction. If the debugger can decode the instruction, it displays the instruction. If the debugger cannot decode the instruction, it displays a warning message. The length of the instruction displayed varies depending on the opcode and addressing modes.

/LONG

Specifies that the data value is to be interpreted as a longword integer. The contents of each longword (4 bytes) of the value examined is displayed in the radix mode specified by a command qualifier or in the current radix mode.

/NOSYMBOLIC

Specifies that addresses should be displayed as absolute virtual addresses. The debugger displays the addresses being examined and any addresses in decoded instructions as absolute virtual addresses.

/OCTAL

Specifies that the radix is octal. The debugger displays all numbers and interprets all numbers in the command in octal radix.

/SYMBOLIC

Specifies that addresses should be displayed as symbols or offsets from symbols. The debugger displays the addresses being examined and any addresses in decoded instructions as symbols or offsets from symbols.

/WORD

Specifies that the data value is to be interpreted as a word integer. The contents of each word (2 bytes) of the value examined is displayed in the radix mode specified in a command qualifier or in the current radix mode.

Examples

1.
```
DBG> EXAMINE Flight_Number
FLIGHT\FLIGHT FLIGHT_NUMBER:10
```

The contents of variable Flight_Number are displayed.

2.
```
DBG> EXAMINE Entered_Flight
FLIGHT\FLIGHT Entered_Flight:0
```

The contents of Entered_Flight are displayed.

3.
```
DBG> EXAMINE Flights[1].Reservations^.Name
FLIGHT\FLIGHT FLIGHTS[1].Reservation ^.Name : Jones
```

The contents of the field Reservation^.Name in the record specified by the first element of the array Flights are examined.

4.
```
DBG> EXAMINE 600
600: 1
```

The contents of the nonsymbolic virtual address 600 (decimal) are examined.

5.
```
DBG> EXAMINE
ARRVAR[2]:2.5
```

The contents of the logical successor (the next array element here) of the current location are displayed.

EXIT

5.3.14 EXIT Command

The EXIT command lets you exit from the debugger when you are ready to terminate a debugging session. You must use this command to return to DCL command level.

Format

EXIT

Command Parameters

None.

Command Qualifiers

None.

Examples

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

The debugging session is terminated by the EXIT command at the end of normal program execution.

5.3.15 GO Command

The GO command starts the execution of a program for a debugging session.

Format

GO [[address-expression]]

Command Parameters

address-expression

Specifies the address at which execution is to begin; if no address is specified, execution begins at the current location.

Command Qualifiers

None.

NOTE

You must not restart a program from the beginning unless you first exit from the debugger. Otherwise, unpredictable results may occur.

Examples

```
DBG> GO

routine start at MAIN\MAIN
break at MOD1\MOD1\%LINE 3
```

The debugger responds to the GO command by indicating that execution started at the routine MAIN\MAIN and that a breakpoint has been reached in module MOD1\MOD1 at line 3.

HELP

5.3.16 HELP Command

The HELP command displays a description of the command specified. You can also get descriptions of command parameters and qualifiers.

The HELP command displays a description of debugger commands, their format, and the parameters and qualifiers you can use with them. You can find out the topics that have help descriptions by entering the HELP command with no topics.

Format

HELP topic [subtopic ...]

Command Parameters

topic

Specifies the command that you want information about.

subtopic

Specifies the command keyword, qualifier, or parameter that you want information about. If you want information about a specific qualifier, specify the qualifier (including the initial slash) as the subtopic. If you want information about all qualifiers or all parameters, specify QUALIFIER or PARAMETER, respectively. If you want all the information about a command, specify an asterisk (*) as the subtopic.

Command Qualifiers

None.

Examples

```
DBG> HELP GO
GO

Starts or continues program execution.

If the GO command is specified without an address-expression as a
parameter, execution resumes at the point of suspension or, in
the case of debugger start-up, at the transfer address.

If the GO command is specified with an address-expression as a
parameter, execution resumes at the location denoted by the
address-expression. Note that using an address-expression as a
parameter in the GO command can produce unpredictable results if
the state of your program required to commence execution at the
location specified by the address-expression is not identical to
the state of your program at the time the GO command is issued.

Format:

GO [[address-expression]]

Additional information available:

Parameters
```

SET BREAK

5.3.17 SET BREAK Command

The SET BREAK command specifies a label, line number, or address at which program execution is to be suspended. This location is called a breakpoint. The DO command-string specifies one or more of the debugger commands that are to be performed when the breakpoint is reached.

Note that you cannot set breakpoints, tracepoints, or watchpoints at the same location. The most recently issued command overrides any other.

Format

```
SET BREAK [/qualifier] address-expression [DO(command-string [;command-string] ...)]
```

Command Qualifiers

/AFTER:count

Command Parameters

address-expression

Specifies the address at which the program execution is suspended. Note that execution is suspended just before the specified address. Section 5.1.4 describes how addresses are specified.

DO(command-string)

Tells the debugger to perform the specified debugger command(s) specified by the command-string when the breakpoint is reached.

Command Qualifiers

/AFTER:count

Specifies that the debugger should not stop execution until the breakpoint is reached the number of times specified by count.

Examples

1. DBG> SET BREAK %LINE 100 DO(EXAMINE Total; EXAMINE Area)

   In this example, the values of the variables Total and Area are displayed when the breakpoint at line 100 is reached. The line number is located on the source and machine-code listings.

2. DBG> SET BREAK/AFTER:3 %LINE 20

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 body of a WHILE loop, and you want the breakpoint to occur the third time through the loop, specify the /AFTER qualifier with a value of 3, as shown in this example.

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.

SET EXCEPTION BREAK

5.3.18 SET EXCEPTION BREAK Command

The SET EXCEPTION BREAK command specifies exceptions (conditions that interrupt execution of your program) as breakpoints. For more information on exceptions, refer to the VAX/VMS System Services Reference Manual and the VAX-11 Run-Time Library Reference Manual.

Format

SET EXCEPTION BREAK

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG> SET EXCEPTION BREAK
```

A breakpoint will occur when an exception occurs during program execution.

SET LANGUAGE

5.3.19 SET LANGUAGE Command

The SET LANGUAGE command tells the debugger to conduct the dialog according to the conventions of the specified language. The default language is the language of the main program.

Format

SET LANGUAGE language-name

Command Parameters

language-name

Specifies the language to be used.

Command Qualifiers

None.

Examples

```
DBG> SET LANGUAGE PASCAL

The current language is set to PASCAL.
```

SET LOG

5.3.20 SET LOG Command

The SET LOG command specifies the name of the log file that the debugger uses when the output is directed to the log file. A log file contains a copy of all the user's commands and the debugger responses during a debugging session.

The SET LOG command controls only the name of the log file; it does not control whether a log file is being written. The SET OUTPUT command (5.3.23) does this. By default, the debugger uses the file name DEBUG and the file type LOG. If the debugger is currently creating a log file and you enter SET LOG, the debugger closes the existing log file. A new file is opened using the newly specified file name and all subsequent debugger commands are recorded in the new log file.

Format

SET LOG file-spec

Command Parameters

file-spec

Specifies the name of the log file. You must enclose the file specification in quotation marks or apostrophes only if the file specification begins with a special symbol (such as a square bracket).

Command Qualifiers

None.

Examples

```
DBG> SET LOG CALC.DBG

DBG> SET LOG "[CODEPROJ]FEB29.TMP"
```

Log files are created with the names CALC.DBG and "[CODEPROJ]FEB29.TMP."

SET MODE

5.3.21 SET MODE Command

The SET MODE command sets the default entry and display modes. The entry and display modes control the current radix and whether addresses and instruction operands are displayed symbolically or as absolute virtual addresses. The default radix controls how the debugger interprets numbers that you enter and how it displays numbers.

You can override the current radix mode by using a radix mode qualifier in a DEPOSIT, EVALUATE, or EXAMINE command or by specifying a radix control operator (%X, %O, %B).

In SYMBOLIC mode, addresses are displayed as symbols, virtual addresses or offsets from symbolic locations. The debugger first looks for an exact match with a local symbol or a global symbol. If no exact match can be found, the debugger searches for the symbol whose value is less than the address and closest to it. If it cannot find any symbolic definition, it displays the address as a virtual address in the current radix mode. Probable causes of the debugger displaying a virtual address are that the module's symbols are not included in the the debugger symbol table or that the address is outside the program's address space.

In NOSYMBOLIC mode, addresses are displayed as virtual addresses in the current radix mode. Note that you can always enter symbolic addresses even when the mode is NOSYMBOLIC.

Format

SET MODE mode-keyword [,mode-keyword]

Command Parameters

mode-keyword

Specifies the entry and display mode. Mode-keyword can be:

- DECIMAL -- Sets the current radix to decimal
- HEXADECIMAL -- Sets the current radix to hexadecimal
- OCTAL -- Sets the current radix to octal
- NOSYMBOLIC -- Specifies that addresses should be displayed as absolute virtual addresses
- SYMBOLIC -- Specifies that addresses should be displayed as offsets from symbolic locations

Command Qualifiers

None.

Examples

```
DBG> SET MODE DEC,NOSYM
```

The modes decimal and nosymbolic are set as the current entry and display modes.

SET MODULE

5.3.22 SET MODULE Command

The SET MODULE command places the symbols defined in the specified program unit or units in the symbol table. By default, the debugger initializes the symbol table to include all global symbols and local symbols of the main program unit which contains the entry point or the point at which execution begins.

Format

SET MODULE [[/qualifier]] [[module-name [[,module-name...]]]]

Command Qualifiers

/ALL

Command Parameters

module-name

Specifies the name of the module whose symbols are to be included in the symbol table. A PASCAL module-name is either a program or a module.

Command Qualifiers

/ALL

Tells the debugger to use the symbols of all known (linked) modules. If there is insufficient space, the debugger displays an error message.

Examples

1. DBG> SET MODULE MAIN, MOD1

   The symbols from the program units MAIN and MOD1 are added to the active symbol table.

2. DBG> SET MODULE/ALL

   All the symbols from the programs and modules are included in the active symbol table.

SET OUTPUT

5.3.23 SET OUTPUT Command

The SET OUTPUT command controls whether the debugger displays output on the terminal or writes it to a log file and controls whether the debugger echoes commands in command procedures.

When the debugger is initiated, all responses are displayed on the terminal, no log file is created, and command procedures and DO command-string sequences are not echoed on the terminal.

The log file contains all the commands that you enter at a terminal and all the responses of the debugger. The log file does not contain the DBG> prompt. The debugger's responses are preceded by an exclamation mark. To reproduce a debugging session, you can use the log file as a command procedure.

Format

SET OUTPUT option [,option ...]

Command Parameters

option

Specifies the mode of output. Option can be LOG, TERMINAL, or VERIFY, or the negative form of each.

- LOG -- starts writing output to the log file. The log file contains all the commands that you enter at the terminal and all debugger responses. The debugger responses are preceded by an exclamation mark (comment indicator) to allow you to use the log file as a command procedure. The default setting, NOLOG, inhibits output to the log file.

- TERMINAL -- starts writing output to the terminal. The default setting, TERMINAL, causes all the debugger responses to be displayed on the terminal. The NOTERMINAL parameter causes the debugger to stop displaying all responses except error messages on the terminal. The NOTERMINAL parameter is useful when you want to write output only to a log file. If you specify SET OUTPUT NOLOG, NOTERMINAL, the debugger displays a warning message telling you that output is being lost.

- VERIFY -- causes the debugger to echo commands executed from command procedures and DO command-string sequence of SET BREAK commands. The commands are displayed on the terminal and written to the log file depending on the other options specified for the SET OUTPUT command. The default setting, NOVERIFY, causes the debugger to not echo commands as they are executed in a command procedure or DO command-string sequence.

Command Qualifiers

None.

Examples

1. DBG> SET OUTPUT NOLOG ,TERMINAL ,VERIFY

   The output is displayed on the terminal, commands, procedures, and DO command-string sequence of SET BREAK commands are echoed, and a log file is not produced.

2. DBG> SET OUTPUT LOG

   The log file is to be created.

SET SCOPE

5.3.24 SET SCOPE Command

The SET SCOPE command specifies scopes to be searched to find a symbol. By default, the debugger searches for symbols (specified without pathnames) in the current scope (the scope that contains the current PC). If the debugger does not find the symbol, it searches its symbol table for a unique symbol. When you enter SET SCOPE, the debugger modifies its search rules: it searches the scopes in the order you specify. If it does not find the symbol in these scopes, it then searches for a unique symbol.

SET SCOPE command allows you to modify the default symbol search. You can specify the default scope (scope 0) in any position in the scope list. You can also specify that global symbols be searched in any position in the scope list by using the backslash (\) character.

If the debugger cannot find a symbol in the scopes specified with the SET SCOPE command, it searches all the modules in the active symbol table for a unique symbol. There is no way to suppress the search for a unique symbol.

Format

SET SCOPE scope-element [,scope-element ...]

Command Parameters

scope-element

Specifies the name of a scope, the digit 0, a backslash (\), or the number of a scope. These scope elements have the following meanings:

- Name of scope -- in general, consists of a module name and block or routine names separated by backslashes. The simplest case is a scope consisting of a module name. When you specify a name of scope, the debugger adds the symbols for the module specified to the symbol table if they are not already included.
- 0 -- specifies the current scope, the scope that contains the current PC. The current scope is used as the default scope if you enter the CANCEL SCOPE command. The current scope changes as different sections of your program are executed.
- \ -- specifies the scope consisting of all the global symbols defined in your image.
- Number of scope -- specifies scope by the level of active calls. The number 0 represents the scope currently being executed; the number 1 represents the scope that contained the call to the current scope; the number 2 represents the scope that contained the call before that.

Command Qualifiers

None.

Examples

1. DBG> SET SCOPE MOD3

   This command sets the scope to MOD3

2. You can also define a scope list to specify the order in which the debugger should search for symbols.

   DBG> SET SCOPE MOD1\MAR,MOD2\JAN,MOD2\FEB

   The command above instructs the debugger to search for symbols first in the context of routine MAR. If the debugger cannot find a specified symbol in MAR, the debugger continues the search in JAN and, if necessary, FEB.

   The scope defined in a SET SCOPE command becomes the default scope for all symbolic addresses until you explicitly change or cancel the scope.

   Note that when you explicitly use SET SCOPE to set the scope to a program unit name, the debugger implicitly performs a SET MODULE command. Therefore, symbols for the program unit specified in your SET SCOPE command are placed in the symbol table. However, if you use the debugger's default scope, you must also use the SET MODULE command (Section 5.3.22) to place symbols for the program unit other than the main program in the symbol table.

SET STEP

5.3.25 SET STEP Command

The SET STEP command specifies the current default step conditions. The SET STEP command controls how the unit of the program is to be executed (INSTRUCTION or LINE), whether a called procedure or a subroutine is treated as a series of instructions or as a single instruction (INTO or OVER). You can override the current default step condition by specifying a mode qualifier in the STEP command.

Format

SET STEP [qualifier(s)]

Command Qualifiers

INSTRUCTION
INTO
LINE
[NO]SYSTEM
OVER

Command Parameters

None.

Command Qualifiers

INSTRUCTION

Tells the debugger to step through the program VAX-11 MACRO instruction by instruction.

INTO

Tells the debugger to step through routines when one is called; that is, it is to step into the subprogram.

LINE

Tells the debugger to step through the program line by line. This is the default for VAX-11 PASCAL.

[NO]SYSTEM

Tells the debugger to count steps wherever they occur, including system address space. The default is NOSYSTEM.

OVER

Tells the debugger to ignore calls to routines as it steps through the program. That is, it is to step over the call.

Examples

```
DBG> SET STEP INTO
```

The debugger is to step into routines when it encounters routine calls.

SET TRACE

5.3.26 SET TRACE Command

The SET TRACE command specifies the location at which the debugger is to suspend program execution, display that the tracepoint has been reached, and resume execution.

Note that you cannot set tracepoints, watchpoints or breakpoints at the same location. The most recently issued command overrides any other.

Format

SET TRACE [[/qualifier]] [[address-expression]]

Command Qualifiers

/BRANCH
/CALL

Command Parameters

address-expression

Specifies the location at which the tracepoint is located.

Command Qualifiers

/BRANCH

Traces all branch, jump, and case instructions in the image.

/CALL

Traces all instructions that call routines in the image.

Examples

```
DBG> SET TRACE %LABEL 10

DBG> GO
routine start at MAIN\MAIN
trace at %label 10
   .
   .
   .
```

The tracepoint is established at label 10. In response to the GO command, the debugger displays the routine it is in and the starting point of the trace.

```
DBG>  SET TRACE CIRCLE

DBG> SET TRACE/BRANCH
DBG> SET TRACE/CALL
DBG> SHOW TRACE
tracepoint at CIRCLE\CIRCLE

tracing   /CALL instructions:  CALLS, CALLG, BSBW, BSBB, JSB, RSB
          and RET
tracing   /BRANCH instructions:  BNEQ, BEQL,  BGTR,  BLEQ,  BGEQ,
          BLSS,  BGTRU,  BLEQU, BVC, BVS, BGEQU, BLSSU, BRB, BRW,
          JMP, BBS, BBC, BBSS, BBCS, BBSC,  BBCC,  BBSSI,  BBCCI,
          BLBS,  BLBC,  ACBB,  ACBW,  ACBL,  ACBF,  ACBD, AOBLEQ,
          AOBLSS, SOBGEQ, SOBGTR, CASEB, CASEW and CASEL
```

SET TYPE

5.3.27 SET TYPE Command

The SET TYPE command sets the default types for the DEPOSIT and EXAMINE commands. The SET TYPE command sets the default type when there is no assigned data type in the symbol table. The SET TYPE/OVERRIDE command sets the default type for all data items in DEPOSIT and EXAMINE commands.

Format

SET TYPE [/qualifier] type-keyword

Command Qualifiers

/OVERRIDE

Command Parameters

type-keyword

Specifies the default data type. Type-keyword can be ASCII:length, BYTE, INSTRUCTION, LONG, or WORD.

- ASCII:length -- Specifies that the default data type is an ASCII string. The number of characters in the string is specified by length. If you do not specify length, the debugger assumes a length of 4.
- BYTE -- Specifies that the default data type is byte integer.
- INSTRUCTION -- Specifies that the default data type is instruction.
- LONG -- Specifies that the default data type is longword integer.
- WORD -- Specifies that the default data type is word integer.

Command Qualifiers

/OVERRIDE

Specifies that the specified data type should be used even when a symbol has an assigned data type in the symbol table. The CANCEL TYPE/OVERRIDE command cancels the effects of SET TYPE/OVERRIDE and specifies that the debugger should use the symbol's assigned default data type.

Examples

1. DBG> SET TYPE ASCII:8

   The type is set ASCII with a string length of 8 characters.

2. DBG> SET TYPE/OVERRIDE LONG

   The type LONG is to be used even if the symbol has an assigned data type.

SET WATCH

5.3.28 SET WATCH Command

The SET WATCH command allows you to monitor a specific location to determine when that location's value is changed.

Note that you cannot set watchpoints, tracepoints, and breakpoints at the same location. The most recently issued command overrides any other.

Format

SET WATCH var

Command Parameters

var

Specifies the location to be monitored. You cannot monitor variables declared in a procedure or function because these variables are allocated on the stack. However, variables declared at the program or module level or variables dynamically allocated by the NEW procedure can be monitored because these variables are allocated in static storage. If a variable is located on the same memory page as a PASCAL file variable, a file system error can occur when the file variable is accessed.

If the value of the location being monitored changes, the debugger stops program execution and displays the modifying statement, the location's old value, and its new value. The debugger then prompts for another command.

Command Qualifiers

None.

Examples

```
DBG> SET WATCH Area
DBG> GO
write to MAIN\AREA at PC=000002C4
       old value = 12
       new value = 16
DBG>
```

A watchpoint is set for the location Area in module MAIN. During execution of the program, the value of Area changed from 12 to 16.

SHOW BREAK

5.3.29 SHOW BREAK Command

The SHOW BREAK command displays the location of all breakpoints currently set in the program.

Format

SHOW BREAK

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG> SHOW BREAK
breakpoint at MOD1\MOD1\%LINE 3
```

The breakpoint is at LINE 3 of module MOD1.

SHOW CALLS

5.3.30 SHOW CALLS Command

The SHOW CALLS command displays a traceback list of procedures and function calls. This traceback list shows you the sequence of calls leading to the current subprogram from the most recent call to the first call.

Format

SHOW CALLS [[integer]]

Command Parameters

integer

Specifies the number of most recent calls to be displayed. If a call count is not specified, all the calls are displayed.

Command Qualifiers

None.

Examples

```
DBG> SHOW CALLS

module name    routine name      line    relative PC    absolute PC

   MAIN            MOD2           20     00000002       000008B0

   MAIN            MOD1           10     0000006C       00000B63
```

The first line in the report refers to the current call level. One call, at line 10, preceded this level.

SHOW LANGUAGE

5.3.31 SHOW LANGUAGE Command

The SHOW LANGUAGE command displays the current language the debugger is currently interpreting.

Format

SHOW LANGUAGE

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG> SHOW LANGUAGE
language:  PASCAL
```

The debugger displays the current language as PASCAL.

SHOW LOG

5.3.32 SHOW LOG Command

The SHOW LOG command displays the log file name and whether the debugger is writing to the log file.

Format

SHOW LOG

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG> SHOW LOG
logging to FEB29.TMP
```

The log file being used is FEB29.TMP.

SHOW MODE

5.3.33 SHOW MODE Command

The SHOW MODE command displays the current radix mode and the current symbolic or nonsymbolic mode.

The current modes are those last established by the SET MODE command or the default modes associated with the current language.

Format

SHOW MODE

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG> SHOW MODE
modes: symbolic, hexadecimal
```

The debugger displays that the mode is symbolic, hexadecimal.

SHOW MODULE

5.3.34 SHOW MODULE Command

The SHOW MODULE command displays the names of all program units whose symbols are potentially available to the symbol table.

The SHOW MODULE command displays the following information about each program or module in the image compiled with the /DEBUG qualifier.

- The module's name
- The symbols that are or are not in the active symbol table (yes/no)
- The language the module is written in
- The size of the module's active symbol table in bytes
- The total number of modules
- The amount of memory that can still be allocated for the symbol table

Format

SHOW MODULE

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG> SET MODULE Main, Mod1
DBG> SHOW MODULE

Module Name     Symbols     Language     Size

MAIN              yes        PASCAL      204

MOD1              yes        PASCAL      164

MOD2              no         PASCAL      184

total modules:  3.  remaining size:  61568.
```

In this example, there are three PASCAL modules. Modules Main and Mod1 have symbols in the active symbol table: Mod2 symbols are not included in the symbol table. The run-time symbol table has 61568 bytes of remaining space allocated.

SHOW OUTPUT

5.3.35 SHOW OUTPUT Command

The SHOW OUTPUT command displays the output configurations and reports whether the debugger is displaying output on the terminal, writing output to a log file, or verifying command procedures or DO command-string sequences.

Format

SHOW OUTPUT

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG> SHOW OUTPUT

output:  no verify, terminal, not logging into DEBUG.LOG
```

The debugger is not verifying command procedures or DO command-string sequences, the log file is not in use, and output is to a terminal.

SHOW SCOPE

5.3.36 SHOW SCOPE Command

The SHOW SCOPE command displays the routines that are currently set as scope.

Format

SHOW SCOPE

Command Parameters

None.

Command Qualifiers

None.

Examples

1. DBG> SHOW SCOPE
   scope: MOD2,MOD1

   The message the debugger returns shows that the current scope is set first to MOD2, then to MOD1. The debugger checks for a symbol in the module MOD2 first.

2. DBG> SHOW SCOPE
   scope: 0 [= MOD2\MULT]

   The symbol 0 shows that the current scope is the default scope; the default scope is routine MULT in program unit MOD2.

5.3.37 SHOW STEP Command

The SHOW STEP command displays the current default step conditions.

Format

SHOW STEP

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG> SHOW STEP
step type:  no system, by instruction, over routine calls
```

The debugger is skipping address reserved for the system, is stepping through the program by instruction, and is skipping over routines calls.

SHOW TRACE

5.3.38 SHOW TRACE Command

The SHOW TRACE command displays either the location (as a line number, label, or hexadecimal address) of all active tracepoints or a message indicating there are no active tracepoints.

Format

SHOW TRACE

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG> SHOW TRACE

tracepoint at %LINE 20
tracepoint at %LINE 25
```

The tracepoints are located at lines 20 and 25.

SHOW TYPE

5.3.39 SHOW TYPE Command

The SHOW TYPE command displays the current default data type. If you specify the /OVERRIDE qualifier, the debugger displays the current default override data type.

If the data type is /ASCII:length, the debugger displays length as a decimal number even if the current radix is not decimal.

Format

SHOW TYPE [/qualifier]

Command Qualifiers

/OVERRIDE

Command Parameters

None.

Command Qualifiers

/OVERRIDE

Displays the current override default data type.

Examples

```
DBG> SHOW TYPE
type:  long integer
```

The data type now being used is long integer.

SHOW WATCH

5.3.40 SHOW WATCH Command

The SHOW WATCH command displays all the watch points that are active and the size of each watchpoint in number of bytes. A watchpoint remains active until a CANCEL WATCH command is issued, or a SET BREAK or SET TRACE command is given for the same location, or the debugging session ends.

Format

SHOW WATCH

Command Parameters

None.

Command Qualifiers

None.

Examples

```
DBG> SHOW WATCH

Watchpoint at MOD\ROUT\AREA for 4 bytes
```

A watchpoint has been established for AREA in the routine ROUT in the module MOD. AREA is 4 bytes long.

STEP

5.3.41 STEP Command

The STEP command allows you to execute a predetermined number of statements.

You can specify the qualifiers each time you issue a STEP command, or you can use one SET STEP command to set the conditions for a series of STEP commands. Refer to Section 5.3.25 for the SET STEP command.

Format

STEP [[qualifier(s)]] [[integer]]

Command Qualifiers

INSTRUCTION
INTO
LINE
[NO]SYSTEM
OVER

Command Parameters

integer

Specifies the number of statements to be executed. If you specify 0 or omit the integer, a default of 1 is assumed. Note, however, that if you issue a STEP command while your program is stopped in a module whose symbols are not in the active symbol table, VAX-11 MACRO instructions (not PASCAL statements) are executed.

Command Qualifiers

INSTRUCTION

Tells the debugger to step through the program instruction by instruction.

INTO

Tells the debugger to step through routines when one is called. That is, it is to step into the subprogram.

LINE

Tells the debugger to step through the program line by line. This is the default for VAX-11 PASCAL.

[NO]SYSTEM

Tells the debugger to count steps wherever they occur, including system address space. The default is NOSYSTEM.

OVER

Tells the debugger to ignore calls to routines as it steps through the program. That is, it is to step over the call.

Examples

1. DBG> SET STEP INSTRUCTION, INTO, SYSTEM

   In this example, the debugger overrides the defaults applicable to PASCAL programs. When you subsequently issue a STEP command without qualifiers, these qualifiers (INSTRUCTION, INTO, and SYSTEM) are in effect. You can, however, supersede them individually by including a qualifier in a STEP command.

2. DBG> STEP/LINE 10

   This example tells the debugger to execute 10 lines, regardless of the SET STEP command.

   Sometimes it is advisable to use STEP to execute only a few instructions at a time. To execute many instructions and stop, use a SET BREAK command to set a breakpoint, and then issue a GO command to execute all statements up to the breakpoint.

5.3.42 @file-spec Command

The @file-spec command directs the debugger to begin taking commands from a specified command procedure.

Format

@file-spec

Command Parameters

file-spec

Specifies the name of a VAX/VMS file containing the debugger commands. The default file type is COM.

Command Qualifiers

None.

Examples

The command procedure DBGPROG.COM contains the following debugger commands:

```
SET BREAK %LINE 10
GO
EXAMINE A^.NAME
EXAMINE A^.ADDRESS
EXAMINE A^.SSNUMBER
GO
```

During a debugging session to execute the commands in this file, you would enter:

```
DBG> @DBGPROG.COM
```

The commands from this file are executed as if you had entered each of the commands yourself.

5.4 A DEBUGGING EXAMPLE

This section presents a debugging example using the program Flight_Reservation to track reservations for 10 different flights. The program prompts for input of a flight number, passenger name, and class of reservation desired. When 0 is entered for the flight number, the program should terminate. A bug has been included in this program for the purpose of demonstrating how to use the VAX-11 Symbolic Debugger to find a logic error in a program.

Figure 5-3 is the source listing for Flight_Reservation and Figure 5-4 is an actual interactive debugging session. The circled numbers correspond to the text that follows the figures and gives a detailed description of the debugging session.

```
PROGRAM Flight_Reservations (INPUT, OUTPUT);
CONST
    Alfa_Length             = 30;           { Maximum length of a name }
    Max_Flights             = 10;           { 10 different flights possible }
    Max_Reservations = 100;          { Maximum number of reservations per flight }

TYPE
    Alfa             = PACKED ARRAY[1..Alfa_Length] OF CHAR;
    Flight_Class     = (Standby,Coach,Business,First);
    Reservation_Pointer = ^Reservation_Record;
    Reservation_Record = RECORD
                                Name             : Alfa;
                                Class            : Flight_Class;
                                Next_Reservation: Reservation_Pointer;
                         END;

    Flight_Description = RECORD
                                Flight_Number               : INTEGER;
                                Number_Of_Reservations      : INTEGER;
                                Reservations                : Reservation_Pointer;
                         END;
    Flight_Range     = 1..Max_Flights;
    Flight_List      = ARRAY[Flight_Range] OF Flight_Description;

VAR
    Current_Reservation      : Reservation_Pointer;
    Entered_Flight           : INTEGER;    { Requested flight number }
    Accepting_Reservations   : BOOLEAN;    { True = Still accepting reservations }
    Flights                  : Flight_List;

PROCEDURE Initialize_Flights;
                { PURPOSE:
                        Initialize the flight reservation array}

VAR
    Flight   : Flight_Range;         { Current flight record }

BEGIN { Initialize_Flights }

    FOR Flight := 1 TO Max_Flights DO
       WITH Flights[Flight] DO
           BEGIN
              Flight_Number                 := Flight;
              {The array index is the flight number }
              Number_Of_Reservations        := 0;
              Reservations                  := Nil;
           END;

END; { Initialize_Flights }
```

Figure 5-3 Source Program for the Program Flight_Reservations

```
FUNCTION Flight_Available (Flight_To_Check : INTEGER) : BOOLEAN;
         { PURPOSE:
                Check whether or not a particular flight is open
           PARAMETERS:
                Flight_To_Check - Flight number to check for }

LABEL 1;

VAR
    Flight    : INTEGER;          { Index into Flights array }

BEGIN { Flight_Available }

    Flight_Available := False; { Assume the flight won't be available }

    FOR Flight := 1 TO Max_Flights DO
       WITH Flights[Flight] DO
           IF Flight_To_Check = Flight_Number
           THEN
               IF (Number_Of_Reservations + 1) <= Max_Reservations
               THEN
                   BEGIN
                      Flight_Available := True;
                      GOTO 1;                { Exit }
                   END;
1:
END; { Flight_Available }


FUNCTION Reservations_Available : BOOLEAN;

                { PURPOSE:
                       Check whether or not all flights are full
                  FUNCTION VALUE:
                       True  = There are still open flights
                       False = There are no more open flights}

LABEL 1;

VAR
    Flight    : INTEGER;       { Index in Flights array }

BEGIN { Reservations_Available }

  Reservations_Available := False;    { Assume no more open flights }

  FOR Flight := 1 TO Max_Flights DO
      WITH Flights[Flight] DO
         IF Number_Of_Reservations < Max_Reservations
         THEN
             BEGIN
                 Reservations_Available := True;
                 GOTO 1;                { Exit }
             END;
1:
END; { Reservations_Available }
```

Figure 5-3(Cont.) Source Program for the Program Flight_Reservations

```
FUNCTION Read_Reservation (VAR Reservation : Reservation_Pointer) : BOOLEAN;
        { PURPOSE:
              Read reservation information from the terminal.
          PARAMETERS:
              Reservation  - Pointer to a reservation record (output parameter)
          FUNCTION VALUE:
              True = The reservation can be made
              False        = The reservation cannot be made }

BEGIN { Read_Reservation }

    { Allocate space for the reservation }

  NEW(Reservation);

  WITH Reservation^ DO
    BEGIN
      Next_Reservation := Nil;

      { Prompt for and read reservation information }

      WRITELN;
      WRITE ('Enter Flight Number:');
      READ (Entered_Flight);
      IF Entered_Flight = 0 THEN
          Read_Reservation := False
      ELSE IF NOT Flight_Available(Entered_Flight)
      THEN
          BEGIN
            WRITELN('    Flight ', Entered_Flight: 1, ' is full.');
            Read_Reservation := False;
          END
      ELSE

          { Read the rest of the reservation information }

          BEGIN
            WRITELN;
            WRITE('               Name: ');
            READ(Name);
            WRITE('              Class: ');
            READ(Class);
            Read_Reservation := True;
          END;

      END; { WITH Current_Reservation^ }

END; { Read_Reservation }
```

Figure 5-3(Cont.) Source Program for the Program Flight_Reservations

```
PROCEDURE Enter_Reservation (Reservation : Reservation_Pointer);
                { PURPOSE:
                        Enters a reservation in the reservation data base
                  PARAMETERS:

                                Reservation     - Pointer to the reservation to enter }

LABEL 1;

VAR
    Flight    : INTEGER;       { Index into Flights array }

BEGIN { Enter_Reservation }

    FOR Flight := 1 TO Max_Flights DO
       WITH Flights[Flight] DO
           IF Flight_Number = Entered_Flight
           THEN
               BEGIN
                  Number_Of_Reservations        := Number_Of_Reservations + 1;
                  Reservation^.Next_Reservation := Reservations;
                  Reservations                  := Reservation;
                  GOTO 1;              { Exit }
               END;
1:
END; { Enter_Reservation }


BEGIN { Flight_Reservations }

  Initialize_Flights;

  Accepting_Reservations := True;
  WHILE Accepting_Reservations DO
     BEGIN
        IF Read_Reservation(Current_Reservation)
        THEN
           Enter_Reservation(Current_Reservation);
        IF Accepting_Reservations
        THEN
           Accepting_Reservations := Reservations_Available;
  END;

END. { Flight_Reservations }
```

Figure 5-3 (Cont.) Source Program for the Program Flight_Reservations

```
$RUN FLIGHT

                        VAX-11 DEBUG VERSION 2.3

%DEBUG-I-INITIAL, language is PASCAL, module set to 'FLIGHT_RESERVATIONS' (1)

DBG> GO

                ENTER  Flight_Number : 1
                              Name : Smith
                             Class : First
                                                    (2)
                ENTER  Flight_Number : 0
                ENTER  Flight_Number :^Z

%PAS-F-ERRACCFIL, error in accessing file PAS$INPUT
%RMS-F-EOF, end of file detected

DBG> EXIT (3)


$RUN FLIGHT

                        VAX-11 DEBUG VERSION 2.3

%DEBUG-I-INITIAL, language is PASCAL, module set to 'FLIGHT_RESERVATIONS'

DBG> SET BREAK Read_Reservation
                                                    (4)
DBG> SET BREAK Enter_Reservation

DBG> SET BREAK Reservations_Available

DBG> GO                                             (5)
routine start at FLIGHT_RESERVATIONS
routine break at FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\READ_RESERVATION

DBG> STEP 7
routine start at  FLIGHT_RESERVATIONS\READ_RESERVATION (6)
                ENTER Flight_Number: 1
stepped to FLIGHT_RESERVATIONS\READ_RESERVATION %LINE 140

DBG> STEP
start at FLIGHT_RESERVATIONS\READ_RESERVATION %LINE 140
stepped to FLIGHT_RESERVATIONS\READ_RESERVATION %LINE 142

DBG> EXAMINE Entered_Flight
FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\ENTERED_FLIGHT: 1

DBG> STEP 16                                                (7)
start at FLIGHT_RESERVATIONS\READ_RESERVATION %LINE 142
                Name: Smith
               Class: First
routine break at FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\ENTER_RESERVATION
```

Figure 5-4 Interactive Debugging Session for the Program Flight_Reservations

❶ The program Flight_Reservations is compiled and linked using the /DEBUG qualifier. The debugger is entered by using the command RUN FLIGHT.

The debugger responds with its version number, the date, a message that the initial programming language is PASCAL and the main module is Flight_Reservations.

The debugger now has control of the program, and the prompt DBG> replaces the dollar sign ($) prompt on the screen.

❷ In this first session, the program Flight_Reservations is executed using only the debugger command GO. This allows you to observe the functioning of the program without interruption. After receiving the GO command, the debugger gives control to the program which executes as it would normally. The program prompts for input (the flight number, name, class). On the next request for a flight number, a 0 is entered which should signal the program to stop requesting input and to terminate execution. However, the program fails to act as expected because another prompt for input occurs. The CTRL/Z command is issued to exit from the program and return control to the debugger.

❸ To reenter the program, first exit from the debugger using the EXIT command and then, reissue the RUN command.

❹ To debug the program, the next step is to set breakpoints at places where results need to be verified, such as functions and procedures. In this session, the breakpoints are set at Read_Reservation, Enter_Reservation, and Reservations_Available.

❺ Once the breakpoints have been established, the command GO is issued. The debugger responds with the starting point address and execution of the program stops when a breakpoint is reached. The address of the first breakpoint is then displayed.

❻ The debugger halts at the breakpoint and prompts for a command. The command STEP 7 is issued. This command instructs the debugger to continue program execution. The next seven lines of code from the routine Read_Reservation are processed. Program execution starts at the BEGIN instruction of the function Read_Reservation.

The debugger displays the starting address in the Read_Reservation module. The program requests input for the variable Entered_Flight. The value assigned to the variable Entered_Flight is verified using the EXAMINE command. The debugger responds with the value 1.

❼ The command STEP 16 continues execution of the program until the 16th line of executable code from the current address is reached. The debugger responds with the starting address in the module and the program prompts for a name and class.

```
DBG> STEP                                                    (8)
routine start at FLIGHT_RESERVATIONS\ENTER_RESERVATION
stepped to FLIGHT_RESERVATIONS\ENTER_RESERVATION %LINE 178

DBG> EXAMINE Reservation^.Name
FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\ENTER_RESERVATION\RESERVATION^.NAME: 'Smith'  (9)

DBG> EXAMINE Reservation^.Class
FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\ENTER_RESERVATION\RESERVATION^.CLASS: FIRST

DBG> SET BREAK Enter_Reservation\%LABEL 1

DBG> GO                                                      (10)
start at FLIGHT_RESERVATIONS\ENTER_RESERVATION %LINE 178
break at FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\ENTER_RESERVATION\%LABEL 1

DBG> EXAMINE Flights[1].Number_Of_Reservations
FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\FLIGHTS[1].NUMBER_OF_RESERVATIONS: 1

DBG> EXAMINE Flights[1].Reservations^.Name
FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\FLIGHTS[1].RESERVATIONS^.NAME: ' Smith'  (11)

DBG> EXAMINE
FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\FLIGHTS[1].RESERVATIONS^.CLASS: FIRST

DBG> GO
start at FLIGHT_RESERVATIONS\ENTER_RESERVATION %LINE 189                (12)
routine break at FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\RESERVATIONS_AVAILABLE

DBG> STEP 6
routine start at FLIGHT_RESERVATIONS\RESERVATIONS_AVAILABLE              (13)
stepped to FLIGHT_RESERVATIONS\RESERVATIONS_AVAILABLE %LINE 108

DBG> EXAMINE Reservations_Available
FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\RESERVATIONS_AVAILABLE:  TRUE

DBG> GO
start at FLIGHT_RESERVATIONS\RESERVATIONS_AVAILABLE %LINE 108            (14)
routine break at FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\READ_RESERVATION

DBG> STEP 7
routine start at FLIGHT_RESERVATIONS\READ_RESERVATION
      ENTER Flight_Number : 0
stepped to FLIGHT_RESERVATIONS\READ_RESERVATION %LINE 140

DBG> STEP
start at FLIGHT_RESERVATIONS\READ_RESERVATION %LINE 140                  (15)
stepped to FLIGHT_RESERVATIONS\READ_RESERVATION %LINE 141

DBG> EXAMINE Entered_Flight
FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\ENTERED_FLIGHT: 0

DBG> GO
start at FLIGHT_RESERVATIONS\READ_RESERVATION %LINE 141                  (16)
routine break at FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\RESERVATIONS_AVAILABLE
```

Figure 5-4 (Cont.) Interactive Debugging Session for the Program Flight_Reservations

❽ The STEP command is given.

❾ The fields Reservation^.Name and Reservation^.Class are examined.

❿ A new breakpoint is set at LABEL 1 in the module Enter_Reservation and the GO command continues execution of the procedure Enter_Reservation. The contents of the array Flights are verified by the EXAMINE command when LABEL 1 is reached.

⓫ The contents of the array Flights are displayed by the use of EXAMINE commands. In each case, the debugger responds with the pathname, array element name, and the value contained in that element.

The field Reservations^.Class is examined by simply entering EXAMINE followed by a return. The debugger displays the contents of the next logical successor, which is Reservation^.Class.

⓬ Program execution is resumed with the GO command and continues until the debugger encounters the next breakpoint. The debugger responds with the starting point address and the address of the breakpoint reached in Reservation_Available.

⓭ The function Reservations_Available is executed when the command STEP 6 is given. The command EXAMINE displays the contents of the field Reservations_Available as TRUE.

⓮ Program execution is resumed with the GO command until the breakpoint at the beginning of the routine Read_Reservation is encountered.

The command STEP 7 is given to step through seven lines of executable code of the routine Read_Reservations to the point where the prompt for input is displayed. The program requests the input for Flight_Number. A 0 is entered indicating that the last reservation has been entered. The debugger displays the current address after executing six lines in the routine Read_Reservation.

⓯ The STEP command is given and the contents of Entered_Flight are examined. Entered_Flight contains a 0.

⓰ The GO command causes program execution to resume until the breakpoint Reservations_Available is encountered.

```
DBG> EXAMINE Accepting_Reservations                                    (17)
FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\ACCEPTING_RESERVATIONS: TRUE

DBG> SET BREAK Reservations_Available\%LABEL 1 (18)

DBG> GO                                (19)
routine start at FLIGHT_RESERVATIONS\RESERVATIONS_AVAILABLE
break at FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\RESERVATIONS_AVAILABLE\%LABEL 1

DBG> EXAMINE Reservations_Available
FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\RESERVATIONS_AVAILABLE\RESERVATIONS_AVAILABLE: TRUE

DBG> DEPOSIT Reservations_Available = FALSE (20)

DBG> EXAMINE Reservations_Available
FLIGHT_RESERVATIONS\FLIGHT_RESERVATIONS\RESERVATIONS_AVAILABLE\RESERVATIONS_AVAILABLE: FALSE

DBG> CANCEL BREAK/ALL (21)

DBG> GO                                                                (22)
start at FLIGHT_RESERVATIONS\RESERVATIONS_AVAILABLE %LINE 111
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'

DBG> EXIT (23)
$
```

Figure 5-4 (Cont.) Interactive Debugging Session for the Program Flight_Reservations

⑰ The contents of the field Accepting_Reservations are examined. The debugger displays the contents of Accepting_Reservations as TRUE. The bug has been found because this field should have been set to FALSE when the flight number 0 was entered in the module Read_Reservation.

To verify that this is the bug, the contents of the field Reservations_Available are changed and the program is run to completion as described below.

⑱ A breakpoint at LABEL 1 is established in the module Reservations_Available. The module and label names are used as the address in a SET BREAK command.

⑲ Program execution is started with the GO command. The contents of Reservations_Available are displayed as being TRUE.

⑳ The DEPOSIT command places the value FALSE into Reservations_Available. The change is verified by the EXAMINE command.

㉑ The CANCEL BREAK/ALL command terminates all breakpoints, to allow the program to run to completion without halting.

The program must be edited, recompiled, and relinked to test whether this error was in fact the only incorrect logic in the program.

㉒ To verify, the program execution is started with the GO command. At completion of the run, the debugger responds with the message that normal and successful completion of the program has occurred.

㉓ The EXIT command is given to end the debugger session.

CHAPTER 6

INPUT AND OUTPUT

This chapter describes input and output (I/O) for VAX-11 PASCAL. In particular, it provides information about PASCAL I/O in relation to VAX-11 Record Management Services (VAX-11 RMS). The topics covered include:

- Logical names
- PASCAL file characteristics
- PASCAL record formats
- OPEN procedure parameters
- Local interprocess communication by means of mailboxes
- Remote communication by means of DECnet-VAX

6.1 LOGICAL NAMES

The VAX/VMS operating system provides the logical name mechanism as a way of making programs device and file independent. If you use logical names, your program need not specify the particular device on which a file resides or the particular file that contains data. Specific devices and files can be defined at run time.

A logical name is an alphanumeric string, up to 63 characters long, that you specify in place of a file specification. The operating system provides a number of predefined logical names, already associated with file specifications. Table 6-1 lists the logical names of special interest to PASCAL users.

Logical names provide great flexibility because they can be associated not only with a complete file specification, but also with a device, a device and a directory, or even another logical name.

Table 6-1
Predefined System Logical Names

Name	Meaning	Default
SYS$DISK	Current default device	As specified by the user
SYS$ERROR	Default error message file	User's terminal (interactive); batch log file (batch)
SYS$COMMAND	Default command input stream	User's terminal (interactive); batch command file (batch)
SYS$INPUT	Default input file	User's terminal (interactive); batch command file (batch)
SYS$OUTPUT	Default output file	User's terminal (interactive); batch log file (batch)

You can create a logical name and associate it with a file specification by means of the ASSIGN command. Thus, before program execution, you can associate the logical names in your program with the file specifications appropriate to your needs. For example:

```
$ ASSIGN  DBA0:[SMITH]Test.DAT;2  Data
```

This command creates the logical name Data and associates it with the file specification DBA0:[SMITH]Test.DAT;2. The system uses this file specification when it encounters the logical name Data during program execution. For example:

```
OPEN (Indata, 'Data',HISTORY:= OLD);
```

In executing this PASCAL statement, the system uses the file specification DBA0:[SMITH]Test.DAT;2 for the logical name Data. To specify a different file when you execute the program again, issue another ASSIGN command. For example:

```
$ ASSIGN DBA2:[JONES]Real.DAT;7  Data
```

This command associates the logical name Data with a different file specification and replaces the previous logical name assignment. The OPEN statement above will now refer to the file DBA2:[JONES]Real.DAT;7.

You can also assign logical names with the MOUNT and DEFINE commands (see the VAX/VMS Command Language User's Guide).

6.2 FILE CHARACTERISTICS

A clear distinction must be made between the way files are organized and the way records are accessed.

The term "file organization" applies to the way records are physically arranged on a storage device. The term "record access" refers to the method used to read records from or write records to a file, regardless of the file's organization. A file's organization is specified when the file is created and cannot be changed. Record access is specified each time the file is opened and can vary.

6.2.1 File Organization

VAX-11 PASCAL supports sequential file organization. Sequential files consist of records arranged in the order in which they are written to the file (the first record written is the first record in the file, the second record written is the second record in the file, and so on). Records can only be added at the end of a new file, with a HISTORY equal to NEW.

6.2.2 Record Access

You specify record access mode as a parameter to the OPEN procedure. VAX-11 PASCAL provides two ways of accessing records:

- Sequential
- Direct

If you select sequential access mode, records are written to or read from the file, starting at the beginning and continuing through the file one record after another.

Sequential access to a file means that you can read a particular record only after reading all the records preceding it. New records can be written only at the end of a file that is open for sequential access.

If you select direct access mode, you can specify the order in which records are accessed. Each FIND procedure call must include the relative record number indicating the record to be read. You can directly access a file only if it contains fixed-length records, resides on disk, and is open for input (reading).

6.3 RECORD FORMATS

Records are stored in one of two formats:

- Fixed length
- Variable length

You can access fixed-length records in either sequential or direct mode. Variable-length records can be accessed only in sequential mode.

6.3.1 Fixed-Length Records

When you specify fixed-length records (see Section 6.4.4), you are specifying that all records in the file contain the same number of bytes. A file opened for direct access must contain fixed-length records, to allow the record location to be computed correctly.

6.3.2 Variable-Length Records

Variable-length records can contain any number of bytes, up to the buffer size specified when the file was opened. Variable-length records are prefixed by a count field, indicating the number of bytes in the record. The count field comprises 2 binary bytes on a disk device and 4 decimal digits on magnetic tape. The value stored in the count field indicates the number of data bytes in the record.

6.4 OPEN PROCEDURE PARAMETERS

This section supplements the description of the OPEN procedure that appears in the VAX-11 PASCAL Language Reference Manual. In particular, it describes how the VAX-11 Record Management Services (VAX-11 RMS) affect VAX-11 PASCAL. For more information, refer to the VAX-11 Record Management Services Reference Manual.

The OPEN procedure has the following formats:

1.
```
OPEN (file-variable, 'VAX/VMS-file-spec',
file-status,
record-length,
record-access-mode,
record-type,
carriage-control);
```

2.
```
OPEN (FILE_VARIABLE := file-variable
     [,FILE_NAME := file-name]
     [,HISTORY := file-status]
     [,RECORD_LENGTH := record-length]
     [,RECORD_ACCESS_MODE := record-access-mode]
     [,RECORD_TYPE := record-type]
     [,CARRIAGE_CONTROL := carriage-control]
```

File status, record length, record-access mode, record type, and carriage control are VAX-11 RMS-dependent attributes described in this section.

6.4.1 File Status or History

The file-status or history parameter indicates whether the specified file exists or must be created. The possible values for this parameter are:

```
NEW
OLD
```

A file status of NEW indicates that a new file must be created with the specified characteristics. NEW is the default value.

A file status of OLD indicates that an existing file is to be opened. An error occurs if the file cannot be found. A type of OLD is illegal for internal files, which are newly created each time the declaring program unit is executed.

6.4.2 Record Length

The record length parameter is an integer that specifies the maximum record size in bytes for a text file. The default for a VAX-11 PASCAL text file is 133 bytes. For a nontext file this parameter has no meaning.

6.4.3 Record-Access Mode

The record-access mode parameter specifies the mode of access to records in the file. The possible values for this parameter are:

```
SEQUENTIAL
DIRECT
```

The default record access mode is SEQUENTIAL. In SEQUENTIAL mode, you can access files that have fixed- or variable-length records.

DIRECT mode allows you to use the FIND procedure to read files with fixed-length records. You cannot access a file with variable-length records in DIRECT mode.

6.4.4 Record Type

The record-type parameter specifies the structure of records in a file. The possible values for this parameter are:

```
FIXED
VARIABLE
```

A value of FIXED indicates that all records in the file have the same length. A value of VARIABLE indicates that the records within the file can vary in length. VARIABLE is the default for a new file. For an existing file, the default is the record type associated with the file at its creation.

6.4.5 Carriage Control

The carriage-control parameter specifies the type of carriage control in effect for an output text file. The possible values for this parameter are:

```
LIST
CARRIAGE
FORTRAN
NOCARRIAGE
NONE
```

A value of LIST indicates that each record is preceded by a line feed and followed by a carriage return when the file is output to a terminal or line printer. LIST is equivalent to the VAX-11 RMS record attribute CR. LIST is the default for all text files.

The values of CARRIAGE or FORTRAN indicate that the first byte of each record contains a carriage control character. CARRIAGE or FORTRAN is equivalent to the VAX-11 RMS record attribute FTN. Table 6-2 lists the carriage-control characters.

Table 6-2
Carriage Control Characters

Character	Meaning
'+'	Overprinting: starts output at the beginning of the current line
space	Single spacing: starts output at the beginning of the next line
'0'	Double spacing: skips a line before starting output
'1'	Paging: starts output at the top of a new page
'$'	Prompting: starts output at the beginning of the next line, and suppresses carriage return at the end of the line

Characters other than those in Table 6-2 are ignored.

The values of NOCARRIAGE or NONE indicate that the record contains no carriage control information. NOCARRIAGE or NONE is equivalent to the VAX-11 RMS record attribute PRN with all bits equal to zero.

6.5 LOCAL INTERPROCESS COMMUNICATION: MAILBOXES

The exchange of data between processes is often useful; for example, to synchronize execution or to send messages.

A mailbox is a record-oriented, pseudo-I/O device that allows data to be passed from one process to another. Mailboxes are created by the Create Mailbox (SYS$CREMBX) system service (see Section 7.2.1 for an example using SYS$CREMBX). This section describes how to send and receive data using mailboxes.

Data transmission by means of mailboxes is synchronous; that is, a PASCAL program that writes a message to a mailbox must wait until that message is read, and a program that reads a message from a mailbox must wait until a message is written. When the writing program closes the mailbox, an end-of-file condition is returned to the reading program. VAX-11 RMS ensures that the message transmission is complete before it returns control to the user program.

For example:

```
PROGRAM Mail (MBX, OUTPUT);

    VAR MBX : TEXT;

    BEGIN
      OPEN (MBX, 'Mailbox', HISTORY := OLD, ACCESS_METHOD := SEQUENTIAL);
      RESET (MBX);
      WHILE NOT EOF (MBX) DO
      BEGIN
        WHILE NOT EOLN (MBX) DO
          BEGIN
            WRITE (MBX^);
            GET (MBX)
          END;
        WRITELN;
        GET (MBX)
      END;
      CLOSE (MBX)
    END.
```

This program reads messages from a mailbox known by the logical name Mailbox. The messages are lines of text, which are then printed at the user's terminal.

6.6 COMMUNICATING WITH REMOTE COMPUTERS:NETWORKS

If your computer is one of the nodes in a DECnet network, you can use VAX-11 PASCAL I/O procedures to communicate with other nodes in the network. These procedures allow you to exchange data with a program at the remote computer (task-to-task communication) and to access files at the remote computer (resource sharing).

Both task-to-task communication and resource sharing between systems are transparent. That is, these intersystem exchanges do not appear to be different from local interprocess and file-access exchanges.

To communicate across the network, specify a node name as the first element of a file specification. For example:

```
Boston::DBA0:[SMITH]Test.DAT;2
```

Remote task-to-task communication requires a special form of file specification. You must use the notation TASK= in place of the device name and supply the task name, as in the following example:

```
Boston::"TASK=UNA"
```

The example specifies the task named UNA on the BOSTON node of the network.

The following program fragment shows how messages can be received from a remote program by means of VAX-11 PASCAL I/O procedures.

```
OPEN (FILE_VARIABLE:=Netjob,
FILE_NAME:='Boston::"TASK=UNA"',HISTORY:=OLD);
RESET (Netjob);
Rdat := Netjob^;
NET_Proc (Rdat,Wrtdat);
CLOSE (Netjob);
```

The effect of these statements is to establish a link with a job (TASK) named UNA at the node BOSTON and to receive a component from the file variable associated with the remote program. The variable RDAT contains the data. Then the procedure Net_Proc is called to process the data and the link is broken.

The next example shows how you can write a remote file using VAX-11 PASCAL I/O procedures.

```
PROGRAM UPDATE (Newdat, Branch);

  VAR Newdat : FILE OF INTEGER;
      Branch : FILE OF INTEGER;

  BEGIN

    OPEN (Newdat,'NEWDAT.DAT',HISTORY:=OLD);
    RESET (Newdat);
    OPEN (Branch,'Nashua"PLUGH XYZZY"::Master.Dat',HISTORY:=NEW);
    REWRITE (Branch);
    WHILE NOT EOF(Newdat) DO

          BEGIN
            Branch^:=Newdat^;
            GET (Newdat);
            PUT (Branch)
          END;

    CLOSE (Branch);
    CLOSE (Newdat)

  END.
```

The sample program writes records in a remote file at the node Nashua. It reads data from a local file known by the logical name Newdat and writes the data across the network to the remote file Master.DAT in the directory [PLUGH] with password XYZZY.

If you use logical names in your program, you can equate the logical names with either local or remote files. Thus, if your program normally accesses a remote file, and the remote node becomes unavailable, you can bring the volume set containing the file to the local site. You can then mount the volume set and assign the appropriate logical name. For example:

Remote Access

```
$ ASSIGN REM::APPLIC_SET:file-name  LOGIC
```

Local Access

```
$ MOUNT device-name APPLIC_SET
$ ASSIGN APPLIC_SET:file-name  LOGIC
```

The MOUNT and ASSIGN commands are described in detail in the VAX/VMS Command Language User's Guide.

DECnet capabilities are described in the DECnet-VAX Reference Manual.

CHAPTER 7

CALLING CONVENTIONS

In the context of the VAX/VMS operating system, a procedure is a routine entered by a CALL instruction. In a PASCAL program, such a routine can be a function or procedure written in PASCAL, a function or procedure written in some other language, a VAX/VMS system service, or a VAX-11 Run-Time Library procedure. In many cases, procedures perform calculations that are used widely and repeatedly in many applications. In PASCAL, you can write each procedure once and call it from many other programs.

This chapter describes how to call procedures that are not written in PASCAL and provides information on calling VAX/VMS system services and VAX-11 Run-Time Library procedures. See the VAX-11 PASCAL Language Reference Manual for information on defining and invoking PASCAL functions and procedures.

The material presented here assumes some knowledge of procedure calling and argument passing mechanisms. You should be familiar with these subjects before you attempt to use the features described in this chapter. Refer to the VAX-11 Run-Time Library Reference Manual and the VAX-11 Architecture Handbook for more information.

7.1 VAX-11 PROCEDURE CALLING STANDARD

Programs compiled by the VAX-11 PASCAL compiler conform to the standard defined for VAX-11 procedure calls (see Appendix C of the VAX-11 Architecture Handbook). This standard prescribes how arguments are passed, how function values are returned, and how procedures receive and return control. VAX-11 PASCAL also provides features that allow programs to call system services and procedures written in other native-mode languages supported by VAX/VMS.

VAX-11 PASCAL uses the VAX-11 CALLS instruction to call procedures. Appendix B illustrates the events that occur during a procedure call and show the structure of the run-time stack after each event.

7.1.1 Argument Lists

Each time you call a procedure, VAX-11 PASCAL constructs an argument list. The VAX-11 procedure calling standard defines an argument list as a sequence of longword (4-byte) entries. The first byte of the first entry in the list is an argument count, which indicates how many arguments follow in the list.

The arguments in the list are based on the passing mechanisms specified in the formal parameter list and the values in the actual parameter list. The argument list contains the arguments actually passed to the procedure.

7.1.2 Parameter Passing Mechanisms

Non-PASCAL procedures require arguments as addresses, immediate values, or descriptors. The VAX-11 procedure calling standard defines three mechanisms by which arguments are passed to procedures:

1. By-reference -- The argument list entry is the address of the value.

2. By-immediate-value -- The argument list entry is the value.

3. By-descriptor -- The argument list entry is the address of a descriptor of the value.

The following sections describe what you must specify in your VAX-11 PASCAL program to correctly pass arguments to non-PASCAL subprograms using each of these mechanisms. Note that this information pertains only to subprograms written in languages other than PASCAL. For information about passing arguments to PASCAL subprograms from non-PASCAL programs, see Section 7.1.5. Refer to the VAX-11 PASCAL Language Reference Manual for a description of parameter passing between PASCAL subprograms.

7.1.2.1 **By-Reference Mechanism** - The by-reference mechanism passes the address of the actual parameter. By default, PASCAL uses this mechanism to pass all parameters except dynamic array parameters. You can invoke the by-reference mechanism in two ways:

1. By omitting the mechanism specifier from the formal parameter list. This syntax implies PASCAL by-value semantics.

2. By using the VAR specifier in the formal parameter list. This syntax implies PASCAL by-reference semantics.

If you omit the mechanism specifier, PASCAL passes the address of the actual parameter. PASCAL expects the called procedure to copy the value from the specified address to local storage. You should use this method only if the called procedure does not change the value of the corresponding actual parameter. When you omit the mechanism specifier, the actual parameter can be an expression.

For example, the following function declaration and corresponding function call use this method:

```
FUNCTION MTH$TANH (Angle : REAL):REAL;  EXTERN;

Tanh := MTH$TANH (Radians);
```

This example declares the VAX-11 Run-Time Library function MTH$TANH as an external subprogram. The MTH$TANH function returns, as a real value, the hyperbolic tangent of its argument. The input parameter to this function is the size of the angle (in radians), and it must be passed by-reference. Because the function MTH$TANH does not change the value of the angle, you can omit the mechanism specifier when you declare the function. The returned value is assigned to the variable

Tanh by the assignment statement shown. (See Section 7.3 for more information on calling Run-Time Library procedures.)

Use the VAR specifier to pass an actual parameter that can be changed by the execution of the procedure. You must use VAR when passing a file variable as a parameter. The VAR specifier is also useful to prevent the copying of large parameters. Specify VAR in the following format:

```
VAR formal-parm-list : type;
```

The formal parameter list specifies one or more formal parameters of the indicated type. Each of these parameters will be passed using the by-reference mechanism and with by-reference semantics.

When you call the procedure, the argument list contains the address of the value to be passed. The actual parameter must be a variable or a component of an unpacked, structured variable; constants, expressions, procedure names, and function names are not allowed.

The following declarations and corresponding function call show how to pass an address to an external routine.

```
TYPE Bit64 = PACKED ARRAY [1..2] OF INTEGER;

VAR Systime : Bit64;

FUNCTION SYS$GETTIM (VAR Bintim :  Bit64) : INTEGER;  EXTERN;

STATUS := SYS$GETTIM (Systime);
```

This example declares the Get Time (SYS$GETTIM) system service, which returns the system time. The actual parameter Systime is a 64-bit variable into which the system service writes the time.

7.1.2.2 **By-Immediate-Value Mechanism** - VAX/VMS system services and Run-Time Library procedures sometimes require that the calling program pass an immediate value, that is, the value itself. To direct PASCAL to pass a value instead of an address, use the %IMMED mechanism specifier, as follows:

```
%IMMED formal-parm-list : type;
```

The formal parameter list specifies one or more formal parameters of the indicated type. Variables that require more than 32 bits of storage, including all file variables, cannot be passed as immediate values. %IMMED can be used for value parameter only. You can use %IMMED only when declaring external non-PASCAL routines.

When you call the routine, the actual parameter list contains the value of each parameter for which you specified %IMMED. The actual parameter can be a constant, a variable, or an expression. Note that %IMMED can also modify procedure and function formal names, as described in Section 7.1.3.

The following declarations and corresponding function call show how to pass an immediate value to a system service procedure.

```
VAR Event_Flag : INTEGER;

FUNCTION SYS$WAITFR (%IMMED EFN : INTEGER) : INTEGER; EXTERN;

STATUS := SYS$WAITFR (Event_Flag);
```

This example declares the wait for Single Event Flag (SYS$WAITFR) system service, which waits for a single event flag. SYS$WAITFR requires one value parameter, the number of the event flag for which to wait. This number is passed as an immediate value, copied from the integer variable Event_Flag.

7.1.2.3 **By-Descriptor Mechanism** - The by-descriptor mechanism passes the address of a string, array, or scalar descriptor, as described in Appendix C of the VAX-11 Architecture Handbook. VAX-11 PASCAL includes the %STDESCR mechanism specifier for passing string descriptors and the %DESCR mechanism specifier for passing array and scalar descriptors. You cannot pass a component of a packed structure using either of these specifiers. You can use these specifiers only with non-PASCAL subprograms. Note that, by default, PASCAL passes an array descriptor to a formal dynamic array parameter.

To pass a string descriptor, specify %STDESCR as follows:

```
%STDESCR formal-parm-list : type;
```

The formal parameter list specifies one or more parameters of the indicated type. Only string constants, packed character arrays with subscripts from 1 to n, and packed dynamic character arrays with subscripts of an integer or integer subscript type can be passed by string descriptor.

When you call the procedure, the argument list contains the address of each string descriptor. For example, the Broadcast (SYS$BRDCST) system service requires two string descriptors as parameters:

```
TYPE Msgtype = PACKED ARRAY[1..80] OF CHAR;
     Devtype = PACKED ARRAY[1..6] OF CHAR;

VAR Message : Msgtype;
    Terminal : Devtype;

FUNCTION SYS$BRDCST (%STDESCR MSG :  MSGTYPE;
                     %STDESCR DEV : DEVTYPE):
                     INTEGER; EXTERN;

STATUS := SYS$BRDCST (Message, Terminal);
```

The %STDESCR specifier indicates that both parameters must be passed by string descriptor. The actual parameters Message and Terminal are packed arrays of 80 and 6 characters, respectively.

Routines written in other high-level languages may require array or scalar descriptors. To pass an array or scalar descriptor, use %DESCR in the following format:

```
%DESCR formal-parm-list : type;
```

The formal parameter list specifies one or more parameters of the indicated scalar or array type. The type can be any predefined scalar type or an unpacked array (fixed or dynamic) of a predefined scalar type. The argument list contains the address of the descriptor of an array or scalar variable.

The following example shows how an array descriptor might be passed to a FORTRAN subroutine.

```
TYPE Foray = ARRAY [1..10,1..10] OF CHAR;

PROCEDURE Formatrix (%DESCR ARRDES : Foray); FORTRAN;
```

The FORTRAN subroutine Formatrix expects the array to be passed by-descriptor. A call to Formatrix might be the following:

```
Formatrix (Chararr);
```

The actual parameter Chararr specifies a character array, and the argument list contains the address of a descriptor for this array. (Note that VAX-11 FORTRAN treats character parameters as CHARACTER*1 variables.)

7.1.3 Passing Functions and Procedures as Parameters

You can pass procedure and function names as immediate values to routines written in other languages, using the following format:

```
%IMMED PROCEDURE procedure-name-list;

%IMMED FUNCTION function-name-list :  type;
```

The procedure name list specifies the name of one or more formal procedure parameters. The function name list specifies the name of one or more formal function parameters of the indicated type. The corresponding actual parameter lists specify the names of the actual procedures and functions to be passed as parameters.

For example:

```
PROCEDURE Forcaller (%IMMED PROCEDURE Utility);  FORTRAN;
```

The FORTRAN subroutine Forcaller calls a PASCAL procedure and requires that the name of the procedure be passed as an immediate value. The argument list contains the address of the PASCAL procedure's entry mask. A call to the FORTRAN procedure might be:

```
Forcaller (Printer);
```

Any subprogram passed with %IMMED should access only its own variables and those declared at program level.

For information on passing procedure and function names between PASCAL subprograms, see the VAX-11 PASCAL Language Reference Manual.

7.1.4 Function Return Values

A function returns a value to the calling program by assigning that value to the function's name. The value must be a scalar, subrange, or pointer type; structured types are not allowed. The method by which a value is returned depends on its type, as listed below.

Type	Return Method
Integer, real, single, character, Boolean, pointer, user-defined scalar	General Register R0
Double	R0: Low-order result R1: High-order result

7.1.5 Passed Arguments to PASCAL Subprograms

When calling a PASCAL subprogram from a non-PASCAL subprogram, you must ensure that the arguments are in the correct form. By default, VAX-11 PASCAL expects most parameters to be passed by-reference.

For a PASCAL value parameter (declared without a mechanism specifier), the argument list must contain the address of the value. The PASCAL subprogram will copy the value from the passed address upon entry.

For a VAR parameter, the argument list must contain the address of the variable. The subprogram does not copy the value, but instead uses the address to access the actual parameter variable. Actual parameter variables that can change in value as a result of subprogram execution must be passed in this manner. In addition, all files must be passed as PASCAL VAR parameters.

For a formal procedure or function parameter (indicated by the PROCEDURE or FUNCTION specifier), the argument list must specify the address of the bound procedure value, which consists of two longwords. The first longword contains the address of the entry mask for the subprogram; the second longword contains the environment pointer. (This process implements the VAX-11 by-reference mechanism for a procedure or function.)

For a formal dynamic array parameter, the argument list must contain the address of an array descriptor.

7.2 CALLING VAX/VMS SYSTEM SERVICES

You can declare any VAX/VMS system service as an external function or procedure and then call it from your PASCAL program. When declaring a system service, specify an identifier in the following form:

```
SYS$service-name
```

For example, the name of the $FAO system service is SYS$FAO.

You pass parameters to the system service according to its particular requirements: a value, an address, or the address of a descriptor may be needed, as described in Section 7.1.2. To invoke the system service, use a function or procedure call in your PASCAL program. See the VAX/VMS System Services Reference Manual for a full description of each system service.

The system provides three files containing condition symbol definitions. When you declare a system service or Run-Time Library procedure, you should specify the appropriate file in the CONST section to define the condition values in your PASCAL program. Use the %INCLUDE directive to specify the file name, as described in the VAX-11 PASCAL Language Reference Manual.

The three files are SYS$LIBRARY:LIBDEF.PAS, SYS$LIBRARY:MTHDEF.PAS, and SYS$LIBRARY:SIGDEF.PAS, as described below.

SYS$LIBRARY:LIBDEF.PAS

This file contains definitions for all condition symbols from the general utility Run-Time Library procedures. These symbols have the form:

```
LIB$_xyz
```

For example:

```
LIB$_NOTFOU
```

SYS$LIBRARY:MTHDEF.PAS

This file contains definitions for all condition symbols from the mathematical procedures library. These symbols have the form:

```
MTH$_xyz
```

For example:

```
MTH$_SQUROONEG
```

SYS$LIBRARY:SIGDEF.PAS

This file contains miscellaneous symbol definitions used in condition handlers. These symbols have the form:

```
SS$_xyz
```

For example:

```
SS$_FLTOVF
```

7.2.1 Calling System Services by Function Reference

In most cases, you will want to declare a system service as a function so that you can check its return status. Each system service returns a VAX-11 condition value indicating whether completion was successful. These condition values can be interpreted as integer codes that correspond to symbolic names such as SS$_ACCVIO. Odd numbered codes indicate successful completion and even numbered codes indicate failure.

For example, the following procedure defines and calls the Create Mailbox (SYS$CREMBX) system service.

```
PROCEDURE Create_Mailbox;

CONST %INCLUDE 'SYS$LIBRARY:SIGDEF.PAS'

TYPE Status = (Int_Stat, Bool_Stat);
     Word = 0..65535;
     Sub63 = 1..63;
     Chan_Stat = (Int_Chan,Dummy_Chan);
     Chan_Type = PACKED RECORD
         CASE Chan_Stat OF
              Int_Chan : (Chan_No : INTEGER);
              Dummy_Chan : (Bot_Chan : Word)
          END;

VAR Mbx_Rec : RECORD
          CASE Status OF
               Int_Stat : (Mbx_Int : INTEGER);
               Bool_Stat : (Mbx_Bool : BOOLEAN)
          END;
     Chan_Rec : Chan_Type;

FUNCTION SYS$CREMBX (%IMMED PRMFLG : INTEGER;
                     Var Chan : Chan_Type;
                     %IMMED Maxmsg, Bufquo, Promsk, Acmode : INTEGER;
                     %STDESCR Lognam : PACKED ARRAY [Sub63] OF CHAR):
                     INTEGER; EXTERN;

BEGIN
WITH Mbx_Rec DO BEGIN
     Mbx_Int := SYS$CREMBX(0,Chan_Rec,0,0,0,0,'Mailbox');
     IF NOT Mbx_Bool THEN BEGIN
                     WRITELN ('Error when trying to create mailbox');
                     HALT
                     END
     END
END;
```

The function reference allows a return status to be stored in the record Mbx_Rec. If the function's return status is false (represented by any even integer), indicating failure, an error occurs and error processing can be undertaken. You can also check for a particular return status, such as lack of privileges, by comparing the return status to one of the status codes defined by the system. For example:

```
IF Mbx_Rec.Mbx_Int = SS$_NOPRIV THEN
     WRITELN ('No privilege to create mailbox');
```

Refer to the VAX/VMS System Services Reference Manual for information about return status codes. The relevant return status codes are described with each system service.

7.2.2 Calling System Service as Procedures

If you do not need to check the return status, you can declare a system service as an external procedure rather than as an external function. Procedure calls to system services are made in the same way that calls are made to any other procedure. For example, to use the Create Mailbox system service, define and call the procedure SYS$CREMBX, as follows:

```
PROCEDURE SYS$CREMBX (%IMMED PRMFLG : INTEGER;
                      VAR Chan : Chan_Type;
                      %IMMED MAXMSG, Bufquo, Promsk, Acmode : INTEGER;
                      %STDESCR Lognam : ARRAY [Sub63] OF CHAR);
                      EXTERN;

SYS$CREMBX (0,Chan_Rec,0,0,0,0,'Mailbox');
```

You should declare Chan_Rec and Chan_Type as in the previous section.

This procedure call corresponds to the function reference, but does not allow you to test the status code returned by the system service.

7.2.3 Passing Parameters to System Services

Most system services require input parameters to be passed as immediate values. When declaring these parameters, you must use the %IMMED mechanism specifier. Some system services, however, require input parameters to be passed by-reference. For input parameters passed by-reference, you should use the default (that is, omit the mechanism specifier) so that the actual parameters can be expressions.

In addition, most system services require output parameters to be passed by-reference. For these parameters, you must use the VAR mechanism specifier to ensure that PASCAL correctly interprets the output data. The VAX/VMS System Services Reference Manual lists the mechanism by which each parameter to a system service must be passed.

7.2.3.1 **Input and Output By-Reference Parameters** - You may need to tell the system service where to find input values and where to store output values. Thus, you must ascertain the hardware data type of the parameter: byte, word, longword, or quadword.

For input parameters that refer to byte, word, or longword values, you can specify either constants or variables. If you specify a variable, it must be of a type that is allocated an equal or greater amount of storage than is allocated to the hardware data type required.

For output parameters, you must declare a variable of exactly the length required, to avoid including extraneous data. For example, if the system returns a byte value in a word-length variable, the leftmost 8 bits of the variable will not be overwritten on output and the variable will not contain the data you expect. Table 7-1 lists the suggested input and output variable types.

Table 7-1
Suggested Variable Data Types

VAX/VMS Hardware Type Required	Input Parameter Declaration	Output Parameter Declaration
Byte	INTEGER, CHAR	CHAR
Word	INTEGER	Appropriate packed record
Longword	INTEGER	INTEGER
Quadword	Properly dimensioned array	Properly dimensioned array

To store the output produced by a system service, you must allocate sufficient space to contain the output. You can do so by declaring variables of the proper size. For example, the Create Mailbox (SYS$CREMBX) system service returns a 2-byte value. Thus, you can set up storage space as follows:

```
TYPE Word = 0..65535;
     Sub63 = 1..63;
     Chan_Type = PACKED RECORD
                 Bot_Chan : WORD
                 END;
   .
   .
VAR Chan_Rec : Chan_Type;
   .
   .
VALUE Chan_Rec := (0);
FUNCTION SYS$CREMBX (%IMMED PRMFLG : INTEGER;
                     VAR Chan : Chan_Type;
                     %IMMED Maxmsg, Bufquo, Promsk, Acmode : INTEGER;
                     %STDESCR Lognam : PACKED ARRAY [Sub63] OF CHAR):
                     INTEGER; EXTERN;
   .
   .
Mbx_Rec.Mbx_Int := SYS$CREMBX (0,Chan_Rec, 0, 0, 0, 0,'Mailbox');
```

If the output is a quadword value, you must declare an array or record of the proper size. For example, the Get Time (SYS$GETTIM) system service returns the time as a quadword binary value. Thus, you would need to specify the following:

```
TYPE Quad = ARRAY [1..2] OF INTEGER;
  .
  .
VAR Systim : Quad;
    Istat : INTEGER;
     .
     .
FUNCTION SYS$GETTIM (VAR F_Systim : Quad): INTEGER; EXTERN:
     .
     .
ISTAT := SYS$GETTIM (Systim);
```

7.2.3.2 **Optional Parameters** - VAX-11 PASCAL does not allow you to omit parameters from procedure or function calls. If you choose not to supply an optional parameter, you should pass the value zero using the immediate value (%IMMED) mechanism. For example, the Translate Logical Name (SYS$TRNLOG) system service has three optional parameters. If you do not specify values for these parameters, you must include zeros in their places, as follows:

```
Istat := SYS$TRNLOG ('Cygnus', Namlen,Namdes, 0,0,0);
```

7.2.3.3 **Passing Character Parameters** - Some VAX/VMS system services require character parameters for either input or output. For example, the Translate Logical Name (SYS$TRNLOG) system service accepts a logical name as input and returns the associated logical name or file specification, if any, as output.

VAX/VMS system services usually require strings to be passed by string descriptor. Specify %STDESCR in the function or procedure declaration to pass the required string parameters by string descriptor. On input, a character constant or packed array of characters must be passed to the system service by descriptor. On output, two parameters are required: (1) a packed array of characters to hold the output string and (2) an INTEGER variable, which is set to the actual length of the output string. For example:

```
TYPE Word = 0..65535;
     Sub63 = 1..63;
     string_Buf = PACKED ARRAY [1..128] OF CHAR;

VAR  Icode : INTEGER;
     Namlen : WORD;
     Namdes : String_Buf;

   .
   .
   .
PROCEDURE Error;
   .
   .
   .
FUNCTION SYS$TRNLOG (%STDESCR Cygnus : PACKED ARRAY [Sub63] OF CHAR;
                     VAR Rsllen : Word;
                     stdescr rslbuf : String_Buf;
                     %IMMED Table, Acmode, Dsbmsk : INTEGER) : INTEGER; EXTERN;
BEGIN
   .
   .
   .
     Icode := SYS$TRNLOG ('Cygnus', Namlen,Namdes,0,0,4);
     IF NOT ODD (Icode) THEN Error;
```

The logical name Cygnus is translated to its associated name or file specification, and the output values (length and associated name or file specification) are stored in the locations you specified -- Namlen and Namdes, respectively. The last parameter, with the value 4, causes the system to disable its search of the process logical name table; only the system and group tables are searched.

Section 7.4 presents another, complete system service example.

7.3 CALLING RUN-TIME LIBRARY PROCEDURES

The VAX-11 Run-Time Library provides mathematical procedures that you can call from PASCAL programs. These procedures are described in the VAX-11 Run-Time Library Reference Manual.

You can invoke a Run-Time Library procedure from a PASCAL program by defining it as an external function and including the appropriate function reference. For example:

```
VAR Seed_Val : INTEGER;
    Rand_Rslt : REAL;
    .
FUNCTION MTH$RANDOM (Seed : INTEGER) : REAL; EXTERN;
    .
Rand_Rslt := MTH$RANDOM(SEED_VAL);
```

This example uses the uniform pseudorandom number generator (MTH$RANDOM).

When defining a function for a Run-Time Library procedure, you should note the following:

- The mechanism by which each parameter is passed (by-immediate-value, by-reference, or by-descriptor)
- The types appropriate for the parameters and the result

In the pseudorandom number generator, the seed parameter is passed by reference and the result is a real number, as shown.

7.4 COMPLETE SYSTEM SERVICE EXAMPLE

This section presents a sample PASCAL procedure that declares and calls the Get Job/Process Information (SYS$GETJPI) system service. The procedure declares SYS$GETJPI as an INTEGER function, so that upon return, it can check for successful completion.

SYS$GETJPI is used here to get the process identification and name of the current process. When used for this purpose, SYS$GETJPI requires information in only the fourth of seven parameters. The first, second, third, fifth, sixth, and seventh parameters must be null. The fourth parameter contains the address of a list of descriptors that describe the specific information requested and point to buffers to receive the information. In this example, the list is constructed as a record. It contains fields corresponding to each required item as noted in the description of SYS$GETJPI in the VAX/VMS System Services Reference Manual.

```
PROCEDURE Getjpinfo  (VAR Name: Prcnam; VAR ID: INTEGER);

        (*This procedure calls the SYS$GETJPI system service to
        get the process ID and process name, which are
        used as formal parameters.  It uses these types
        and variables:

          Word -- 16 bits to contain buffer length and request code
          Ptr_Pid -- Address of process ID
          Ptr_Pidlen -- Address of process ID length
          Ptr_Prcnam -- Address of process name string
          Ptr_Prcnamlen -- Address of length of process name string
          Jpirec -- Record containing item list parameter
          Icode -- Status returned by SYS$GETJPI function
```

```
The following type is declared in the main program:
   PRCNAM = PACKED ARRAY [1..15] OF CHAR*)

CONST Null = 0;

TYPE Word = 0..65535;
     Recj = RECORD
            Pidinfo : PACKED RECORD
                         Pidlen, Jpi$_Pid : Word
                         END;
            Ptr_Pid : ^INTEGER;
            Ptr_Pidlen : ^INTEGER;
            Prcnaminfo : PACKED RECORD
                          Prcnamlen, Jpi$_Prcnam : WORD
                          END;
            Ptr_Prcnam : ^Prcnam;
            Ptr_Prcnamlen : ^INTEGER;
            Endlist : INTEGER
     END;

 VAR Icode : INTEGER;
     Jpirec : Recj;

 FUNCTION SYS$GETJPI (%IMMED A,B,C : INTEGER; VAR Itmlst : Recj;
                      %IMMED X,Y,Z : INTEGER) : INTEGER; EXTERN;
        (*A,B,C and X,Y,Z are null parameters*)

 BEGIN

    (*Set up record parameter*)

  WITH Jpirec DO
       BEGIN
          Pidinfo.Pidlen := 4;          (*length of ID buffer*)
          Pidinfo.Jpi$_Pid := %x319;    (*Hex of JPI$_PID*)
          NEW (Ptr_Pid);                (*Get address of ID*)
          Ptr_Pid^ := 0;                (*Zero ID variable*)
          NEW (Ptr_Pidlen);             (*Get address of length*)
          Ptr_Pidlen^ := 0;             (*Zero ID length variable*)
          Prcnaminfo.Prcnamlen := 15;   (*Length of name buffer*)
          Prcnaminfo.Jpi$_Prcnam := %x31C;
                                        (*Hex of JPI$_PRCNAM*)
          NEW (Ptr_Prcnam);             (*Get address of name*)
          Ptr_Prcnam^ := '                ';
                                        (*Blank-fill name string*)
          NEW (Ptr_Prcnamlen);          (*Get address of length*)
          Ptr_Prcnamlen^ := 0;          (*Zero name length variable*)
          ENDLIST := 0                  (*List must end with 0*)
  END;

    (*Call function and return status in STATUS variable*)

       Icode := SYS$GETJPI (Null, Null, Null, Jpirec, Null, Null, Null);
       IF NOT ODD(Icode) THEN                    (*If error, *)
            BEGIN
               WRITELN ('Error in GETJPI process'); (*print error message*)
               HALT                             (*and halt*)
            END
          ELSE
            BEGIN                           (*If successful,*)
            NAME := Jpirec.Ptr_Prcnam^;     (*assign name to Name param*)
            ID := Jpirec.Ptr_Pid^           (*and assign id to ID param*)
            END
 END;
```

CHAPTER 8

ERROR PROCESSING AND CONDITION HANDLERS

During the execution of a VAX-11 PASCAL program, various conditions, including errors or exceptions can occur. These conditions result from errors during I/O operations, invalid input data, incorrect calls to library routines, errors in arithmetic, or system-detected errors. VAX-11 PASCAL provides two methods of error control and recovery:

- Run-Time Library default error-processing procedures
- VAX-11 Condition Handling Facility (including user-written condition handlers)

These error-processing methods are complementary and can be used in the same program.

The Run-Time Library can provide all condition handling needed by an application program. The Run-Time Library provides default error processing by generating error messages for all error or exception conditions that occur during the execution of a PASCAL program. Section 8.1 describes error processing by the Run-Time Library as applied to VAX-11 PASCAL. Appendix A describes the error messages that can be generated for a PASCAL program.

The VAX-11 Condition Handling Facility provides, at the lowest level, all condition handling for the Run-Time Library and can provide routines (user-written condition handlers) for processing conditions that occur during your program's execution. The use of condition handlers requires considerable programming experience and should not be undertaken by novice users. You should understand the condition handling descriptions in the VAX/VMS System Services Reference Manual, the VAX-11 Run-Time Library Reference Manual, or the VAX-11 Architecture Handbook before attempting to write a condition handler.

The following terms are used in this chapter:

- **Condition handler** -- A function that has been specified by a particular routine as the handler to be called when an exception condition is signaled.
- **Condition value** -- An INTEGER value that identifies a particular condition.
- **Establish** -- The process of placing the address of a condition handler in the stack frame of the current routine activation. A condition handler established for a routine activation is automatically called when a condition occurs. In PASCAL, condition handlers are established by means of the LIB$ESTABLISH routine.

- **Routine activation** -- The environment in which a routine executes. This environment includes a unique stack frame on the run-time stack; the stack frame contains the address of a condition handler for the routine activation. A new current routine activation is created every time a routine is called; the routine activation is deleted when the routine returns.

- **Program Exit Status** -- The program completion status at program completion.

- **Resignal** -- The means by which a condition handler indicates that the signaling facility is to continue searching for a condition handler to process a previously signaled error. To resignal, a condition handler returns the SS$_RESIGNAL value.

- **Signal** -- The means by which the occurrence of an exception condition is made known. Signals are generated by the operating system in response to I/O events and hardware errors, by the system supplied library routines and by user routines. All signals are initiated by calling the signaling facility. There are two entry points to the signaling facility.

  - LIB$SIGNAL -- Used to signal a condition and, possibly, to continue program execution

  - LIB$STOP -- Used to signal a severe error and discontinue program execution, unless a condition handler performs an unwind operation

- **Stack frame** -- A standard data structure built on the stack during a procedure call, starting from the location addressed by the FP to lower addresses and popped off during a return from procedure.

- **Unwind** -- To return control to a particular routine activation, bypassing any intermediate procedure activations. For example, if X calls Y and Y calls Z and Z detects an error, a condition handler associated with X or Y can unwind to X, bypassing Y. Control returns to X immediately following the point at which X called Y.

8.1 RUN-TIME LIBRARY DEFAULT ERROR PROCESSING

The Run-Time Library, by default, prints a message and terminates PASCAL program execution when a run-time error occurs. These default actions occur unless your program includes a condition handler.

Run-time errors are reported by default in the following format:

```
%PAS-F-code, text
```

The code is an abbreviation of the error message text. VAX-11 PASCAL run-time errors and recovery procedures are described in Appendix A. Most Run-Time Library routines provide their own error messages, as described in the VAX-11 Run-Time Library Reference Manual.

8.2 OVERVIEW OF VAX-11 CONDITION HANDLING

When the VAX/VMS system creates a user process, a system-defined condition handler is established in the absence of any user-written condition handler. The system-defined handler processes errors that occur during execution of the user image when no user-defined handlers are present. Thus, by default, a run-time error causes the system-defined condition handler to print one of the standard error messages and to terminate or continue execution of the image, depending on the severity code associated with the error.

When a condition is signaled, the system searches for condition handlers to process the condition. The system conducts the search for condition handlers by proceding down the stack, frame by frame, (see Section 8.2.1 for definition) until a condition handler is found that does not resignal. The default handler calls the system's message output routine to send the appropriate message to the user. Messages are sent to the SYS$OUTPUT file and to the SYS$ERROR file, if both files are present. If the condition is not a severe error, program execution continues. If the condition is a severe error the default handler forces program termination and the condition value becomes the program exit status.

You can create and establish your own condition handlers according to the needs of your applications. For example, your condition handler could create and display messages that specifically describe conditions encountered during execution of an application program, instead of relying on the standard system error messages.

8.2.1 Condition Signals

A condition signal consists of a call to one of the two signaling facility entry points, LIB$SIGNAL and LIB$STOP. These entry points must be declared external, for example:

```
PROCEDURE LIB$SIGNAL (%IMMED Condition : INTEGER); EXTERN;

PROCEDURE LIB$STOP (%IMMED Condition : INTEGER); EXTERN;
```

If a condition occurs in a routine that cannot handle it, notification is passed to other active routines by issuing a signal. If the current routine can continue after the signal is propagated, LIB$SIGNAL is called. A higher-level routine can then determine whether program execution is to continue. If the nature of the condition does not allow the current routine to continue, LIB$STOP is called.

Condition values are usually expressed as condition symbols, for example:

```
LIB$SIGNAL (MTH$_FLOOVEMAT);
```

Additional parameters can be included to provide supplementary information about the error.

When called, the signaling facility searches for condition handlers by examining the preceding stack frames until it activates a condition handler that does not resignal.

8.2.2 Handler Responses

A condition handler responds to an exception condition by taking action in three major areas:

- Condition correction
- Condition reporting
- Condition control

First, the handler determines whether the condition is correctable. If possible, the handler takes the appropriate action, and execution continues. If the handler cannot correct the condition, the condition may be resignaled. That is, the handler requests that another condition handler be located to process the condition.

Condition reporting performed by handlers can involve one or more of the following actions:

- Maintaining a count of exceptions encountered during program execution
- Resignaling the same condition to send the appropriate message to your terminal or log file
- Changing the severity field of the condition value and resignaling the condition
- Signaling a different condition, for example, to produce a message oriented to a specific application

Execution can be affected in a number of ways:

- Continuing from the signal. If the signal was issued through a call to LIB$STOP, the program exits.
- Unwinding to the establisher at the point of the call that resulted in the exception. The handler can determine the function value returned by the called routine.
- Unwinding to the establisher's caller (the routine that called the routine that established the handler). The handler can determine the function value returned by the called routine.

8.3 WRITING CONDITION HANDLERS

The following sections describe how to code and establish condition handlers and provide some simple examples. See the VAX-11 Architecture Handbook, the VAX-11 Run-Time Library Reference Manual, and the VAX/VMS System Services Reference Manual for more details on condition handlers.

8.3.1 Establishing and Removing Handlers

When a routine is called, a condition handler is not initially established. To use a condition handler, first define both the Run-Time Library procedure LIB$ESTABLISH and the condition handler as follows:

```
FUNCTION HANDLER : INTEGER;  EXTERN;
PROCEDURE LIB$ESTABLISH (%IMMED FUNCTION Handler : INTEGER);
   .       EXTERN;
   .
   .
LIB$ESTABLISH (ERR_HANDLER)
```

To establish the handler, call LIB$ESTABLISH. To remove an established handler, define the Run-Time Library procedure LIB$REVERT, and call as follows:

```
PROCEDURE LIB$REVERT; EXTERN;
   .
   .
   .
LIB$REVERT
```

As a result of this call, the condition handler established in the current stack frame is removed. When a routine returns, any condition handler established during that activation is automatically removed.

Note that condition handlers written in PASCAL can access only their own local data and data declared at program or module level.

8.3.2 Parameters for Condition Handlers

A PASCAL condition handler is an INTEGER function that is called when a condition is signaled. Two formal VAR parameters must be defined for a condition handler:

1. An integer array to refer to the parameter list from the call to the signal routine (the signal parameters); that is, the list of parameters included in calls to LIB$SIGNAL or LIB$STOP (see Section 8.2.1).

2. An integer array to refer to information concerning the routine activation that established the condition handler (the mechanism array).

For example, a condition handler may be defined as follows:

```
TYPE Mecharr = ARRAY[0..4] OF INTEGER;
     Sigarr = ARRAY [0..9] OF INTEGER;
   .
   .
   .
FUNCTION HANDLER (VAR Sigargs : Sigarr;
                  VAR Mechargs : Mecharr) : INTEGER;
     BEGIN
     .
     .
     .
     END;
```

```
PROCEDURE LIB$ESTABLISH (%IMMED FUNCTION Handler : INTEGER); EXTERN;
   .
   .
   .
BEGIN
   .
   .
   .
LIB$ESTABLISH (HANDLER);
   .
   .
   .
END.
```

The array Sigargs receives the values listed below from the signal procedure.

Value	Meaning
Sigargs[0]	Indicates how many parameters are being passed in this array (parameter count).
Sigargs[1]	Indicates the condition being signaled (condition value). See Section 8.3.4 for a discussion of condition values.
Sigargs[2 to n]	Indicates optional parameters supplied in the call to LIB$SIGNAL or LIB$STOP; note that the dimension bounds for the Sigargs array should specify as many entries as necessary to refer to the optional parameters.

The array Mechargs receives information about the procedure activation status of the routine that established the condition handler. The values from this routine are listed below.

Value	Meaning
Mechargs[0]	Specifies the number of parameters in this array (4).
Mechargs[1]	Contains the address of the stack frame that established the handler.
Mechargs[2]	Contains the number of calls that have been made (that is, the stack frame depth) from the routine activation, up to the point at which the condition was signaled.
Mechargs[3]	Contains the value of register R0 at the time of the signal.
Mechargs[4]	Contains the value of register R1 at the time of the signal.

8.3.3 Handler Function Return Values

Condition handlers specify function return values to control subsequent execution. The function return values and their effects are listed below.

Value	Effect
SS$_CONTINUE	Continues execution from the signal. If the signal was issued by a call to LIB$STOP, the program exits.
SS$_RESIGNAL	Resignals to continue the search for a condition handler to process the condition.

In addition, a condition handler can request a stack unwind by calling SYS$UNWIND before returning. Declare SYS$UNWIND as follows:

```
FUNCTION SYS$UNWIND (Depth : INTEGER; %IMMED Newpc : INTEGER):
          INTEGER; EXTERN;
```

When SYS$UNWIND is called, the function return value is ignored. The handler modifies the saved registers R0 and R1 in the mechanism parameters to specify the called procedure's function value.

A stack unwind can be made to one of two places:

- Unwind to the establisher, at the point of the call that resulted in the exception. Specify:

```
Status := SYS$UNWIND (Mechargs[2],0);
```

- Unwind to the routine that called the establisher. Specify:

```
Status := SYS$UNWIND (Mechargs[2]+1,0);
```

8.3.4 Condition Values and Symbols

VAX-11 uses condition values to indicate that a called routine has either executed successfully or failed, and to report exception conditions. Condition values are 32-bit packed records (usually interpreted as integers), consisting of fields that indicate which system component generated the value, the reason the value was generated, and the severity of the condition. The definition of a condition value has the form:

```
TYPE Condition_Value=PACKED RECORD

     (* Field          Bits          Meaning*)

     Severity:0..7;    (*2:0         Specifies a severity code as
                                     follows:
                                          0 - warning
                                          1 - success
                                          2 - error
                                          3 - information
                                          4 - severe error
                                       5,6,7 - reserved *)

     Message:0..8191;  (*15:3        Describes the condition that
                                     occurred.  Bit 15 = 1 indicates
                                     that the message is specific to a
                                     single facility.  Bit 15 = 0
                                     indicates a system-wide
                                     message.*)

     Facility:0..4095; (*27:16       Identifies the software component
                                     that generated the condition
                                     value.  Bit 27 = 1 indicates a
                                     customer facility.  Bit 27 = 0
                                     indicates a DIGITAL facility.*)

     Control:0..15     (*31:28       Control bits.*)

     END;
```

A warning severity code (0) indicates that output was produced, but the results may be unpredictable. An error severity code (2) indicates that output was produced even though an error was detected. Execution can continue, but the results may not be correct. A severe error code (4) indicates that the error was of such severity that output was not produced. An even-numbered error code is a warning. A condition handler can alter the severity code of a condition value to allow execution to continue or to force an exit, depending on the circumstances.

In the example in Section 8.3.2 the condition value is passed as the second element of the array Sigargs. Occasionally, a condition handler may require a particular condition be identified by an exact match. That is, each bit of the condition value (31:0) must match the specified condition. For example, you may want to process a floating overflow condition only if the severity code is still 4 (that is, if no previous condition handler has changed the severity code). As noted above, a typical condition handler response is to change the severity code and resignal.

In many cases, however, response to a condition, regardless of the value of the severity code is desired. To ignore the severity and control fields of a condition value, declare and call the LIB$MATCH_COND function, as follows:

```
FUNCTION LIB$MATCH_COND (Condval,Compval : INTEGER):
        BOOLEAN;EXTERN;
.
.
.
IF LIB$MATCH_COND(Sigargs[1],PAS$_Erraccfil)
        THEN...
```

8.3.5 Floating-Point Operation

Some conditions involving floating-point operations require special action to continue execution. Operations that involve, for example, floating overflow, dividing by 0, or computing the square root of a negative number, storing a unique result known as a floating reserved operand. If a subsequent floating-point operation accesses this result, a hardware reserved operand fault is generated and signaled. This can continue indefinitely if the condition handler does not change the reserved operand, because the operand is accessed each time the computation is retried.

To allow computation to continue, change the reserved operand by defining and calling the Run-Time Library routine LIB$FIXUP_FLT, as follows:

```
FUNCTION LIB$FIXUP_FLT (VAR Sigadr : Sigvector;
        VAR Mechadr : Mechvector; VAR Op : DOUBLE):
        INTEGER; EXTERN;
.
.
.

Status:= LIB$FIXUP_FLT(Sigargs,Mechargs,Newoperand);
```

The types Sigvector and Mechvector in this example refer to the types of the signal and mechanism argument vectors, which are assumed to be defined in the calling procedure. The third parameter is specified as a double-precision variable to ensure that the reserved operand is changed correctly regardless of precision. Specify 0.0D0 if there is no special value for this parameter. For more information on LIB$FIXUP_FLT, see the VAX-11 Run-Time Library Reference Manual.

8.4 CONDITION HANDLER EXAMPLE

The following example illustrates how to declare and use a condition handler with a typical PASCAL procedure. A condition handler is established, that is called when an error occurs in a file opening procedure. If the error is the general "Error opening/creating the file," signified by the PAS$_ERROPECRE code, an unwind operation is performed. Any other error is resignaled.

```
PROCEDURE Openhand;

(*This procedure shows how to establish and call a  condition  handler
from a PASCAL program.  It uses these types:

        Datarec -- a record type for the accounting file
        Datafile -- a file type with components of type Datarec
        Sigarr -- an array type for signal parameters
        Mecharr -- an array type for mechanism parameters

It declares these global variables:
        Flag -- a Boolean variable set to TRUE when an error occurs
        New_Accounts -- a file variable of type Datafile
        Status -- an all-purpose function return status variable *)

        CONST %INCLUDE 'SYS$LIBRARY:SIGDEF.PAS'

        (*This file contains the PASCAL declarations of the condition
        signals*)

        TYPE Datarec = RECORD
                       Name : PACKED ARRAY [1..30] OF CHAR;
                       Amount : REAL;
                       Cost : REAL;
                       Date : PACKED ARRAY [1..11] OF CHAR
                       END;

             Datafile = FILE OF Datarec;
             Sigarr = ARRAY [0..9] OF INTEGER;
             Mecharr = ARRAY [0..4] OF INTEGER;

        VAR Flag : BOOLEAN;
            New_Accounts : DATAFILE;
            Status : INTEGER;

        PROCEDURE LIB$ESTABLISH (%IMMED FUNCTION Fixer : INTEGER); EXTERN;

        (*The Run-Time Library procedure LIB$ESTABLISH will be called to
        establish the condition handler*)

        PROCEDURE Opener (VAR Accounts : Datafile);

        (*This procedure will be called to open the file and write new
        records in it.  It also performs data entry and cleanup.  Only
        the OPEN processing is shown here for simplicity*)

                (*Local variable declarations*)

                FUNCTION HANDLER (VAR Sigargs : Sigarr; VAR Mechargs : Mecharr):
                        INTEGER;

                (*This function will be called to handle a file opening error during
                the OPENER procedure.  It uses only globally declared variables,
                except for the SYS$UNWIND and LIB$MATCH_COND functions.*)


                        FUNCTION SYS$UNWIND (Depth : INTEGER; %IMMED Newpc : INTEGER):
                                INTEGER; EXTERN;

                        FUNCTION LIB$MATCH_COND (Condval, Compval : INTEGER):
                                BOOLEAN; EXTERN;

                BEGIN
                        IF (LIB$MATCH_COND (Sigargs[1], Pas$_Erropecre)) THEN  (*If error opening file,*)
                                BEGIN
                                Flag := True;                                  (*set file error flag*)
                                Status := SYS$UNWIND(Mechargs[2]+1, 0)         (*and unwind*)
                                END;
                        HANDLER := SS$_RESIGNAL                                (*If some other error, resignal*)
                END;    (*end HANDLER*)

        BEGIN
                LIB$ESTABLISH (HANDLER); (*establish condition handler*)
                OPEN (Accounts, '[DATA]Accounts.DAT',OLD, SEQUENTIAL);

                (*Data entry, storage, and cleanup*)


        END;  (*End OPENER*)

BEGIN
(*This is the start of the outermost procedure*)

        Flag :=FALSE;                   (*initialize Flag to FALSE for test below*)
        OPENER (New_Accounts);          (*open the file*)
        IF Flag THEN WRITELN ('Error in opening file')
                                        (*Print message if error handler was called*)
END; (*End Openhand*)
```

CHAPTER 9

VAX-11 PASCAL SYSTEM ENVIRONMENT

This chapter describes the relationship between the VAX/VMS operating system and the VAX-11 PASCAL compiler. It covers the following topics:

- Use of program sections
- Storage of scalar and pointer types
- Storage of unpacked structured types
- Storage of packed structured types
- Representation of floating-point data

9.1 USE OF PROGRAM SECTIONS

The VAX-11 PASCAL compiler uses contiguous areas of memory, called program sections, to store information about the program. The VAX-11 Linker controls memory allocation and sharing according to the attributes of each program section. Table 9-1 lists the possible program section attributes.

Table 9-1
Program Section Attributes

Attribute	Meaning
PIC/NOPIC	Position independent or position dependent
CON/OVR	Concatenated or overlaid
REL/ABS	Relocatable or absolute
GBL/LCL	Global or local scope
SHR/NOSHR	Shareable or nonshareable
EXE/NOEXE	Executable or nonexecutable
RD/NORD	Readable or nonreadable
WRT/NOWRT	Writeable or nonwriteable

VAX-11 PASCAL implicitly declares three program sections: $GLBL, $CODE, and $PDATA. Table 9-2 summarizes the usage and attributes of these program sections.

Table 9-2
Program Section Usage and Attributes

Program Section Name	Usage	Attributes
$GLBL	Read/write static data declared at module or program level	PIC, OVR, REL, GBL, NOSHR, NOEXE, RD, WRT
$CODE	Read-only generated executable code	PIC, CON, REL, LCL, SHR, EXE, RD, NOWRT
$PDATA	Read-only constants that need storage	PIC, CON, REL, LCL, SHR, NOEXE, RD, NOWRT

Each module in your PASCAL program is named according to the identifier specified in the program or module header. You can use the module name to qualify the program section name in LINK commands. For more information, refer to the VAX-11 Linker Reference Manual.

When the linker constructs an executable image, it divides the executable image into sections. Each image section contains program sections that have the same attributes. The linker controls memory allocation by arranging image sections according to program section attributes.

The linker allows you to use special options to change program section attributes and to influence the memory allocation in the image. You include these options in an options file, which is input to the linker. The options and the file are described in the VAX-11 Linker Reference Manual.

9.2 STORAGE OF SCALAR AND POINTER TYPES

When not part of a packed structure, the scalar types in PASCAL are allocated storage space as summarized in Table 9-3.

Variables of scalar types, with the exception of DOUBLE variables, are aligned on a boundary corresponding to their sizes. DOUBLE variables are aligned on longword boundaries rather than on quadword boundaries.

Variables of subrange types are allocated and aligned in the same way as variables of the associated scalar types. For example, an integer subrange variable is allocated 1 longword and is aligned on a longword boundary. A subrange of an enumerated type Days_Of_Week (with values Sunday, Monday, Tuesday, and so on) is stored in 1 byte and aligned on a byte boundary.

A pointer is simply a longword containing an address.

Table 9-3
Storage of Scalar and Pointer Types

Type	Storage Allocation	Alignment Boundary
Character	8 bits (1 byte)	Byte
Boolean	8 bits (1 byte)	Byte
Integer, single, real	32 bits (1 longword)	Longword
Double	64 bits (1 quadword)	Longword
Enumerated	8 bits (1 byte) if type contains 256 elements or less; 16 bits (1 word) if type contains more than 256 elements	Byte if type contains 256 elements or less; word if type contains more than 256 elements
Pointer	32 bits (1 longword)	Longword

9.3 STORAGE OF UNPACKED STRUCTURED TYPES

The unpacked structured types (sets, arrays, and records) are stored and aligned as described below. Note that this description applies only to data items that are not part of another structure.

A set consists of 32 bytes (8 longwords) aligned on a longword boundary.

An array is stored and aligned according to the type of its elements. For example, each element of an array of integers is stored in 1 longword and aligned on a longword boundary. Similarly, each element of a character array is stored in 1 byte and aligned on a byte boundary.

Records are stored field by field according to the type of each field. The type of the first field in the record establishes the alignment of the entire record. Subsequent fields in the record are always aligned on byte boundaries, regardless of the type of the first field. For example:

```
VAR A :  RECORD
         X :  INTEGER ;
         Y :  BOOLEAN ;
         Z :  INTEGER
         END;
```

Record A is aligned on a longword boundary because its first field, X, contains an integer value, which is stored in a longword. Figure 9-1 shows how this record is stored.

Bytes 0 through 3 (bits 0 through 31) contain the first field, X, which is an integer longword value. Byte 4 (bits 32 through 39) contains the Boolean value of Y, and bytes 5 through 8 (bits 40 through 71) contain the other integer longword value, Z.

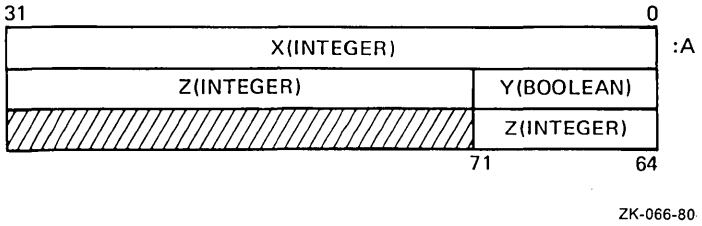


Figure 9-1 Storage of Sample Record

9.4 STORAGE OF PACKED STRUCTURED TYPES

Although you can pack any structured type, packing saves storage space only for sets, arrays, and records. Packing files has no effect on their storage.

In general, storage space for packed types is allocated according to the "32-bit rule," as follows:

- Any data item that is 32 bits or less in length is packed into as few bits as possible.
- Any data item over 32 bits long is allocated the smallest possible number of bytes and is aligned on a byte boundary.

The sections below describe exactly how storage is allocated to each packed type and note any exceptions to the general 32-bit rule. The descriptions apply only to data items that are not part of another structure.

9.4.1 Storage of Packed Sets

A packed set that is not a component of another packed structure is byte-aligned. Each packed set is allocated space according to the 32-bit rule, based on the ordinal value of its largest element:

- If the ordinal value is less than or equal to 31, the set is allocated the number of bits equal to the ordinal value plus 1. For example, a packed set of 2..19 is allocated 20 bits.
- If the ordinal value is greater than 31, the set is allocated the number of bits equal to the ordinal value plus 1, rounded up to the nearest byte boundary. For example, a packed set of subrange type 100..101 is allocated 13 bytes (101+1 bits rounded up to a byte boundary).

Since the size of a set is limited to 256 elements, with ordinal values from 0 to 255, a packed set can occupy at most 32 bytes (8 longwords) of memory.

9.4.2 Storage of Packed Arrays

A packed array that is not a component of another packed structure is aligned on a byte boundary. The elements of the array are packed to the nearest bit. Table 9-4 lists the space requirements for elements of packed arrays.

Table 9-4
Storage of Packed Array Elements

Type	Storage Allocation
Boolean	1 bit
Character	8 bits (1 byte)
Integer, real, single	32 bits (1 longword)
Double	64 bits (1 quadword)
Subrange of integer, character, or enumerated type	Minimum number of bits in which the largest and smallest possible values can be expressed
Enumerated types	Number of bits required for largest ordinal value
Pointers	32 bits (1 longword)
All structured types	Same as structured types not in packed array. However, if the total size of the structured type is greater than 32 bits, the array element is allocated a minimum number of bytes, that is, the 32-bit rule applies. Structured types requiring 32 bits or less space are bit-aligned.

Note that integer subranges are packed into the minimum amount of space needed to hold the largest or smallest value, whichever needs more space. For example, each element of PACKED ARRAY [1..10] OF -128..127 is allocated 8 bits. Each element of PACKED ARRAY [1..64] OF 0..7 is allocated 3 bits.

Enumerated types are packed into the number of bits required to hold the largest ordinal value. For example, an enumerated type with 16 values is allocated 4 bits, because its ordinal values are 0 through 15.

A packed array of an unpacked structured type saves no storage space. The only effect of such a specification is to byte-align the array. Instead, specify a packed array of a packed structured type. The following two examples illustrate this difference.

Examples

1.
```
TYPE Int_Set = SET OF 0..14;
VAR Int_Arr :  PACKED ARRAY [1..5] OF Int_Set;
```

An unpacked set of type Int_Set is stored as 8 longwords. Consequently, each element of Int_Arr requires 8 longwords, for a total of 40 longwords (640 bits) of space.

2. TYPE Int_Set = PACKED SET OF 0..14;
   VAR Int_Arr : PACKED ARRAY [1..5] OF Int_Set;

   A packed set of type Int_Set is allocated 15 bits. Each element of Int_Arr therefore requires 15 bits, for a total of 75 bits for the entire array.

Storage for packed arrays of records and arrays of packed records is allocated similarly.

Multidimensional arrays can also be packed. As for the other structured types, you must specify packing at the innermost level to gain any significant space advantage. For a 2-dimensional array, an array of a packed array generally takes less space than a packed array of an array, as in the following examples.

Examples

1. TYPE Internal_Arr = Array [1..5] of 0..6;
   VAR Sampl_Arr : PACKED ARRAY [1..5] OF Internal_Arr;

   Each element of an array of type Internal_Arr is stored in a longword. Each element of Sampl_Arr, in turn, requires 5 longwords -- enough storage space for 5 elements of type Internal_Arr. The entire array Sampl_Arr therefore occupies 25 longwords (800 bits).

2. VAR Sampl_Arr : PACKED ARRAY [1..5,1..5] OF 0..6;
   VAR Sampl_Arr : ARRAY [1..5] OF PACKED ARRAY [1..5] OF 0..6;

   Specifying PACKED for an array with multiple subscripts results in packing only at the innermost level. Therefore, the two array declarations in this example are equivalent. Each PACKED ARRAY[1..5] of 0..6 requires 15 bits. Because the packed arrays are elements of an unpacked array, their size is rounded up to an even 16 bits. The total size of each SAMPl_ARR is therefore 80 bits. Example 3 shows a slightly more efficient way of allocating this array.

3. TYPE Internal_Arr = PACKED ARRAY [1..5] OF 0..6;
   VAR Samp2_Arr : PACKED ARRAY [1..5] OF Internal_Arr;

   In this example, each element of Internal_Arr requires only 3 bits because the array is packed. Each element of Samp2_Arr can be stored in 15 bits, and the entire array occupies 75 bits.

4. TYPE Internal_Arr = PACKED ARRAY [1..5] OF 0..6;
        Samp3_Arr = PACKED ARRAY [1..5] OF Internal_Arr;
   VAR Sample : PACKED ARRAY [1..5] OF Samp3_Arr;

   This example shows how you can maximize space savings for arrays of more than two dimensions by specifying PACKED at every level. As in Example 3, each element of Internal_Arr requires 3 bits, and each element of Samp3_Arr requires 15 bits. The entire array Sample, then, requires 375 bits.

9.4.3 Storage of Packed Records

A packed record that is not a component of another packed structure is aligned on a byte boundary. The fields within the record are allocated space depending on their sizes and types.

Fields of scalar types, if less than or equal to 32 bits long, are packed to the nearest bit. A field that requires more than 32 bits is aligned on a byte boundary and is allocated space as for a field of an unpacked record.

Except for its alignment, a field that contains an unpacked array, set, or record occupies the same amount of space in a packed or unpacked record. To pack such a field, you must explicitly declare the type of the field to be packed. For example:

```
VAR Sample1 : PACKED RECORD
                A :  -128 ..127;
                B :  BOOLEAN;
                C :  ARRAY [1..5] OF 0..30
                END;
```

This record is byte-aligned and is allocated storage as follows:

Field	Storage Allocation
A	8 bits
B	1 bit
C	160 bits

Field A, an integer subrange, is stored in the smallest possible amount of space, 7 data bits plus a sign bit. Field B, a Boolean, takes up 1 bit, leaving 7 bits unused. Field C is an unpacked array, which is allocated storage as for integers, 32 bits (1 longword) for each of 5 elements.

Figure 9-2 shows how this record is stored.

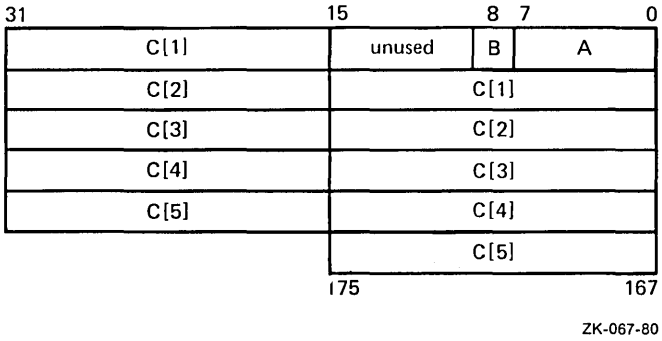


Figure 9-2 Storage of Sample Record

This record requires a total of 176 bits (11 words) of storage.

Compare the preceding example with the next one, which specifies a packed array.

```
VAR Sample2 : PACKED RECORD
                A :  -128..127;
                B :  BOOLEAN;
                C :  PACKED ARRAY [1..5] OF 0..30
                END;
```

This record is byte-aligned and is allocated storage as follows:

Field	Storage Allocation
A	8 bits
B	1 bit
C	25 bits

Fields A and B are allocated the same amount of space in both sample records. Field C, however, requires much less space in SAMPLE2 because it is packed. Each element of the packed array C occupies only 5 bits. Figure 9-3 shows how this record is stored.

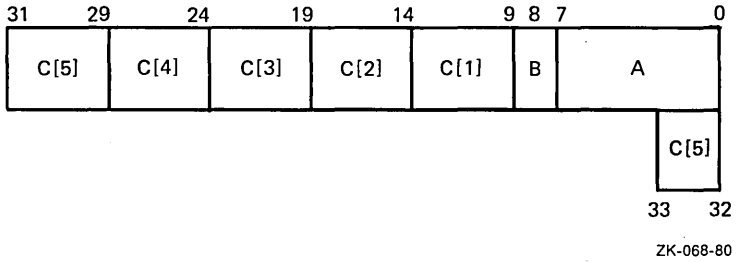


Figure 9-3 Storage of Sample Packed Record Containing Packed Array

This record requires a total of 34 bits of storage.

9.5 REPRESENTATION OF FLOATING-POINT DATA

The following sections summarize the internal representation of single-precision (REAL and SINGLE types) and double-precision (DOUBLE type) floating-point numbers. For more detailed information, see the VAX-11 Architecture Handbook.

9.5.1 Single-Precision Floating-Point Data (SINGLE, REAL Types)

A single-precision floating-point value is represented by four contiguous bytes. The bits are numbered from the right 0 through 31, as shown in Figure 9-4.

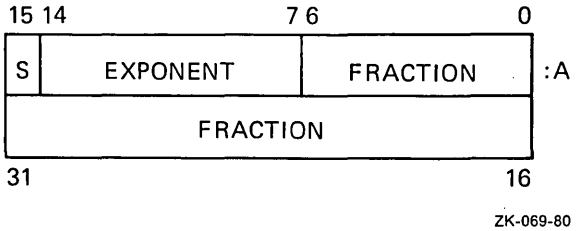


Figure 9-4 Single-Precision Floating-Point Data Representation

A single-precision floating-point value is specified by its address A, the address of the byte containing bit 0. The form of the value is sign magnitude with bit 15 the sign bit, bits 14 through 7 an excess 128 binary exponent, and bits 6 through 0 and 31 through 16 a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from 16 through 31 and 0 through 6.

The 8-bit exponent field encodes the values 0 through 255. An exponent value of 0, with a sign bit of 0, indicates that the floating-point value has a value of 0. Exponent values of 1 through 255 indicate binary exponents of -127 through +127. An exponent value of 0, with a sign bit of 1, is taken as a reserved operand. Floating-point instructions processing a reserved operand take a reserved operand fault.

The value of a floating-point number is in the approximate range of .29*(10**-38) through 1.7*(10**38). The precision of a single-precision value is approximately one part in 2**23, or 7 decimal digits.

9.5.2 Double-Precision Floating-Point Data (DOUBLE Type)

A double-precision floating-point value is represented by 8 contiguous bytes. The bits are numbered from the right 0 through 63, as shown in Figure 9-5.

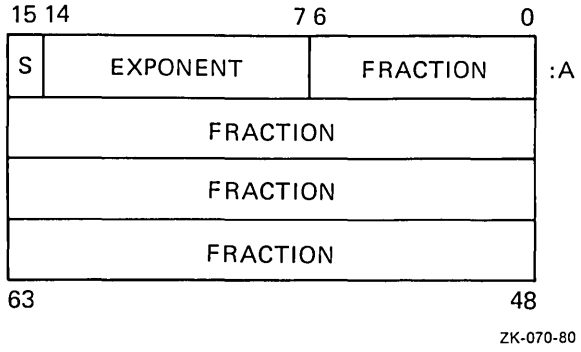


Figure 9-5 Double-Precision Floating-Point Data Representation

A double-precision floating-point value is specified by its address A, the address of the byte containing bit 0. The form of a double-precision floating-point value is identical to a single-precision floating-point value except for an additional 32 low-significance fraction bits. Within the fraction, bits of increasing significance are numbered 48 through 63, 32 through 47, 16 through 31, and 0 through 6.

The exponent conventions and approximate range of values are the same for double-precision floating-point values as for single-precision floating-point values. The precision of a double-precision floating-point value is approximately one part in 2**55, or 16 decimal digits.

APPENDIX A

DIAGNOSTIC MESSAGES

This appendix summarizes the error messages that can be generated by a PASCAL program at compile time and at run time.

A.1 COMPILER DIAGNOSTICS

VAX-11 PASCAL reports compile-time diagnostics in the source listing (if one is created) and summarizes them on the terminal (in interactive mode) or in the batch log file (in batch mode). Compile-time diagnostics are preceded by the following:

%PAS{ W / F }DIAGN

A level of W indicates a warning-level error which will not prevent your program from linking or executing. A level of F indicates a fatal error which you must correct for your program to link and execute properly.

The diagnostic messages that the PASCAL compiler can print are listed below. All messages are printed with both number and text. Messages with numbers less than 400 indicate serious syntax errors that you must correct for proper compilation. Messages with numbers greater than 400 indicate the use of VAX-11 PASCAL extensions and illegal compiler options.

1 Error in simple type

The declaration for a base type of a set or the index type of an array contains a syntax error.

2 Identifier expected

The statement syntax requires an identifier, but none can be found.

3 PROGRAM or MODULE expected

The statement syntax requires the reserved word PROGRAM or MODULE.

4 **')' expected**

The statement syntax requires the right-parenthesis character.

5 **':' expected**

The statement syntax requires a colon character.

6 **Illegal symbol**

The statement contains an illegal symbol, such as a misspelled reserved word or illegal character.

7 **Error in parameter list**

The parameter list contains a syntax error, such as a missing comma, colon, or semicolon character.

8 **OF expected**

The statement syntax requires the reserved word OF.

9 **'(' expected**

The statement syntax requires the left-parenthesis character.

10 **Error in type**

The statement syntax requires a data type, but no type identifier is present.

11 **'[' expected**

The statement syntax requires the left square bracket character.

12 **']' expected**

The statement syntax requires the right square bracket character.

13 **END expected**

The compiler cannot find the delimiter END, which marks the end of a compound statement, subprogram, or program.

14 **';' expected**

The statement syntax requires the semicolon character.

15 **Integer expected**

The statement syntax requires an integer, for example, as a statement label.

16 **'=' expected**

The statement syntax requires the equal sign to separate a constant identifier from a constant value or to separate a type identifier from a type definition.

17 **BEGIN expected**

The compiler cannot find the delimiter BEGIN, which marks the beginning of an executable section.

18 **'..' expected**

The compiler cannot find the range symbol (..), which is required between the endpoints of the subrange.

19 **Error in field-list**

The field list in a record declaration contains a syntax error.

20 **',' expected**

The statement syntax requires a comma character.

21 **Empty parameter (successive ',') not allowed**

The parameter list attempts to specify a null or missing parameter, or contains an extra comma. In PASCAL, you cannot omit optional parameters.

22 **Illegal (nonprintable) ASCII character**

The program contains a character that is not a printable ASCII character.

50 **Error in constant**

A constant contains an illegal character or is improperly formed.

51 **':=' expected**

The statement syntax requires the assignment operator.

52 **THEN expected**

The compiler cannot find the reserved word THEN to complete the IF-THEN statement.

53 **UNTIL expected**

The compiler cannot find the reserved word UNTIL to complete the REPEAT statement.

54 DO expected

The compiler cannot find the reserved word DO to complete the FOR statement or the WHILE statement.

55 TO/DOWNTO expected

The compiler cannot find the reserved word TO or DOWNTO in the FOR statement.

58 Invalid expression

The statement syntax requires an expression, but the first symbol the compiler finds is not valid in an expression.

59 Error in variable

A reference to an array element or record field contains a syntax error.

60 ARRAY expected

The compiler cannot find the reserved word ARRAY in the type definition.

97 Strings in excess of 65535 characters not allowed in comparisons

Relational operators cannot be applied to strings longer than 65535 bytes.

98 Parameter count exceeds 255

The number of parameters to a procedure or function cannot exceed 255.

99 End of input encountered before end of program. Compilation aborted.

The end of the input file was encountered before an entire program had been parsed.

100 Array size too large

A declared array is larger than 2,147,483,647 bytes or 2,147,483,647 bits for a packed array.

101 Identifier declared twice

An identifier is declared twice within a declaration section. You can redeclare identifiers only in different declaration sections.

102 Lowbound exceeds highbound

The lower limit of a subrange is greater than the upper limit of the subrange, based on their ordinal values in their base type.

103 Identifier is not of appropriate class

The identifier names the wrong class of data. For example, it names a constant where the syntax of the statement requires a procedure.

104 Identifier not declared

The program uses an identifier that has not been declared.

105 Sign not allowed

A plus or minus sign has occurred before an expression of nonnumeric type.

107 Incompatible subrange types

The subrange types are not compatible according to the rules of type compatibility.

108 File not allowed in variant part

A file type cannot appear in the variant part of a record.

109 Type must not be REAL or DOUBLE

You cannot specify a real value here. Real values cannot be used as array subscripts, control values for FOR loops, tag fields of variant records, elements of set expressions, or boundaries of subrange types.

110 Tagfield type must be scalar or subrange

The tag field for a variant record must be a scalar or subrange type.

111 Incompatible with tagfield type

The case label and the tag field are of incompatible types. These two items must be compatible according to the general compatibility rules.

112 Index type must not be REAL or DOUBLE

Array subscripts cannot be real values; if numeric, they must be integer or integer subrange values.

113 Index type must be scalar or subrange

Array subscripts must be scalar or subrange values, and cannot be of a structured type.

114 Base type must not be REAL or DOUBLE

The base type of this set or subrange cannot be one of the real types.

115 Base type must be scalar or subrange

The base type of this set or subrange must be scalar or subrange values, and cannot be of a structured type.

116 Actual parameter must be a set of correct size

The actual parameter must be of correct size when passed as a VAR parameter.

117 Undefined forward reference in type declaration: <name>

The base type of a pointer was not defined in the TYPE section.

118 VALUE initialization must be in main program

A VALUE initialization statement can appear only in the main program block; you cannot initialize variables in subprograms.

119 Forward declared; repetition of parameter list not allowed

You cannot repeat the parameter list after the forward declaration of a subprogram.

120 Function result type must be scalar, subrange, or pointer

The function specifies a result that is not a scalar, subrange, or pointer type. Function results cannot be structured types.

121 File value parameter not allowed

A file cannot be passed as a value parameter.

122 Forward declared function; repetition of result type not allowed

The result of the function appears in both the forward declaration and in the later complete declaration. The result can appear only in the forward declaration.

123 Missing result type in function declaration

The function heading does not declare the type of the result of the function.

124 F-format for REAL and DOUBLE only

You can specify two integers in the field width (such as R:3:2) for real, single, and double values only.

125 Error in type of predeclared function parameter

A parameter passed to a predeclared function is not of the correct type.

126 Number of parameters does not agree with declaration

The number of actual parameters passed to the subprogram is different from the number of formal parameters declared for that subprogram. You cannot add or omit parameters.

127 Parameter cannot be element of a packed structure

You cannot pass one element of a packed structure to a subprogram; you must pass the entire structure if you want to use it.

128 Result type of actual function parameter does not agree with declaration

The result of an actual function parameter is not of the type specified in the formal parameter list.

129 Operands are of incompatible types

Two or more of the operands in an expression are of incompatible types. For example, the program attempted to compare a numeric and a character variable.

130 Expression is not of set type

The operators you specified are valid only for set expressions.

131 Type of variable is not set

The statement syntax requires a set variable.

132 Strict inclusion not allowed

You must use the <= and >= operators to test set inclusion. PASCAL does not allow you to use the less than (<) and greater than (>) signs.

133 File comparison not allowed

Relational operators cannot be applied to file variables.

134 Illegal type of operand(s)

You cannot perform the specified operation on data items of the specified types.

135 Type of operand must be Boolean

This operation requires a Boolean operand.

136 Set element must be scalar or subrange

The elements of a set must be scalar or subrange types. Sets cannot have elements of structured types.

137 Set element types not compatible

The elements of this set are not all of the same type.

138 Type of variable is not array

A variable that is not of an array type is followed by a left square bracket or a comma inside square brackets.

139 Index type is not compatible with declaration

The specified array subscript is not compatible with the type specified in the array declaration.

140 Type of variable is not record

A period appears following a variable that is not a record type.

141 Type of variable must be file or pointer

A circumflex character appears after the name of a variable that is not a file or pointer.

142 Illegal parameter substitution

The type of an actual parameter is not compatible with the type of the corresponding formal parameter.

143 Loop control variable must be an unstructured, non-floating point scalar

The control variable in a FOR loop must be an integer, integer subrange, or user-defined scalar type; it cannot be a real variable.

144 Illegal type of expression

The specified expression evaluates to a type that is incompatible in this position.

145 Type conflict between control variable and loop bounds

The type of the control variable in a FOR loop is incompatible with the type of the bounds you specified.

146 Assignment of files not allowed

You cannot assign one file to another. Output procedures must be used to give values to files.

147 Label types incompatible with selecting expression

The type of a case label is incompatible with the type to which the selecting expression evaluates. Case labels and selecting expressions must be of compatible types.

148 Subrange bounds must be scalar

You can specify subranges of scalar types only. You cannot specify a real or string subrange.

149 Index type must not be integer

The index type of a nondynamic array cannot be integer, although it can be an integer subrange.

150 Assignment to this function is not allowed

You cannot assign a value to an external or predeclared function identifier.

151 Assignment to formal function parameter is not allowed

You cannot assign a value to the name of a formal function parameter.

152 No such field in this record

You attempted to access a record by an incorrect or nonexistent field name.

153 Error count exceeds error limit. Compilation aborted

The number of errors exceeds 30, the limit set by the ERROR_LIMIT option.

154 Type of parameter must be integer

The actual parameter passed to this function or procedure must be an integer.

155 Recursive %INCLUDE not allowed. Compilation aborted

The %INCLUDE directive cannot include the file in which the directive appears.

156 Multidefined case label

The same case label refers to more than one statement. Each case label can be used only once within the CASE statement.

157 Case label range exceeds 1000

The range of ordinal values between the largest and smallest case labels must not exceed 1000.

158 Missing corresponding variant declaration

In a call to NEW or DISPOSE, more tagfield constants were specified than the number of nested variants in the record type to which the pointer refers.

159 Double, real or string tagfields not allowed

Tag fields cannot be real or string variables, but must be scalar.

160 Previous declaration was not forward

The reiteration of a procedure or function that was not forward declared is illegal.

161 Procedure/function has already been forward declared

The subprogram has already been forward declared.

162 Undeclared procedure or function: <name>

A procedure or function was forward-declared but its block was never declared.

163 Type of parameter must be real or integer

The subprogram requires a real or integer expression as a parameter.

164 This procedure/function cannot be an actual parameter

The specified predeclared procedure or function cannot be an actual parameter. If you must use it in the subprogram, call it directly.

165 Multidefined label

A label appears in front of more than one statement in a single executable section.

166 Multideclared label

The program declares the same label more than once.

167 Undeclared label

The program contains a label that has not been declared.

168 Undefined label: <label>

The program defines a label, but does not use the label in the executable section.

169 Set element value must not exceed 255

The ordinal value of an element of a set must be between 0 and 255.

170 Value parameter expected

A subprogram that is passed as an actual parameter can have only value parameters.

171 Type of variable must be textfile (FILE OF CHAR)

The specified operation or subprogram requires a text file variable as an operand or parameter.

172 Undeclared external file

The program heading specifies an external file that has not been declared at program or module level.

173 Negative set elements not allowed

The value of an integer set element must be between 0 and 255.

174 Type of parameter must be file

The specified subprogram requires a file as a parameter.

175 INPUT not declared as an external file

The program makes an implicit reference to the file variable INPUT, but INPUT is either not declared or has been redeclared at an inner level.

176 OUTPUT not declared as an external file

The program makes an implicit reference to the file variable OUTPUT, but OUTPUT is either not declared or has been redeclared at an inner level.

177 Assignment to function identifier not allowed here

Assignment to a function identifier is allowed only within the function block.

178 Multidefined record variant

A constant tag field value appears more than once in the definition of a record variant.

179 File of file type not allowed

You cannot declare a file that has components of a file type.

181 Array bounds too large

The bounds of an array are too large to allow the elements of the array to be accessed correctly.

182 Expression must be scalar

The expression must specify a scalar value; structured variables are not legal.

183 %IMMED, %DESCR, %STDESCR allowed only in external procedure/function

These extended parameter specifiers are allowed only for procedures and functions which are declared EXTERN.

184 External procedure has same name as main program

Program and procedure names must be unique.

185 Formal procedures may have at most 20 parameters

A procedure name that is defined as a formal parameter can have at most 20 value parameters.

186 Formal procedures may not have dynamic array parameters

You cannot pass a dynamic array as a parameter to a procedure that is itself passed as a parameter.

187 Illegal dynamic array assignment

The program attempts to perform an illegal assignment involving dynamic arrays.

188 Parameter must be scalar and not real or double

The parameters to the predeclared functions SUCC and PRED must be scalar types, and cannot be one of the real types.

189 Actual parameter must be a variable

When you use VAR with a formal parameter, the corresponding actual parameter must be a variable and not a general expression.

190 READLN/WRITELN/PAGE are defined only for textfiles

The predeclared procedures READLN, WRITELN, and PAGE operate only on text files.

191 READ/WRITE require input/output parameter list

The READ and WRITE procedures require at least one parameter; you cannot omit the parameter list.

192 Illegal type of input/output parameter

Arrays, sets, records, and pointers cannot be parameters to the READ and WRITE procedures.

193 Field width parameter must be of type INTEGER

The field width you specify must be an integer.

194 Variable must be of type PACKED ARRAY[1..11] OF CHAR

The DATE and TIME procedures require a parameter of this type.

195 Type of variable must be pointer

The statement syntax requires a variable of pointer type.

196 Type of variable does not agree with tagfield type

The type of a variable in a tag value list is incompatible with the tag field type.

197 Type of parameter must be REAL or DOUBLE

The statement syntax requires a real (single- or double-precision) value.

198 Type of parameter must be DOUBLE

The statement syntax requires a double-precision value.

199 Parameter must be of numeric type

The procedure or function requires an integer or real number value.

200 Parameter must be scalar or pointer and not real

The procedure or function requires an integer, user-defined scalar, Boolean, integer subrange, user-defined scalar subrange, or pointer parameter.

201 Error in real constant: digit expected

A real constant contains a nonnumeric character where a numeral is required.

202 String constant must not exceed source line

The end of the line occurs before the apostrophe that closes a string. Make sure that the second apostrophe has not been left out.

203 Integer constant exceeds range

An integer constant is outside the permitted range of integers (that is, -2**31 to 2**31-1).

204 Actual parameter is not of correct type

The actual parameter is not compatible in type with the corresponding formal parameter.

205 Zero length string not allowed

You cannot specify a string that has no characters.

206 Illegal digit in octal or hexadecimal constant

An octal or hexadecimal constant contains an illegal digit.

207 Real or double constant out of range

A single- or double-precision real number is outside the permitted range -- 0.29*10**(-38) to 1.7*(10**38) for positive numbers and -0.29*10**(-38) to -1.7*(10**38) for negative numbers.

208 Data type cannot be initialized

This variable contains a type that cannot be initialized, such as a file.

209 Variable has been previously initialized

You can specify only one VALUE declaration for a variable.

210 Variable is not array or record type

The VALUE initialization for a variable that is not a record or an array contains a constructor.

211 Incorrect number of values for this variable

The VALUE declaration contains too many or too few values for the variable being initialized.

212 Repetition factor must be positive integer constant

The repetition factor in an array initialization must be a positive integer constant.

213 Type identifier does not match type of variable

The optional type identifier must be compatible with the type of variable to be initialized.

214 Incorrect type of value element

A constant appearing in a VALUE initialization has a type other than that of the variable, record field, or array element to be initialized.

215 RMS record size is out of range

The record size specified in the OPEN procedure call exceeds the maximum.

216 Type OLD is not allowed for this file

You cannot specify OLD for an internal file.

217 %DESCR, %STDESCR not allowed for procedure or function parameters

The only extended mechanism specifier that can be applied to PROCEDURE and FUNCTION parameters is %IMMED.

218 Array must be unpacked

An array parameter to PACK or UNPACK is not unpacked correctly.

219 Array must be packed

An array parameter to PACK or UNPACK is not packed correctly.

220 Packed bounds must not exceed unpacked bounds

The bounds of the packed array exceed the unpacked bounds.

221 %STDESCR not allowed for this type

This mechanism specifier can be applied only to strings and to packed dynamic arrays of CHAR indexed by integer.

222 %DESCR not allowed for this type

This mechanism specifier can be applied only to the predefined scalar types and to unpacked arrays of these types.

223 %IMMED not allowed for this type

This mechanism specifier can be applied only to types that occupy 4 bytes or less, or to PROCEDURE or FUNCTION parameters.

224 %DESCR, %IMMED, %STDESCR not allowed for VAR parameters

You cannot combine the %DESCR, %STDESCR, and %IMMED mechanism specifiers with the VAR specifier.

225 Illegal file attribute specification

You specified an attribute in the OPEN statement that is not recognized by the compiler.

250 Too many nested scopes of identifiers

You can have only 20 levels of nesting. A new nesting level occurs with each block or WITH statement.

251 Too many nested procedures and/or functions

Subprograms can be nested no more than 20 levels deep.

252 Assignment to function not allowed here. Probable name/scope conflict

This error is generated when a function is nested inside a function with the same name.

255 Too many errors on this source line

The PASCAL compiler diagnoses only the first 20 errors on each source line.

259 Expression too complicated

The expression is too deeply nested. To correct this error, you should separately evaluate some parts of the expression.

260 Too many nonlocal labels

The subprogram contains more than 1000 labels that are declared at a higher level, that is, not locally declared.

261 Declarations out of order or repeated declaration sections

The declarations must be in the following order: labels, constants, types, variables, values, and subprograms. Only the main program can contain value declarations.

263 Program segment too large: branch displacement exceeds 32767 bytes

A statement is too large to allow the generation of a branch instruction to span the statement. Use subprogram calls to break the program into smaller units.

300 Division by zero

The program attempts to divide by zero.

302 Index expression out of bounds

The value of the expression is outside the range of the subscripts of this array.

303 Value to be assigned is out of bounds

The value to the right of the assignment operator is out of range for the variable to which it is being assigned.

304 Element expression out of range

The value of the expression is out of range for the array element to which you are assigning it.

305 Dimension specification out of range

The second argument to UPPER or LOWER specifies an array dimension greater than the number of dimensions of the first argument.

306 Index type of dynamic array parameter exceeds range of declaration

The index type of the actual dynamic array parameter extends beyond the range declared in the formal parameter list.

401 Warning: Identifier exceeds nn characters

Identifiers can be any length, but PASCAL scans only the first 15 characters for uniqueness.

402 Warning: Error in option specification

A compiler option is incorrectly specified in the source code.

403 Warning: Source input after "END." ignored

The compiler ignores any characters after the END that terminates the program.

404 Warning: Duplicate external procedure name

Two external procedures or functions have been declared with the same name. They refer to the same externally compiled subprogram.

405 Warning: LABEL Declaration in module ignored

The compiler ignores label declarations at the outermost level in a module.

450 Nonstandard Pascal: Exponentiation

451 Nonstandard Pascal: Value declaration

452 Nonstandard Pascal: OTHERWISE clause

453 Nonstandard Pascal: %INCLUDE directive

454 Nonstandard Pascal: MODULE declaration

456 Nonstandard Pascal: '$' OR '_' in identifier(s)

457 Nonstandard Pascal: Dynamic arrays

458 Nonstandard Pascal: %IMMED, %DESCR, or %STDESCR parameter

459 Nonstandard Pascal: Octal or hexadecimal constant

460 Nonstandard Pascal: Double precision constant

461 Nonstandard Pascal: External procedure declaration

462 Nonstandard Pascal: Octal or hexadecimal data output

463 Nonstandard Pascal: Output of user-defined scalar

464 Nonstandard Pascal: Input of string or user-defined scalar

465 Nonstandard Pascal: Input/output of double precision data

466 Nonstandard Pascal: Implementation-defined type, function, or procedure

A.2 RUN-TIME ERROR MESSAGES

When an error occurs at run-time, VAX-11 PASCAL issues an error message and aborts execution. The run-time error messages appear in the format:

```
%PAS-F-code, Text
```

code

An abbreviation of the message text. Messages are alphabetized by this code.

Text

The explanation of the error.

Some conditions, particularly I/O errors, may cause several messages to be printed. The first message is a general diagnostic specifying the file being accessed (if any) when the error occurred. Then a more specific PASCAL message may be issued to clarify the nature of the error. Finally, a VAX-11 RMS error message may be printed. In most cases, you should be able to understand the error by looking up the first two messages in the list below. If not, refer to the VAX/VMS System Messages and Recovery Procedures Manual for an explanation of the VAX-11 RMS error message.

ATTDISINV Attempt to dispose invalid pointer value xxx at user PC xxx

The DISPOSE procedure was called with an illegal parameter value, probably because of an uninitialized pointer. You should use the NEW procedure to correctly allocate the pointer.

CASSELBOU CASE selector out of bounds at user PC xxx

In a CASE statement, the case selector expression does not correspond to one of the case label values, and no OTHERWISE clause is specified. This message occurs only when the CHECK option is in effect.

ERRACCFIL Error in accessing file nnnnnn

This message identifies the file being accessed when an I/O error occurred.

ERRCLOFIL Error closing file

An error occurred while a file was being closed. This is an internal PASCAL error and should be reported to DIGITAL. Please submit a Software Performance Report (SPR), including an example program if possible.

ERROPECRE Error opening/creating file

An error occurred when the system attempted to open or create the file. The parameters specified in the OPEN procedure (or the defaults, if the OPEN procedure was not used) are probably incorrect for this file.

ERRRESFIL Error resetting file

An error occurred during execution of the RESET procedure. This is an internal PASCAL error and should be reported to DIGITAL. Please submit a Software Performance Report (SPR), including an example program if possible.

ERRREWFIL Error rewriting file

An error occurred during execution of the REWRITE procedure. This is an internal PASCAL error and should be reported to DIGITAL. Please submit a Software Performance Report (SPR), including an example program if possible.

FILBUFNOT File buffer not allocated

The system could not find enough space to allocate the file buffer. This means that too many files are open or too many pointers are in use.

FILNOTCLO Files INPUT and OUTPUT cannot be closed by user

You cannot call CLOSE for the predeclared file variables INPUT and OUTPUT.

FILOUTINV File OUTPUT opened with invalid parameters

You can specify only a carriage control option when you open the predeclared file variable OUTPUT.

FILTYPNOT File type not appropriate

You tried to open for direct access (DIRECT) a file of type TEXT, or a file with variable-length records.

INPCONERR Input conversion error

The system found erroneous input when reading a text file.

INVASGINC Invalid assignment of incompatible dynamic arrays at user pc xxx

The program tried to assign incompatible dynamic arrays to one another. For the assignment to be legal, the arrays must have the same element type and the same upper and lower bounds for each dimension. This message appears only if CHECK is enabled.

LINLENEXC Line length exceeded, line length = xxx

The length of an output line was greater than the maximum allowed by the record size for this file. Check to be sure that you did not omit a call to the WRITELN procedure. If you must write lines of this length, increase the record size in the OPEN statement.

LINLIMEXC LINELIMIT exceeded, LINELIMIT = xxx

The number of lines output to the specified file exceeds the limit. Make sure that the excessive output was not caused by an infinite loop and increase the line limit if necessary.

OUTCONERR Output conversion error

The program tried to write data of an incorrect type to a text file. Make sure that all output values are properly defined.

PROEXCHEA Process exceeds heap maximum size at user PC xxx

The system could not find enough space to allocate storage for a pointer variable. This error is probably caused by an infinite loop that calls the NEW procedure, thus attempting to allocate an infinite amount of heap storage. If the program does not include an infinite loop, your process may actually require more heap storage than the maximum process size allows. In this case, try to make your program smaller; if you cannot, ask your system manager about increasing the maximum process size.

PROEXCSTA Process exceeds stack maximum size at user PC xxx

The system could not expand the stack to make room for the last procedure or function called. This error is probably caused by infinite recursion, where a number of procedures and functions call each other without returning. Make sure that the program does not include this type of logic error. If the program logic is sound, the process may actually require more space than the maximum process size allows. In this case, try to make your program smaller; if you cannot, ask your system manager about increasing the maximum process size.

RESREQACC RESET required before accessing file

You can use the FIND procedure only on files that are open for input.

RESREQREA RESET required before reading the file

The program did not call the RESET procedure before trying to read the file.

REWREQWRI REWRITE required before writing to file

The program did not call the REWRITE procedure before trying to write to the file.

SETASGBOU Set assignment out of bounds at user PC xxx

The program tried to assign an illegal value to a set variable. Make sure that all set assignments specify values that are within the bounds of the set. This message appears only if CHECK is enabled.

SUBASGBOU Subrange assignment out of bounds at user PC xxx

The program tried to assign an illegal value to a subrange variable. Make sure that all subrange assignments specify values that are within the bounds of the subrange. This message appears only if CHECK is enabled.

APPENDIX B

CONTENTS OF RUN-TIME STACK DURING PROCEDURE CALLS

The illustrations in this appendix outline the events that occur during a procedure call, and show the contents of the run-time stack after each event.

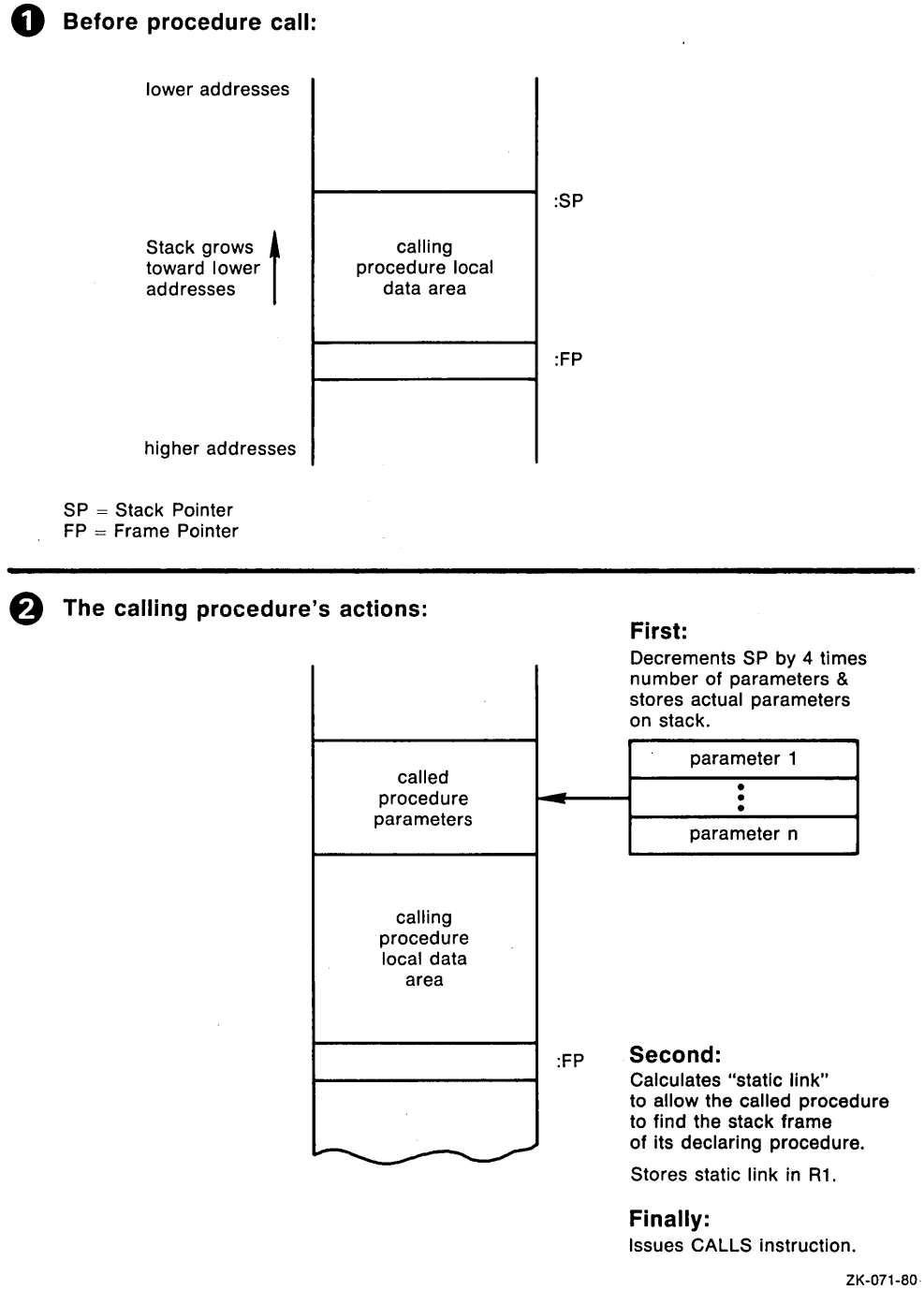
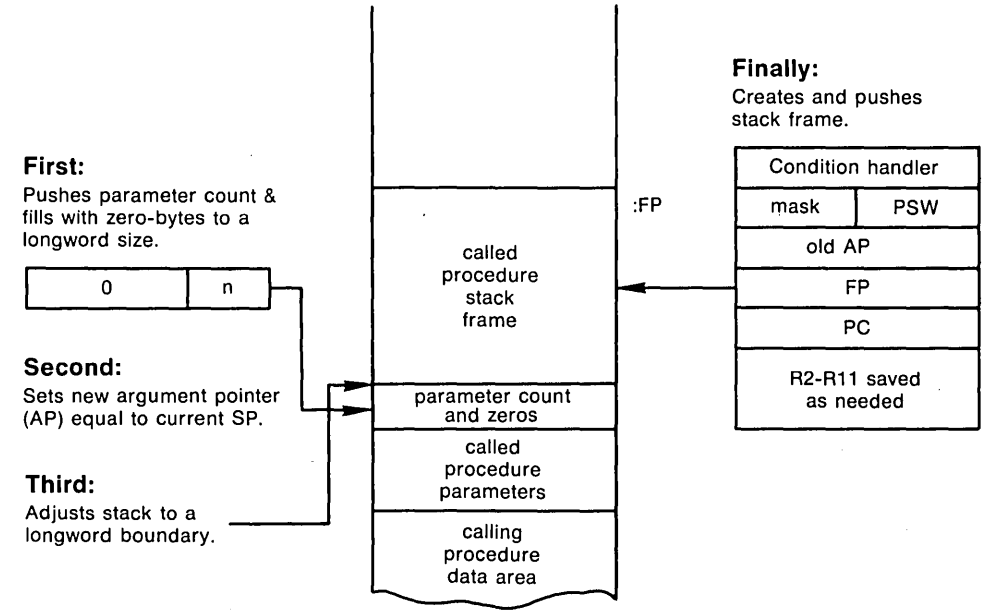


Figure B-1 Contents of Run-Time Stack During Procedure Calls

3 **The CALLS instruction's actions:**



4 **The called procedure's actions:**

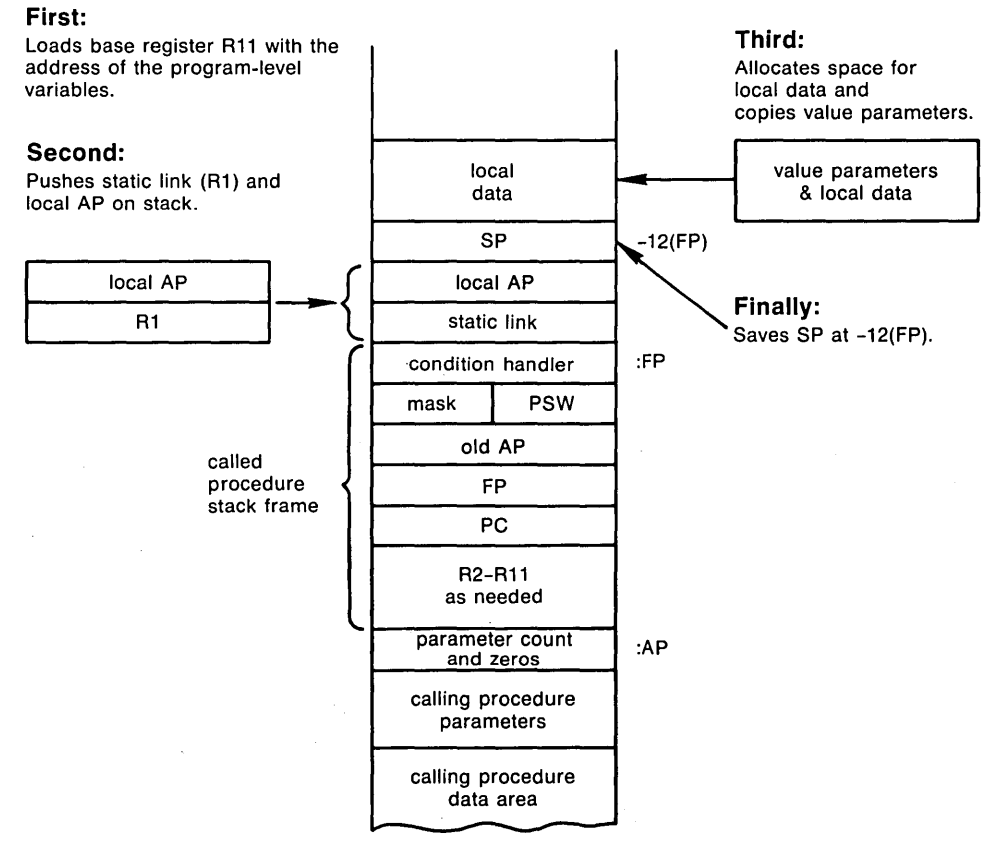


Figure B-1 (Cont.) Contents of Run-Time Stack During Procedure Calls

APPENDIX C

VAX-11 SYMBOLIC DEBUGGER COMMAND SUMMARY

This appendix lists in alphabetical order each debugger command with a format description, a list of qualifiers, a list of parameters, and a brief description of the command's function. The boldface letters indicate the minimum abbreviation that you must type for the debugger to recognize the command name, qualifier, or parameter.

VAX-11 SYMBOLIC DEBUGGER COMMAND SUMMARY

Format	Function
CANCEL **B**REAK[/**ALL**] [address-expression]	Cancels specified breakpoint or all breakpoints.
CANCEL **EXC**EPTION **B**REAK	Cancels effects of SET EXCEPTION BREAK.
CANCEL **M**ODE	Sets all modes and types to their default values for the current language.
CANCEL **MODU**LE [/**ALL**] [module [,module...]]	Cancels specified modules or all modules.
CANCEL **SC**OPE	Sets scope to its default value (PC scoping).
CANCEL **T**RACE [/qualifier] [address-expression] /**AL**L /**BRA**NCH /**C**ALL	Cancels the specified tracepoint or the specified opcode tracing or all tracepoints and opcode tracing.
CANCEL **TY**PE/**OVERR**IDE	Sets the override type to none.
CANCEL **W**ATCH [/**ALL**] [variable-reference]	Cancels the specified watchpoint or all watchpoints.
<CTRL/Y>	Interrupts execution of the program.
DEPOSIT [/qualifier]...variable-reference = data /**AS**CII:length /**BY**TE /**DEC**IMAL /HEXADECIMAL /INSTRUCTION /**LON**G /**O**CTAL /**WO**RD	Stores the specified value(s) at the specified location.
EVALUATE[/qualifier]...expression /**AD**DRESS /**LON**G /**DEC**IMAL /**BY**TE /HEXADECIMAL /**WO**RD /**O**CTAL /**AS**CII	Evaluates expressions or bit ranges, and displays value.
EXAMINE [/qualifier]...addr-expr /**AS**CII:length /**BY**TE /**DEC**IMAL /HEXADECIMAL /INSTRUCTION /**LON**G /**NOSY**MBOLIC /**O**CTAL /**SY**MBOLIC /**WO**RD	Displays the contents of the specified addresses and range of addresses.
EXIT	Ends a debugging session or specifies the end of a command procedure.
GO [address-expression]	Starts or continues program execution.
HELP topic [subtopic ...]	Displays a description of the specified command.
SET BREAK[/**AF**TER:count] address-expression [**DO** (cmd[;cmd...])]	Establishes a breakpoint at the specified address.
SET EXCEPTION **B**REAK	Requests that the debugger treat external exception conditions as breakpoints.
SET LANGUAGE language-name	Sets the current language.
SET LOG file-specification	Sets the file specification of the log file.
SET MODE mode-keyword [,mode-keyword]	Sets the entry/display modes. Mode-keyword can be: **DEC**IMAL, **H**EXADECIMAL, **O**CTAL, **NOSY**MBOLIC, or **SY**MBOLIC.

(continued on next page)

Format	Function
SET MODULE [/**ALL**] [module-name [,module-name] ...]	Adds the symbols in the specified modules or all modules to the debugger symbol table.
SET OUTPUT option [,option ...]	Controls the debugger's output configuration. Option can be **LOG**, **NOLOG**, **TERMINAL**, **NOTERMINAL**, **VERIFY**, or **NOVERIFY**.
SET SCOPE scope [,scope ...]	Specifies scopes to be searched to find a symbol.
SET STEP condition [,condition ...]	Specifies the step conditions. Condition can be **INSTRUCTION**, **LINE**, **INTO**, **OVER**, **SYSTEM**, or **NOSYSTEM**.
SET TRACE [/qualifier] [address-expression] /**BRANCH** /**CALL**	Establishes a tracepoint at the specified address or establishes the specified opcode tracing.
SET TYPE [/OVERRIDE] type-keyword	Sets the default data type for the DEPOSIT and EXAMINE commands. Type-keyword can be **ASCII**:length, **BYTE**, **INSTRUCTION**, **LONG**, or **WORD**.
SET WATCH variable-reference	Establishes a watchpoint at the specified address.
SHOW BREAK	Displays current breakpoints.
SHOW CALLS [integer]	Displays current location and previous calls.
SHOW LANGUAGE	Displays current language.
SHOW LOG	Displays the name of the log file.
SHOW MODE	Displays current entry/display modes.
SHOW MODULE	Lists the modules in the image and shows which are currently included in the debugger symbol table.
SHOW OUTPUT	Displays the debugger's output configuration.
SHOW SCOPE	Displays the current scope search list.
SHOW STEP	Displays current STEP conditions
SHOW TRACE	Displays current tracepoints and opcode tracing.
SHOW TYPE	Displays current default type or override type.
SHOW WATCH	Displays current watchpoints.
STEP [/qualifier] [integer] /**LINE** /**INTO** /**OVER** /**NOSYSTEM** /**SYSTEM** /**INSTRUCTION**	Executes one or a specified number of instructions or lines.
@file-spec	Accepts commands from specified command procedure.

APPENDIX D

VAX-11 SYMBOLIC DEBUGGER AND PASCAL-SPECIFIC MESSAGES

If the debugger encounters an error, it displays a message on the terminal. The general format of a debugger message is:

%DEBUG-l-code, text

l

A severity level indicator. It has a value of I for informational messages, W for warning messages, E for error messages, and F for fatal messages.

code

An abbreviation of the message text; the message descriptions in this appendix are alphabetized by this code.

text

The explanation of the message.

For example:

%DEBUG-W-DIVBYZERO, attempted to divide by zero

Listed below are the messages displayed by the debugger. Each message is accompanied by an explanation of the cause of the error and the recommended user action to correct the error.

BADOPCODE, opcode xxx is unknown

Explanation: The opcode xxx specified in the DEPOSIT command is unknown to the debugger. If the opcode is a valid VAX-11 MACRO opcode, it is an opcode that has a synonomous opcode. These opcodes, such as MOVAF and MOVAL, generate the same instruction. The debugger only recognizes one of them. Severity is warning.

User Action: Specify a valid opcode or specify the opcode's synonym that the debugger accepts.

BADSCOPE, invalid pathname xxx, SCOPE not changed

Explanation: The scope xxx specified in the SET SCOPE command contained a pathname that does not exist. Severity is warning.

User Action: Specify a valid scope.

BADSTARTPC, cannot access start PC = xxx

Explanation: Location xxx is not an accessible address and therefore cannot be executed. This is often caused when a GO command with no address specification is entered after the program has terminated. The debugger tries to execute an instruction at location 0, which is not accessible. Severity is warning.

User Action: Specify a different address specification in the GO command or, if the program has terminated, you can exit from the debugger and initiate the program with the DCL RUN command.

BADTARGET, target location protected, cannot perform deposit

Explanation: The target specified as the location of a deposit is protected. The deposit operation is not performed. Severity is warning.

User Action: Check the target location specified.

BADWATCH, cannot watch protected address xxx

Explanation: A SET WATCH command specified a protected address. Note that you cannot set a watchpoint for a dynamically allocated variable because these variables are stored on the stack. In PASCAL, argument lists for functions and procedures are dynamically allocated on the stack. Severity is warning.

User Action: Do not set a watchpoint for this address.

BITRANGE, bit range out of limits

Explanation: The EVALUATE command specified a bit field that is too wide. Severity is warning.

User Action: The low limit of the bit field is 0 and the high limit is 31; the maximum range is <0:31>.

BRTOOFAR, destination xxx is too far for branch operand

Explanation: The DEPOSIT command specified a branch instruction with a destination, xxx, too far from the current PC. Severity is warning.

User Action: Change a BRB instruction to BRW or a BRW to JMP or specify a closer address.

DBGBUG, internal DEBUG coding error, please report no. nnn

Explanation: An internal debugger error has been encountered. Severity is informational.

User Action: If the error is reproducible, submit a Software Performance Report (SPR) to DIGITAL and, if possible, enclose both a copy of the program being debugged and a logged debugging session that reproduces the error.

DBGERR, internal DEBUG coding error

Explanation: An internal debugger error has been encountered. Severity is error.

User Action: If the error is reproducible, submit a Software Performance Report (SPR) to DIGITAL and, if possible, enclose both a copy of the program being debugged and a logged debugging session that reproduces the error.

DEBUGBUG, internal DEBUG coding error, please report no. nnn

Explanation: An internal debugger error has been encountered. Severity is error.

User Action: If the error is reproducible, submit a Software Performance Report (SPR) to DIGITAL and, if possible, enclose both a copy of the program being debugged and a logged debugging session that reproduces the error.

DIVBYZERO, attempted to divide by zero

Explanation: An expression contained a division by 0. Severity is warning.

User Action: Reformulate the expression.

EXARANGE, invalid range of addresses

Explanation: The range of addresses specified was in the wrong order. The higher address preceded the lower address. Severity is warning.

User Action: Reenter the command with a valid address range.

EXITSTATUS, is xxx

Explanation: The program has exited with the status xxx. See the VAX/VMS System Services Reference Manual for more information about the VAX/VMS exit status codes. Severity is informational.

User Action: None.

EXPSTKOVR, expression exceeds maximum nesting level

Explanation: The expression is too complex. Severity is warning.

User Action: Reduce the nesting of parentheses and simplify the expression.

EXPTDOPR, expected operator but found operand xxx

Explanation: An operator was expected but the value xxx was found instead. Severity is warning.

User Action: Reenter the command with a correct operator.

FILEASGN, assignment of files not allowed

Explanation: A file cannot appear as the target in a DEPOSIT command. Severity is warning.

User Action: None.

FILEVERS, unsupported file version

Explanation: The file version number generated by the compiler is not supported by the debugger. Severity is warning.

User Action: Submit a Software Performance Report (SPR) to DIGITAL.

FLTOVER, floating overflow at or near xxx

Explanation: A floating-point overflow occurred at or near the location xxx. Severity is warning.

User Action: Correct the calculations that caused the overflow to occur.

FLTUNDER, floating underflow at or near xxx

Explantion: A floating-point underflow occurred at or near the location xxx. Severity is warning.

User Action: Correct the calculation that caused the underflow.

FRERANGE, storage package range error

Explanation: Data used to control internal storage allocation is corrupt. Severity is error.

User Action: If DEPOSIT commands or your program has not modified the debugger's storage area, submit a Software Performance Report (SPR) to DIGITAL.

FRESIZE, storage package size error

Explanation: Data used to control internal storage allocation is corrupt. Severity is error.

User Action: If DEPOSIT commands or the user program has not modified the debugger's storage area, submit a Software Performance Report (SPR) to DIGITAL.

ILLINDTYPE, index type must be scalar or subrange

Explanation: An index for an array must be a scalar variable, constant, or variable of a scalar subrange type. Severity is warning.

User Action: Change the index to a scalar variable or a scalar subrange.

ILLINSET, expression preceding 'IN' incompatible with set base type

Explanation: The expression specified before the IN operator is not compatible with the set base type. Severity is warning.

User Action: Specify an expression compatible with the base type.

ILLOPER, xxx is an illegal operator

Explantion: The operator xxx is an illegal one. Severity is warning.

User Action: Specify a legal operator.

ILLPATH, illegal pathname element xxx

Explanation: The pathname element xxx is illegal. Severity is warning.

User Action: Correct the pathname.

ILLREF, xxx is not a legal reference

Explanation: The reference xxx is not understood in the context stated. Severity is warning.

User Action: Make sure the symbols for the module being debugged are present in the active symbol table and that they are correctly spelled when referenced.

ILLSCALAR, scalar variable xxx has an out of range value of yyy (hex zzz)

Explanation: The scalar variable xxx has a value that is out of range. The variable has the value yyy or in hexadecimal zzz. Severity is warning.

User Action: Correct the value assigned to the variable.

ILLSETELEM, set element type must be scalar or subrange

Explanation: A set element must be of a scalar or scalar subrange type. Severity is warning.

User Action: Correct the type of the set element to scalar or scalar subrange.

ILLTAGVAL, tag field aaa has an illegal or uninitialized value of hex xxx, decimal xxx

Explanation: The tag field aaa has an illegal or uninitialized value xxx. Severity is informational.

User Action: Check the state of the object you are referencing.

ILLTYPE, illegal type of operand(s)

Explanation: The type of the operand is illegal for the operator specified. Severity is warning.

User Action: Change the operand.

IMPTERMNO, improperly terminated numeric string nnn

Explanation: Numeric string nnn with radix control did not have a terminating apostrophe or was followed by a nonnumeric character. Severity is warning.

User Action: Terminate the string with an apostrophe.

INCDSTNES, incorrect DST nesting in module xxx, compiler error

Explanation: The compiler is generating debugger information incorrectly. Severity is error.

User Action: Check that your PASCAL source program is correctly written. Submit a Software Performance Report (SPR) to DIGITAL, enclosing your source program.

INDNOTCOMP, index type not compatible with declaration

Explanation: The index type used to subscript an array is not compatible with its declaration. Severity is warning.

User Action: Change the index type.

INITIAL, language is xxx, module set to yyy

Explanation: This message is displayed when the debugger is invoked by the image activator. The language is set to xxx and the module to yyy. Module yyy is the first module linked containing an entry point and language xxx is the language used in that module. Severity is informational.

User Action: None.

INTEGER, this operation only valid on integers

Explanation: The command specified an operation with operands that did not have integer values when integer values are required. Severity is warning.

User Action: Use only operands that have integer values in specified operation.

INTMEMMER, internal memory-pool error at location xxx

Explanation: An error occurred in memory at the location xxx. Severity is fatal.

User Action: Submit a Software Performance Report to DIGITAL, enclosing your source program.

INTOVER, integer overflow at or near xxx

Explanation: An integer calculation overflowed at or near the location xxx. Severity is warning.

User Action: Correct any calculations in the program that may cause an integer overflow.

INVCHAR, invalid character

Explanation: The command contained a character that is invalid in the command's context. Severity is warning.

User Action: Reenter the command.

INVDIM, subscript error, was declared dimension (string)

Explanation: A reference to an array contained either an incorrect number of subscripts or the value of the subscripts is outside of the bounds of the array. The dimension of the array was specified by string. Severity is warning.

User Action: Specify the correct number of subscripts with values in the correct range.

INVDSTREC, invalid DST record

Explanation: An invalid debugger symbol table record was encountered. Severity is error.

User Action: If DEPOSIT commands or your program has not modified the debugger's storage area, submit a Software Performance Report (SPR) to Digital.

INVEXPR, invalid expression detected at or near xxx

Explanation: An invalid expression was detected at or near location xxx. Severity is warning.

User Action: Correct the expression.

INVFLOAT, variable has invalid floating point format

Explanation: A floating-point number has an invalid bit pattern. This error can be caused by depositing a value with a type qualifier into an address associated with a floating-point type variable. Severity is informational.

User Action: Examine the value of the symbol with a type qualifier to override the floating-point format.

INVNUMBER, invalid numeric string 'nnn'

Explanation: The numeric string 'nnn' is invalid in the current language. Severity is warning.

User Action: Specify the value in another numeric format or set the language to one that accepts this type of numeric string.

INVOPR, unrecognized operator in expression

Explanation: An expression contained a character that the debugger does not recognize. Severity is warning.

User Action: Reenter the command with a valid expression.

INVPATH, improperly terminated pathname beginning with xxx

Explanation: The pathname beginning with xxx is not a valid pathname. Severity is warning.

User Action: Check the pathname for errors and reenter the command.

LABNOTFND, search for %label xxx using scope failed

Explanation: The label xxx could not be found using the specified pathname or the current default scope. Severity is warning.

User Action: Reenter the command with the correct pathname.

LASTCHANCE, stack exception handlers lost, re-initializing stack

Explanation: Error in user program caused the exception handling mechanism to fail. Can be caused when the stack is overwritten by the user program or by DEPOSIT commands. Severity is warning.

User Action: Identify and correct the error in your program.

LINNOTFND, search for %line nnn using scope failed

Explanation: The line nnn specified does not exist in the default scope(s). Note that the debugger does not search for a unique line number but only searches the current default scope list. Severity is warning.

User Action: Specify the scope of the line or the correct line number.

LONGDST, too many modules, some ignored

Explanation: There are too many modules in the image for the debugger to keep track of. The excess modules are ignored. You cannot set the module to any of the ignored modules.

User Action: Use the SHOW MODULE command to determine which modules are included. If crucial modules were omitted, relink the image, specifying first the modules needed for debugging.

MAXDIMSN, maximum number of subscripts is nnn

Explanation: An array reference specified too few or too many subscripts. The array has nnn subscripts. Severity is warning.

User Action: Reenter the command with correct number of subscripts.

MISMODBEG, missing Module-Begin record in DST, compiler error

Explanation: The compiler has incorrectly generated information for the debugger.

User Action: Submit a Software Performance Report (SPR) to DIGITAL, enclosing your source program.

MISMODEND, missing Module-End in DST for xxx, compiler error

Explanation: The compiler has incorrectly generated information for the debugger.

User Action: Submit a Software Performance Report (SPR) to DIGITAL, enclosing your source program.

MODNOTADD, no space to add module yyy

Explanation: There was no room to add the modules specified in the SET MODULE command to the symbol table. Severity is informational.

User Action: Use the SHOW MODULE command to show the modules currently in the symbol table and the remaining space, and then use the CANCEL MODULE command to free the needed space.

MULTOPR, multiple successive operators in expression

Explanation: There were two adjacent operators in expressions. Severity is warning.

User Action: Use angle brackets or parentheses (depending on the current language) to separate the operators or enter a valid expression.

NEEDMORE, unexpected end of command line

Explanation: The command entered was not complete. A required part of a command was omitted. Severity is warning.

User Action: Reenter the complete command.

NEEDOPR, expected operator but found operand: xxx

Explanation: The operand xxx was found instead of an operator. Severity is warning.

User Action: Supply an operator.

NILDEREF, attempt to dereference a nil or null pointer value

Explanation: An attempt to dereference a NIL or null pointer value was made. Severity is warning.

User Action: A nil or null pointer cannot be dereferenced, correct the value of the pointer.

NILREF, NIL or an untyped pointer referenced at or near xxx

Explanation: The NIL pointer or untyped pointer was referenced at or near the point xxx. Severity is warning.

User Action: Expressions containing NIL^ cannot be used. Also do not attempt to dereference untyped pointers defined by other languages.

NOACCESSR, no read access to virtual address nnn

Explanation: The debugger does not have read access to the address specified. This error can be caused when an EXAMINE command with no address specification is entered at the beginning of a debugging session. Severity is warning.

User Action: Specify an address that is within the image.

NOACCESSW, no write access to virtual address nnn

Explanation: A DEPOSIT, SET BREAK, or SET TRACE command specified the address nnn, but the debugger does not have write access to the page. The debugger requires write access for setting up breakpoints and tracepoints. Severity is warning.

User Action: You cannot perform the requested operation.

NOANGLE, unmatched angle brackets in expression

Explanation: An expression did not have a closing right angle bracket. Severity is warning.

User Action: Reenter the command with a complete expression.

NOBRANCH, instruction requires branch-type operand

Explanation: A DEPOSIT command specified a branch instruction using an invalid addressing mode as the operand. Severity is warning.

User Action: Reenter the command using a valid branch operand in the destination field.

NOBREAKS, no breakpoints are set

Explanation: The SHOW BREAK command was entered and no breakpoints were set. Severity is informational.

User Action: None.

NOCALLS, no active call frames

Explanation: The SHOW CALLS command was entered and there were no active calls. There are no active calls after your program has terminated. Severity is warning.

User Action: None.

NOCNVT, incompatible types, no conversion

Explanation: A DEPOSIT command specified incompatible data for the variable type. Severity is warning.

User Action: Reenter the command and either specify the correct data type or use a type qualifier.

NOCURLOC, current location not defined

Explanation: The current location (the last location addressed by a SET BREAK, SET TRACE, SET WATCH, EXAMINE, or DEPOSIT command) is not defined. Severity is warning.

User Action: Check that the current location is correctly defined by using one of the above commands.

NODECODE, cannot decode instruction

Explanation: The address specified in the EXAMINE command is not the beginning of a valid VAX-11 instruction. This can be caused by specifying an address that is in the middle of an instruction or is in a data area. Severity is warning.

User Action: Specify an address that contains a valid instruction.

NODELIMTR, missing or invalid instruction operand delimiter

Explanation: A DEPOSIT command specified an invalid instruction operand format. Severity is warning.

User Action: Reenter the command with valid operands.

NODEPOSIT, cannot deposit value

Explanation: The value following the equal sign (=) in a DEPOSIT command cannot be deposited. Severity is warning.

User Action: Change the value to be deposited.

NOEND, string beginning with xxx is missing end delimiter y

Explanation: A DEPOSIT command specified an ASCII or INSTRUCTION string beginning with characters xxx that did not have a terminating apostrophe. Severity is warning.

User Action: Reenter the command with a terminating apostrophe.

NOEXAM, value cannot be examined

Explanation: The value in an identifier cannot be referenced by the EXAMINE command. Severity is warning.

User Action Check that the symbols being referenced are contained in the symbol table.

NOFIELD, xxx is not a field in this record

Explanation: An attempt was made to reference a field that is not defined in the record. Severity is warning.

User Action: Check the field specified.

NOFREE, no free storage available

Explanation: No free storage is available for the debugger to execute the command. Severity is error.

User Action: Free storage by using the CANCEL MODULE command and then reenter the command.

NOGLOBALS, some or all global symbols not accessible

Explanation: The image was linked with the /NODEBUG qualifier and there are no global symbols in the symbol table. This message can also be caused if the image has too many global symbols. Severity is informational.

User Action: Relink the image with the /DEBUG qualifier or, if the message was caused by an overflow condition, remove some of the global symbol definitions from the image (if possible).

NOINSTRAN, cannot translate opcode at location xxx

Explanation: The address specified in the EXAMINE command is not the beginning of a valid VAX-11 instruction. This can be caused by specifying an address that is in the middle of an instruction or is in a data area. Severity is warning.

User Action: Specify an address that contains a valid instruction.

NOINSTPRED, no valid predecessor to an instruction

Explanation: An attempt was made to reference the precedessor of an instruction, where no such predecessor is defined. Severity is warning.

User Action: None.

NOLABEL, routine xxx has no %label nnn

Explanation: The label nnn does not exist in the scope xxx that is specified in the pathname. Severity is warning.

User Action: Specify a valid pathname -- either change the label or the scope.

NOLASTVAL, last value not defined

Explanation: The request for information about the last value (\) cannot be supplied because the last value is not defined. Severity is warning.

User Action: None.

NOLINE, routine xxx has no %line nnn

Explanation: The line nnn does not exist in the scope xxx that is specified in the pathname. This can be caused by a source line number that does not exist or that does not contain executable code in the source program. Severity is warning.

User Action: Specify a valid pathname -- either change the line number or the scope.

NOLITERAL, no literal translation exists for xxx

Explanation: The command attempted to find a literal translation for a value. The debugger does not support this operation. Severity is warning.

User Action: None.

NOLOCALS, image does not contain local symbols

Explanation: All of the modules in the image were compiled or assembled without traceback information. There is no local symbol information in the image. Severity is informational.

User Action: Recompile or reassemble the modules and then relink them.

NONEWCUR, cannot retain new address. '.' lost

Explanation: The current address has been lost, the new address cannot be determined. Severity is informational.

User Action: None.

NONEWVAL, cannot retain new value, '\' lost

Explanation: The current value has been lost so that the last value referenced by a backslash character (\) cannot be determined. Severity is informational.

User Action: None.

NOOPRND, missing operand in expression

Explanation: One or more operands are missing from an expression. Severity is warning.

User Action: Reenter the command with a complete expression.

NOOPRTR, missing operator in expression

Explanation: The expression specified does not contain an operator. Severity is warning.

User Action: Add an operator to the expression.

NOPERMSYM, permanent symbol xxx is not defined

Explanation: The permanent symbol xxx is not defined. Severity is warning.

User Action: None.

NOPRED, logical predecessor not defined

Explanation: The logical predecessor of the identifier or instruction referenced is not defined. Severity is warning.

User Action: None.

NORSTBLD, cannot build symbol table

Explanation: The debugger is unable to build a symbol table because of errors in the format of the image file. Severity is error.

User Action: Relink the image and, if the error is reproducible, submit a Software Performance Report (SPR) to DIGITAL, explaining how the image file was created.

NOSTMT, routine xxx has no statement sss

Explanation: The routine xxx does not contain the specified statement sss. Severity is warning.

User Action: Check the routine or the statement specified.

NOSUCHBPT, no such breakpoint

Explanation: The CANCEL BREAK command specified an address that is not the address of a breakpoint. Severity is informational.

User Action: Use the SHOW BREAK command to find the location of the current breakpoints and then cancel any of these breakpoints.

NOSUCC, logical successor not defined

Explanation: The logical successor of the instruction or identifier referenced is not defined. Severity is warning.

User Action: None.

NOSUCHLAB, no scope exists to look up %label xxx

Explanation: The label xxx is not listed in the current symbol table. Severity is warning.

User Action: Specify the routine that contains this label as the current setting of scope.

NOSUCHLAN, language xxx is unknown

Explanation: The debugger does not recognize the language specified. This message can be caused by mistyping a language that the debugger supports or by entering a language that the debugger does not currently support. Severity is warning.

User Action: Specify a valid language in the SET LANGUAGE command.

NOSUCHLIN, no scope exists to look up %line xxx

Explanation: The line xxx is not listed in the current symbol table. Severity is warning.

User Action: Specify the routine that contains this line as the current setting of scope.

NOSUCHMODU, module xxx is not in module chain

Explanation: The module xxx, specified in the SET MODULE command, does not exist in the image. This message can be caused when a module name has been entered incorrectly or when the image had too many modules for the debugger to handle. Severity is warning.

User Action: Specify a module that is in the image.

NOSUCHTPT, no such tracepoint

Explanation: The CANCEL TRACE command specified an address that was not the address of a tracepoint. Severity is informational.

User Action: Use the SHOW TRACE command to display the current tracepoints and then cancel any of these tracepoints.

NOSUCHWPT, no such watchpoint

Explanation: The CANCEL WATCH command specified an address that was not the address of a watchpoint. Severity is informational.

User Action: Use the SHOW WATCH command to display the current watchpoints and then cancel any that you want to cancel.

NOSYMBOL, symbol xxx is not in the symbol table

Explanation: The symbol xxx cannot be located in the debugger symbol table. This can be caused when the module that defines the symbol has not been added to the debugger symbol table or when a symbol name that is not in the image has been entered. Severity is warning.

User Action: Add the required module to the debugger symbol table with the SET MODULE command or specify the correct symbol name.

NOTALLSYM, cannot initialize symbols for default module

Explanation: The debugger could not put the symbol table information for the first module specified in the LINK command into the symbol table. Severity is informational.

User Action: Use the SET MODULE command to add modules to the symbol table.

NOTARRAY, type of variable is not array

Explanation: The variable being treated as an array has not been defined as one. Severity is warning.

User Action: Check that the correct variable reference is being made.

NOTASTRUCT, xxx was not declared as a structure

Explanation: A VAX-11 BLISS-32 structure reference specified a symbol xxx that was not declared a structure. Severity is warning.

User Action: Reenter the command with a valid symbol reference.

NOTCMP, incompatible types, no conversion on assignment

Explanation: The types of the data and address expression in a DEPOSIT command are incompatible, so the bit pattern of the data was directly assigned to the address expression. Severity is informational.

User Action: None.

NOTCMPEXT, incompatible types, value zero-extended on assignment

Explanation: The types of data and address expressions specified in a DEPOSIT command are incompatible. The bit pattern of the data was too small; the bit pattern was zero-extended on the left before assignment. Severity is informational.

User Action: None.

NOTCMPTRNC, incompatible types, high order bits truncated on assignment

Explanation: The types of data and address expressions specified in a DEPOSIT command are incompatible. The bit pattern of the data was too large; the bit pattern was left-truncated before assignment. Severity is informational.

User Action: None.

NOTCOMMA, ',' expected instead of xxx

Explanation: A comma (,) was expected but the value xxx was found. Severity is warning.

User Action: Change the expression to include a comma (,).

NOTCOMPAT, operands are of incompatible types

Explanation: The operands specified in the debugger instruction do not conform to PASCAL's rules of compatibility. Severity is warning.

User Action: None.

NOTCONTIG, value of xxx is not contiguous

Explanation: An attempt was made to reference an identifier whose value is stored in noncontiguous storage.

User Action: Reference only contiguously stored values in this particular context.

NOTDONE, xxx not yet a supported feature

Explanation: A DEPOSIT command specified an instruction with a literal that the debugger does not yet support. Severity is warning.

User Action: None.

NOTDOTDOT, '..' expected instead of xxx

Explanation: A subrange (..) was expected but the value xxx was found instead. Severity is warning.

User Action: Change the expression to contain a subrange.

NOTIDENT, identifier expected instead of xxx

Explanation: An identifier was expected but the value xxx was found instead. Severity is warning.

User Action: Change the expression to contain an identifier.

NOTIMPLAN, xxx is not implemented at command level

Explanation: The SET LANGUAGE command specified a language that the debugger recognizes but does not yet support. Severity is warning.

User Action: Specify a language that the debugger supports.

NOTINTID, integer or identifier expected instead of xxx

Explanation: An integer or identifier was expected, the value xxx was found instead. Severity is warning.

User Action: Add an integer or identifier in place of or before xxx.

NOTLABEL, 'xxx' is not a label

Explanation: The value xxx is being treated as a label. This value is not a label in the program. Severity is warning.

User Action: Check the value being specified and specify the correct label value.

NOTLINBND, program is not at a line boundary

Explanation: The GO command specified an address that contains threaded code data. The debugger cannot execute starting from this address. Severity is warning.

User Action: Specify an address that contains executable instructions.

NOTLINE, 'xxx' is not a line

Explanation: The value xxx is being treated as a line number. This value is not a line in the program. Severity is warning.

User Action: Check the value being specified and specify the correct line number.

NOTPNTR, variable is not of pointer type

Explanation: The variable being used as a pointer is not defined as one. Severity is warning.

User Action: Change the variable to one of pointer type.

NOTPTR, variable must be of pointer or file type

Explanation: The variable should be a pointer or file type. Severity is warning.

User Action: Specify a variable of pointer or file type.

NOTRACES, no tracepoints are set, no opcode tracing

Explanation: There are no tracepoints or opcode tracing set. Severity is informational.

User Action: None.

NOTRBRACK, ']' expected instead of xxx

Explanation: A right bracket (]) was expected; the value xxx was found instead. Severity is warning.

User Action: Include a right bracket (]) in the expression.

NOTRPAREN, ')' expected instead of xxx

Explanation: A right parenthesis ()) was expected; the value xxx was found instead. Severity is warning.

User Action: Include a right parenthesis ()) in the expression.

NOUNIQUE, SYMBOL xxx IS NOT UNIQUE

Explanation: The symbol specified was not in a default scope and was defined in more than one scope. Severity is warning.

User Action: Specify the scope of the symbol in a pathname or change the default scope.

NOVALUE, reference does not have a value

Explanation: The reference specified in the command does not have a value. Severity is warning.

User Action: Change the reference.

NOWATCHES, no watchpoints are set

Explanation: No watchpoints are set. Severity is informational.

User Action: None.

NOWBPT, cannot insert breakpoint

Explanation: This is an internal debugger error. Severity is fatal.

User Action: Submit a Software Performance Report (SPR) to DIGITAL, enclosing your source program and a log of your debugging session.

NOWOPCO, cannot replace breakpoint with opcode

Explanation: This is an internal debugger error. Severity is fatal.

User Action: Submit a Software Performance Report (SPR) to DIGITAL, enclosing your source program and a log of your debugging session.

NOWPROT, cannot set protection

Explanation: This is an internal debugger error. Severity is fatal.

User Action: Submit a Software Performance Report (SPR) to DIGITAL, enclosing your source program and a log of your debugging session.

NULLSTRNG, null string not allowed in expression

Explanation: A null string value cannot be used in an expression. Severity is warning.

User Action: Do not use a null string in an expression.

NUMOPRNDS, xxx instructions must have nnn operands

Explanation: A DEPOSIT command xxx instruction with an incorrect number of operands. This instruction requires nnn operands. Severity is warning.

User Action: Reenter the command using the correct number of operands with the instruction.

NUMTRUNC, number truncated

Explanation: The debugger truncated the numeric data because it exceeded the length of the data type. Severity is informational.

User Action: None.

OPSYNTAX, instruction operand syntax error

Explanation: The DEPOSIT command contained an instruction with an operand syntax error. Severity is warning.

User Action: Reenter the command with a valid instruction.

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

Explanation: The SET OUTPUT command has set the output conditions to NOTERMINAL and NOLOG; consequently, the output is not displayed on the terminal or written to a log file. The output normally displayed by the debugger will not be available. Severity is informational.

User Action: Use the SET OUTPUT command to send output to the terminal or to a log file.

OUT_LMT, illegal value for parameter, maximum is xxx given value was zzz

Explanation: The value specified was too large to be deposited in the specified address. Severity is fatal.

User Action: Change the size of the value zzz to less than the maximum xxx.

PARSEERR, internal parsing error

Explanation: This is an internal debugger error. Severity is warning.

User Action: Submit a Software Performance Report (SPR) to DIGITAL, enclosing your source program and a log of your debugging session.

PARSTKOVR, parse stack overflow, simplify expression

Explanation: The expression was too complex for the debugger to evaluate. Severity is warning.

User Action: Simplify the expression.

PATHLABEL, %LABEL must precede pathname when language is set to xxx

Explanation: %LABEL must be used in front of a pathname. Severity is warning.

Using Action: Place the word %LABEL in front of the pathname rather than in the pathname.

PATHTLONG, too many qualifiers on name

Explanation: There were too many elements in a pathname. Severity is warning.

User Action: Reduce the number of elements in the pathname.

PCNOTINSCP, PC is not within the scope of the routine declaring symbol

Explanation: A dynamically allocated variable was referenced and the variable was not defined in the scope that contains the current PC. The value of the variable is undefined when you are not currently executing the scope in which it is defined. The debugger uses a value for the variable that may have no relation to the symbol's current value. Severity is informational.

User Action: Reference only a dynamically allocated variable when you are currently executing the scope in which it is defined.

REDEFREG, register name already defined

Explanation: The DEFINE command attempted to redefine a register name. The command is ignored because you cannot redefine register names. Severity is warning.

User Action: None.

RESOPCODE, opcode xxx is reserved

Explanation: Opcode xxx is reserved for use by DIGITAL. Severity is warning.

User Action: None.

ROPRAND, reserved operand fault at or near xxx

Explanation: A reserved operand fault occurred at or near the location xxx. Severity is warning.

User Action: Do not use a reserved operand in your expression and do not deposit such a value into a variable used by your program.

RPARNFOUND, unmatched right parenthesis found

Explanation: A right parenthesis ()) was found but the left parenthesis (() is missing. Severity is warning.

User Action: Include the left parenthesis (().

RSTERR, error in symbol table

Explanation: There is a format error in the symbol table. Severity is error.

User Action: If this is not caused by a user program error or a DEPOSIT command, submit a Software Performance Report (SPR) to DIGITAL enclosing your source program.

SETNOTCOMP, set element types not compatible

Explanation: A value was specified that is incompatible with the type of elements contained in the specified set. Severity is warning.

User Action: Change the value to one that is compatible.

STEPINTO, cannot step over PC = xxx

Explanation: The debugger was unable to step over the routine and executed a step into the routine instead. Severity is informational.

User Action: None.

STGTRUNC, string truncated

Explanation: The debugger truncated an ASCII string because it exceeded the size of the ASCII data type. Severity is informational.

User Action: None.

STRNGVAL, type of string character value must be integer

Explanation: The value specified inside the parentheses in the extended string syntax must be of type INTEGER. Severity is warning.

User Action: Change the type of the value to INTEGER.

STRVALRNG, string character value must be between 0 and 255

Explanation: The value inside the parentheses in the extended string syntax must be between 0 and 255. Severity is warning.

User Action: Change the value to one between 0 and 255.

SUBOUTBNDS, subscript value zzz out of bounds at or near xxx

Explanation: Subscript value zzz was out of bounds at or near location xxx. Severity is warning.

User Action: Check the subscripts in your expression at the location specified by xxx.

SUBOUTVAL, value xxx of subscript yyy out of bounds

Explanation: An attempt to subscript out of the bounds of an array was made. Severity is warning.

User Action: Change the value of the subscript.

SYMNOTACT, symbol xxx not active or not in active scope

Explanation: The symbol xxx is not an active call frame. Severity is warning.

User Action: Check the symbol specified and, if correct, check that you have defined the scope correctly.

SYNTAX, command syntax error at or near xxx

Explanation: The debugger encountered a command syntax error near element xxx. Severity is warning.

User Action: Reenter the command.

UNCOMP, feature not yet supported

Explanation: This feature is not supported at this time. Severity is warning.

User Action: None.

UNDEXP, undefined exponentiation at or near xxx

Explanation: An attempt was made to perform an exponentiation operation at or near location xxx. The exponentiation operation is undefined for the operand specified. Severity is warning.

User Action: Redefine the exponentiation operation using different operands.

UNKNOWNTYP, type of xxx unknown to language zzz

Explanation: Type xxx is not used in the current language. Severity is warning.

User Action: Change to the type to one supported by the current language.

UNMTCHPARN, unmatched left parenthesis found

Explanation: A left parenthesis (() was found, but the matching right parenthesis ()) is missing. Severity is warning.

User Action: Include the right parenthesis ()).

VALOUTBNDS, value assigned is out of bounds

Explanation: The value assigned to a variable of a subrange type is out of bounds. Severity is informational.

User Action: None.

VERIFYICF, xxx indirect command file yyy

Explanation: The debugger is verifying an indirect command file. This message is displayed before the command file is executed and after all the commands have been displayed. Severity is informational.

User Action: None.

WRONGLANG, language is not xxx

Explanation: The language used to evaluate the last expression or variable reference is not the current language. Severity is warning.

User Action: Do not change languages between the definitions of last value (\), logical predecessor (^), current location (.), logical successor (RET), and their use.

WRONGVAR, field xxx is not in current variant yyy associated with tag field xxx

Explanation: Field xxx is not in the current variant.

User Action: Check your program to verify the expected results.

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z

VAX-11 PASCAL
User's Guide
AA-H485B-TE

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

Please cut along this line.

— — 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 03061

— — Do Not Tear - Fold Here —

Cut Along Dotted Line

digital

PRINTED IN USA